6800/6809 BASIC COMPILER

V1.4

USER'S MANUAL

SOFTWARE DYNAMICS
2111 W. Crescent, Suite G, ▲ Anaheim, CA 92801

SOFTWARE DYNAMICS

6800/6809 BASIC COMPILER V1.4

10TH PRINTING

NOTICE

This manual describes version 1.4 of the Software Dynamics BASIC compiler system for 6800/6809 microprocessors. The information given in this manual has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies.

Software Dynamics reserves the right to change specifications without notice.

```
**********************************************************************
** SD software is sold on a single copy per computer basis, and is **
** covered by U.S. copyright laws.  Unless a written exception is **
** obtained from SD, the software must be used only on the single **
** computer whose unique, SD-assigned serial number matches that **
** for which the software was purchased.  Copies for any purpose **
** other than archival storage, or use on other than the assigned **
** serial numbered CPU are strictly prohibited.                    **
** USE OF THIS MANUAL OR THE SOFTWARE IT DESCRIBES CONSTITUTES **
** AGREEMENT BY THE USER TO THESE TERMS.                           **
**********************************************************************
```

SDOS is a registered trademark of Software Dynamics.

This manual was produced by TYPE, a document-producing program written in SD BASIC.

TABLE OF CONTENTS

TABLE OF CONTENTS

SOFTWARE DYNAMICS

6800/6809 BASIC COMPILER V1.4

10TH PRINTING

NOTICE

This manual describes version 1.4 of the Software Dynamics BASIC compiler system for 6800/6809 microprocessors. The information given in this manual has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies.

Software Dynamics reserves the right to change specifications without notice.

```
*********************************************************************
** SD software is sold on a single copy per computer basis, and is **
** covered by U.S. copyright laws.  Unless a written exception is **
** obtained from SD, the software must be used only on the single **
** computer whose unique, SD-assigned serial number matches that **
** for which the software was purchased.  Copies for any purpose **
** other than archival storage, or use on other than the assigned **
** serial numbered CPU are strictly prohibited.                    **
** USE OF THIS MANUAL OR THE SOFTWARE IT DESCRIBES CONSTITUTES **
** AGREEMENT BY THE USER TO THESE TERMS.                           **
*********************************************************************
```

SDOS is a registered trademark of Software Dynamics.

This manual was produced by TYPE, a document-producing program written in SD BASIC.

PROGRAM ORGANIZATION

BASIC is a procedure-oriented language: the user expresses the activity he desires the computer to perform, in a set of explicit commands to perform computations and make decisions.

Each of the commands is called a statement. BASIC has some 4Ø different kinds of statements and 4Ø different functions; the form and function of each is individually discussed, below.

A statement list is a set of executable statements separated by the backslash ("\") character. An optional <CR> character is allowed after each backslash separating statements, so a statement list may span several physical text lines. Note that a REM statement, if included in a statement list, is always the last statement in that statement list. A single statement (if not trailed by a "\") is a statement list.

A "line" is a statement list, followed by a <CR> character (the traditional BASIC definition of a line is a statement followed by a <CR>). Note that with this definition a "line" can span several physical text lines.

A "block" is a statement that has an embedded set of lines. A block is introduced by certain statement formats, and terminated by a keyword that depends on the introducing statement. The set of lines embedded in the statement is called the block body (an example of a block in traditional BASIC is FOR-NEXT). Execution of the block body is controlled by the introductory statement. Note that a block can be part of a line. Wherever "blockbody" is shown in this manual, it may be replaced by a statement list (see BLOCK BODIES).

A BASIC program consists of a set of lines. Traditionally, each line is numbered to indicate the normal sequence in which the lines are executed. These line numbers appear at the left end of the line and may be any value from 1 to 65535. Good programming practice dictates that line numbers be separated by some numeric distance, say 1Ø, so that if programming errors are found, or changes made to the program, new lines with numbers in between those which already exist can be created. With SD BASIC, line labels may also be used, as well as line numbers. A line label is any sequence of up to 32 letters or digits, starting with a letter, (i.e., LABEL is a valid label). A label name may not be identical to a reserved keyword (see VARIABLES). When a line label is used to "number" a line, a label must be followed by a ":" character. A line label may be on a line by itself, whereas a line number may not. Heavy use of labels makes programs more readable, and thus more maintainable. Throughout the rest of this document, the term line number or line label are used interchangeably. Line numbers are only needed if the line is referenced by another part of the BASIC program.

Example:

```
10 REM **** PRIME NUMBER CHASER ****
20 REM PRINTS OUT FIRST 100 PRIME NUMBERS
30 DIM PRIMES[100],CANDIDATE/3/,NPRIMES/1/
35 PRINT "Prime Finder"
40 LET PRIMES[NPRIMES]=2
LOOKFORANOTHERPRIME:
    FOR PRIMESELECTOR=1 TO NPRIMES
    X=PRIMES(PRIMESELECTOR)
    LET Q=INT(CANDIDATE/X)
    IF Q<X THEN FOUNDNEWPRIME
110 IF Q*X=CANDIDATE THEN 120
105 NEXT PRIMESELECTOR
FOUNDNEWPRIME: LET NPRIMES=NPRIMES+1\PRIMES[NPRIMES]=CANDIDATE
120 LET CANDIDATE=CANDIDATE+2
    IF NPRIMES <> 100 THEN GOTO LOOKFORANOTHERPRIME
    FOR PRIMESELECTOR = 1 TO 100
    PRINT PRIMES(PRIMESELECTOR)
    NEXT PRIMESELECTOR
    PRINT 'All Done!'
200 STOP
    END
```

The above program is numbered in a conventional way for BASIC programs, with the exception of some lines without numbers, two lines with a label, and one line with a line number out of order (see below). Note that when several statements are "grouped" in the same line (line FOUNDNEWPRIME), they must be separated by a "\" (backslash) character. A FOR-NEXT block appears in the program. The lines between FOR and NEXT comprise the body of the FOR-NEXT block.

When a BASIC program is executed, execution starts with the first statement in the first line (the statement at the top of the page of a listing of the program). The statements within a line are executed from left to right if there are more than one.. When a line is completely executed, control flows to the next line down the page (of the program listing), and its component statements are executed from left to right. Certain statements change the flow of control explicitly (i.e., GOTO, GOSUB, NEXT, ON, IF, etc.). If control flow is redirected, SD BASIC executes lines sequentially from the new target point until control flow is changed again. Note that control flow is NOT directed in sorted line-number order as in conventional BASIC programs, but rather in top-to-bottom of page order. This is, however, compatible with standard BASIC programs listed in line-number order.

VALUES

BASIC programs can operate on two kinds of data: real numbers and character strings. A specific real number or string is called a value.

Number values used by BASIC are decimal (floating point) 9 digit precision numbers (decimal is used to facilitate business applications). Numbers used for dollar amounts between plus or minus 1ØØ million dollars are kept to 1Ø digit accuracy (exact to the penny).

Numeric values are limited to the range of plus or minus .9999999999 times 1Ø to the plus or minus 126.

Strings are groups of 8 bit data items (bytes), which normally contain ASCII codes for letters, digits, punctuation, etc. A string value may be from zero to 65534 (not 65535) characters in length.

CONSTANTS

Constants are the means by which the programmer introduces a particular value into the program, permanently. Note that line numbers are not really constants, since they only serve to label a line, not to introduce a value into the program.

Number constants consist of digit strings with an optional exponent specification, and represent real values in the program. At least one digit must be given. A decimal point can be placed anywhere in the digit string. The exponent is specified by writing "E" (or "e") followed by "+" or "-" or nothing, followed by one to three digits for the exponent value itself.

Examples of numeric constants:

```
5       2.7     ØØØØ3ØØ        .9999999999E126
.ØØ7    6E-2    .14Ø1e+76      12E12
```

BASIC also accepts positive integer hexadecimal constants in the range Ø to 65535. The form of a hexadecimal constant is a colon followed by one to four hexadecimal digits (Ø-9, A-F or a-f). A hexadecimal constant may be used anywhere a numeric constant may be used.

Examples of hexadecimal constants:

```
:Ø      :ABC4   :2F     :4f3
```

Two special constants, named TRUE and FALSE, represent the values 1 and Ø, respectively.

String constants consist of a quoted sequence of characters which do not contain the quote. The quote character may be either " or ', but it must be the same at both ends of the string constant. The string value represented is the sequence of ASCII-coded characters which comprise the string body (everything but the quotes). Upper and lower case characters are preserved exactly as written in the body of the string. A single quote may appear in a string constant which is delimited by double quotes, and vice-versa. An end of line character <CR> may not appear in a string constant. The null or empty string is written as '' or "". A string constant may not exceed 127 characters in length including the quotes.

Examples:

```
"ABC"   'BE"FG'        ".14Ø1E+76"     ''      "can't"
```

VARIABLES

BASIC allows the programmer to name quantities which can change. These named quantities are called variables. The name itself is the variable name.

BASIC supports two kinds of variables: numeric and string. Numeric variables are used to represent quantities and can hold any value specified in the section on constant numbers. String variables are used to deal with varying length groups of characters (or bytes), and can hold any value as specified in the section on string constants. String variables are limited to 65534 bytes in length.

Variable names are composed of letters and digits; the first character of a name must be a letter. Lower case letters are treated as being identical to upper case letters in variable names.

Examples of legal numeric variable names:

```
X, B7, INTEREST, Rate, A7773X.
```

The length of a variable name is limited to 32 characters by the assembler. The name of a variable must not be the same as any keyword (statement, function name, etc.), or a syntax error will result (i.e., THEN is not a valid variable name). A list of keywords may be found in the section on KEYWORDS (note that keywords may also be written using lower case).

String variable names require that a "$" character be the last character of the variable name. The numeric variable whose name is the same as a string variable name (except for the "$") is a completely different object from the string variable. String variables have two associated lengths: the current LENgth, which is the number of characters currently held by the string, and MAXLEN, which is the maximum (DIMensioned) length of the string. This difference is subtle but very important; failure to understand the difference will cause many mysterious string subscript errors.

Examples of legal string variable names:

```
CUSTOMERNAME$, IN27F$, BUF2$, TEXT$
```

SUBSCRIPTING

A vector is a variable which represents a list of numeric values. If a vector is named V, the first value in the vector is named (denoted) V[1], the second value is denoted V[2], etc. The value inside the [ ]s is called the subscript. A subscript value may be specified by an expression to allow computation of the desired element of the vector. SD BASIC allows Ø as a subscript on numeric vectors. BASIC accepts ( ) interchangeably with square brackets.

An array is a variable which represents a rectangular matrix of values. The upper left hand corner is named A[Ø,Ø]; this element is in the zeroeth row, zeroeth column. The 2nd element of the 1st row is A[1,2], etc. The value in the Nth row, Mth column is named A[N,M]. N and M may be expressions which compute the selected row and column. N or M may be zero.

While SD BASIC does not directly support 3 or higher dimensional arrays, they may be transparently simulated using the Uniform Reference facility, as outlined in that section.

Strings can be selected in their entirety, or in portions. The notation stringname$[exp1,exp2] means select the substring of the named string starting in the exp1 position of the string for exp2 characters. If B$ has the value "HELLO" at the moment, B$[3,2] is the string value "LL". The substring selected must not overlap the end of the current string value (i.e., exp1<1 or exp1+exp2>current length of the string), or a subscript error will occur. If exp2 is zero, no subscript error can occur.

The notation stringname$[exp] or stringarrayname$(exp1)[exp] is called a byte subscript, and means the "exp"th slot of the string; this form can appear only in numeric expressions and represents the numeric value of the expth byte of the string (as opposed to a single character string). Exp can be from 1 to the DIMensioned size of the string; the current length of the string has no effect on byte subscripts. Zero may not be used as a byte subscript.

A string array is a variable which represents a vector of strings (a better name would have been string vectors, but historical reasons prevent changing it). Each slot of a string array holds a variable length string. The number of strings in a string array is specified in a DIM statement. The notation stringarrayname$[exp] selects the "exp"th string of the array; exp must round to a value greater than or equal to 1.

A substring selector or a byte subscript may be appended to the string selector to select a portion of that string, as desired.

A typical string array might contain one sentence in each slot, with the array representing a limerick. Then the string array would have 5 slots; typical filler material might be:

LIMERICK$[1] has "There is a nice compiler BASIC"
LIMERICK$[2] has "with features that make it like magic"
LIMERICK$[3] has "Programs are easy to read"
LIMERICK$[4] has "and run with great speed"
LIMERICK$[5] has "making other BASICs seem tragic."

Then LIMERICK$[3](14,4) selects the substring "easy" and LIMERICK$[3](1) contains the value :5Ø.

Any variable which can be subscripted must have the maximum bounds specified in a DIM(ension) or COMMON statement.

If any subscript value used is not an integer, BASIC will round it to the closest integer (i.e., it will use INT(value+.5) instead of the value).

SD BASIC allows the result of a string function (see DEF and BUILT-IN FUNCTIONS) to be subscripted. Example:

```
NUM$(35)[3,1]
```

gives the string "5" since NUM$(35) gives " 35".

EXPRESSIONS

Key to the operation of BASIC programs is the ability to compute new values based on old values. Specification of such a computation is done by writing an expression, which looks like an algebraic formula.

An expression is a sequence of constants, variable names and operators (such as add or multiply). BASIC allows two kinds of expressions: those which compute numeric values, and those which compute string values.

Numeric expressions may contain only references to functions that return numeric results, numeric variables (or entries in vectors, arrays, and strings), real constants, or hex constants. The allowed operators are listed below with their function:

OPERATOR	USE	OPERATION
+	A+B	Computes the sum of A plus B
-	A-B	Computes the difference of A minus B
*	A*B	Computes the product of A times B
/	A/B	Computes the quotient of A divided by B
&	A&B	Computes the logical bit product of A and B
!	A!B	Computes the logical bit "or" of A and B
XOR	A XOR B	Computes the logical exclusive or of A and B
**	A**B	Computes the value of A shifted left [B>Ø] or right [B<Ø] bits
^	A^B	Computes the value of A raised to the power B
-	-A	Computes the negative of the value of A
F( )	F(A)	Computes the value of the function F applied to A

For manipulating dollar amounts, arithmetic (+,-,*,/) is accurate to 1Ø digits if the operands and the result have magnitudes in the range .Ø1 to 99999999.99.

Logical operators (& ! XOR **) allow only integer operands in the range Ø to +65535 (exception: the right operand of ** may be a negative integer); the result of an operation such as & is the bitwise "AND" of the binary equivalent of the numbers.

Examples:

```
Ø&17 = Ø          1&:11 = 1        2&17 = Ø          :1F&16 = 16
2!1 = 3           3!7 = 7          :7 XOR 5 = 2      1**1Ø = 1Ø24
7**-2 = 1         7**-3 = Ø        12**2 = 48        1ØØ**25 = Ø
32768**-1 = 16384
```

BASIC has a set of built in functions (explained in the section on FUNCTIONS). The programmer may also define his own functions (see DEF).

One can cause a function to be applied to a value by simply writing the function name, a left parenthesis, then an expression representing the value, and a right parenthesis, i.e. COS(PI+2). Functions that accept multiple arguments are invoked by writing the function name, a left parenthesis, a list of expressions (one for each argument) separated by commas, and a right parenthesis. Functions that require only a single argument do not require the parenthesis (i.e., SIN X is legal).

Evaluation of an expression is based on operator precedence, with operations performed in the following order:

```
1ST:    - (negate) and functions
2ND:    ^ (exponentiation)
3RD:    & * / **
4TH:    XOR ! + - (subtract)
```

so that 3*4+2 gives the value 14, not 18. Parentheses may be used to override precedence to obtain any desired order of evaluation, (i.e., 3*(4+2) gives 18, not 14).

Examples of expressions:

```
2
WHALE+PORPOISE
COS(PI/ATN(BETA[3]))*19
:46-ADDRESS*2
VALUE&:7+"Ø"
INLINE$(I) - ASC 'A' + OFFSET
```

String expressions may contain any sequence of constant strings, string variables, and substring references separated by the operator "CAT", which concatenates strings together. Concatenated strings may not exceed the size of the concatenation buffer (256 by default). Parentheses may not be used in string expressions to change the order of concatenation (but they may be used in subscript computations). If more than one string temporary is present within a statement, "CAT" may be used in only one of those computations which generate temporaries:

```
A$=A$ CAT B$ CAT C$
```

is legal, but, beware:

```
RENAME A$ CAT '.EXT', B$ CAT '.EXT'
```

is illegal.

"ABC" CAT 'DEF' produces the value "ABCDEF". If A$ contains "ABCDEF", then

```
A$[1,3] CAT '*' CAT A$[4,3]
```

gives "ABC*DEF" as its value.

CONDITIONAL EXPRESSIONS

A conditional expression is used to determine the truth or falsity of a relation between two or more values. Conditional expressions are usually found in statements which perform operations conditionally, such as IF and WHILE statements.

Conditional expressions are composed of relations, and logical combinations of relations. The simplest conditional expression is a relation. Relations allow two values to be compared. Possible relations are:

Numeric Values:	(EXP1 and EXP2 are numeric expressions)
EXP1 = EXP2	True if the value of EXP1 equals EXP2
EXP1 <= EXP2	True if EXP1 is less or equal to EXP2
EXP1 >= EXP2	True if EXP1 is greater or equal to EXP2
EXP1 < EXP2	True if EXP1 is strictly less than EXP2
EXP1 > EXP2	True if EXP1 is strictly greater than EXP2
EXP1 <> EXP2	True if EXP1 is not equal to EXP2
EXP	(Interpreted as a relation) means "EXP<>Ø"

String Values:	(EXP1 and EXP2 are string expressions)
EXP1 = EXP2	True if string value of EXP1 exactly equals the value of EXP2, both in content and length
EXP1 <= EXP2	True if EXP1 alphabetically precedes EXP2 (according to ASCII character coding) or EXP1 = EXP2 exactly. Note that strings being compared are not blank extended to the right, and that "ABC" < "ABD" and "ABC" < "ABC ".
EXP1 >= EXP2	True if EXP2 alphabetically precedes EXP1, or EXP1 = EXP2 exactly.
EXP1 < EXP2	True if EXP1 alphabetically precedes EXP2.
EXP1 > EXP2	True if EXP2 alphabetically precedes EXP1.
EXP1 <> EXP2	True if EXP1 is not the same as EXP2, in either length or content.

One cannot compare a string expression with a numeric expression. Also, if "CAT" is used as an operator in EXP1, it cannot be used in EXP2 and vice-versa (see caveat on "CAT" under "EXPRESSIONS").

A condition can be any combination of relations, using the operators "AND", "OR", the logical function "NOT" and parentheses.

A conditional expression of the form A AND B is true if condition A is true, and condition B is true. A conditional expression of the form A OR B is true if condition A is true, or condition B is true. A conditional expression of the form NOT A is true only when the condition A is false. AND has precedence over OR so that

```
A<B AND B<C OR S=Ø
```

is true if S=Ø or both A<B and B<C. Parentheses may be used to change the order of evaluation:

```
A<B AND ( B<C OR S=Ø )
```

is true if both A<B and either B<C or S=Ø. The "NOT" operator can be applied to invert any condition: NOT A<B is the same as A>=B. NOT has higher precedence than AND or OR, so parentheses are needed to invert a complicated condition.

```
NOT( A<B AND B>2 )
```

is the same as:

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

Conditions are evaluated from left to right until their truth is determined. Once the truth value of a condition is determined, the rest of the condition expression is not evaluated, so that:

```
S=Ø OR B/S=2
```

can never give a division by zero error (this kind of test is also useful when checking for an illegal subscript) and also speeds up program execution.

The result of a condition can be used wherever a condition is allowed, or as an element of an expression. When used in an expression, a true condition gives the value 1 (TRUE), and a false condition gives the value Ø (FALSE). So

```
12*(3>4)
```

gives the value Ø since 3>4 gives the value Ø, while

```
-6*(2<3)
```

gives -6 since 2<3 gives 1.

When used as part of a conditional expression, a simple expression is interpreted as "expression<>Ø".

Example:

```
X^2 AND NOT Y
```

is interpreted as

```
X^2<>Ø AND NOT Y<>Ø
```

STATEMENTS

This section describes the format and function of statements in the Software Dynamics version of BASIC. All statements start with a keyword (with the exceptions of optional LET or CALL keywords) and end with the <CR> or a backslash (\).

The statements are listed in order of probable utility.

PRINT

The simple PRINT statement is used to cause printing of values or character strings on the terminal.

The general form is:

```
PRINT list-of-print-fields
```

The print fields can be numeric or string expressions. The print field entries are separated from one another by "," or ";", which affect how spacing is to be performed between the printed values (see below).

Examples:

```
1Ø PRINT A

2Ø PRINT "SUM IS"; A+B, "PRODUCT IS "; A*B

3Ø PRINT "LEFT SUBSTRING: "; A$[1,25]

4Ø PRINT "JUST THIS STRING"

5Ø PRINT
```

The PRINT statement causes the values in the print fields to be converted to a readable form and printed on the terminal. The values are printed out on the terminal from left to right in the same order as they appear in the PRINT statement itself.

String values are printed exactly as the ASCII equivalent of the string contents, i.e., "ABC" is printed as ABC. No insertions or deletions are made by BASIC; if the string contains control characters, they are copied directly to the I/O interface routines. However, the operating system under which BASIC is running may have some conventions regarding such control characters.

Numeric values are printed in a form designed for user convenience, according to the following rules:

If the value is an integer and is less than 1E1Ø:

    sign dddddddddd

Where "sign" is either a single blank or a "-" sign, and "dddd" are digits. Leading zeros are suppressed. The value -1Ø*22 is printed as -22Ø; the value 17 is printed as space 17. Zero is printed as space Ø.

If the value is not an integer and is greater than or equal to 1 in magnitude, and is less than 1E1Ø:

    sign dddddddd.dddddddd

Leading zeroes (when to the left of the decimal point) and trailing zeroes (when to the right of the decimal point) are suppressed. At most 1Ø significant digits are printed. "." is a decimal point.

If the magnitude of the number is less than 1 and greater than or equal to 1E-6:

    sign .dddddddddddddd

Trailing zeros are not printed.

If none of the above conditions describe the number value, "E" notation is used:

    sign.ddddddddddE esign xxx

The first digit to the right of the decimal point is nonzero. Trailing zeros are suppressed. The "E" after the digit string represents a literal "E", and "esign" represents "+" or "-". "xxx" represents a three digit, leading zero supressed, exponent. The printed form corresponds to a value of sign.dddddddddd*1Ø^(esign xxx).

Field separators may be either "," or ";". The comma causes tabbing between fields printed; it forces the terminal to space (at least once) to the column such that the column number modulo 18 is one (i.e., there are column boundaries at 19, 37, 55, 73, etc.). The semicolon causes a space to be printed if the value to its left is numeric; if the value to its left is a string, the semicolon prints nothing.

The statement:

```
60 PRINT A,B,C,D
```

produces a tabular output with the value of A in the first column, B in the second column, C in the third, etc.

```
70 PRINT A;B;C;D
```

produces tightly spaced output with each number being separated from its neighbor by a single space (two spaces if all the values printed are positive).

```
80 PRINT A;B,D;E
```

produces values of A and B tightly packed, then a tab to the values of D and E tightly packed.

```
90 PRINT "ABC"; 5
```

produces ABC space 5.

```
100 PRINT "ABC";S$
```

produces ABCDEF if S$ has the value "DEF".

A print field may contain TAB(exp) instead of a string or a numeric expression. This causes the terminal to space until the column specified by the expression is reached. TAB(1) refers to the leftmost print column on a line. If the expression specifies a column less than the current position of the print head, no spaces are produced. TAB() must always be followed by a ";" separator. The TAB function may only be used in a PRINT statement.

```
110 PRINT A;TAB(10);B;TAB(20);C;TAB(30);D
```

produces a columnar output with 10-space wide columns.

The PRINT statement normally causes a set of values to be printed, followed by a carriage return (i.e., further output is on a new line). The carriage return may be suppressed by ending the print statement with a ";" or "," (which have the same meaning as above). Further printing by other print statements will then resume from where the continuing PRINT left off.

```
120 PRINT A,B
130 PRINT C,D
```

will produce two lines of two column output, while

```
140 PRINT A,B,\REM NOTE TRAILING COMMA AFTER B
150 PRINT C,D
```

will produce output identical to that of

```
160 PRINT A,B,C,D
```

A print statement with no print fields simply causes a carriage-return character to be sent to the terminal. The sequence:

```
170 PRINT "RESULT IS";\REM NOTE SEMICOLON AFTER STRING
180 PRINT SQR(X);
190 PRINT
```

does the same as:

```
200 PRINT "RESULT IS";SQR(X)
```

PRINT USING

The PRINT USING statement is used to perform formatted output of numeric values. The format is specified as a character string; the PRINT USING statement specifies the list of values to be printed according to the format. The form of a PRINT USING statement is:

```
PRINT USING format-string, list-of-values ;
```

The format-string can be a string constant, a string variable (with optional subscripts), or the line number of a FORMAT statement (which contains the format string).

The list-of-values is a set of string or numeric expressions separated by commas. The last item of the list may not be followed by a comma, but may be followed by a semicolon. The print list may be empty, which simply causes output of the contents of the format string.

A format string is a string of ASCII characters containing number formats. The number formats cause numeric values to be printed in a controlled way. The following characters are the only characters that may be used in a number format:

Character	Usage
$	Causes floating dollar sign to be printed
-	Causes sign of number to be printed
#	Causes a digit to be printed
.	Forces printed number to be aligned with decimal point
^	Specifies exponential notation be used

Number formats are character sequences composed of an optional dollar sign, optional minus sign, optional decimal point, hash marks (#), and an optional group of 3 to 5 carets (^). The hash marks indicate digit positions; the decimal point indicates a forced alignment for the decimal point, and the carets force "E" (exponential) notation, and specify the number of printed exponent digits. The "-" sign is used if the number needs a place for a sign ("-" or blank). The dollar sign indicates the need for a floating '$' character to be output immediately preceding the sign of the number. "####" means "4 digit integer" (only positive numbers allowed!). "-##.##" means "signed 4 digit number, two digits to the left and two to the right of the decimal point". "###.##-" means a "5 digit signed number, sign following the last digit". "#.##^^^^^" means "3 significant digits (conventional scientific notation) with 3 digit exponent". "$####.##-" is a typical format used to output dollars and cents up to $9999.99. If the number format begins with a "$", the exponential form ("^^^^^") may not be used. If a trailing "-" is used, then the number format may not have "^^^^^" and vice-versa. There must be at least one "#" in a number format; a maximum of 1Ø is allowed. No other character string is a number format (i.e., "-$" and "$." are not number formats).

The PRINT USING statement operates by alternately outputing parts of the format string and outputing values (from left to right) from the list-of-values. For each value in the print fields, PRINT USING does the following:

If the value is a string, the string is output as is. The format string is unaffected and plays no part in string output.

If the value is a number, then characters from the format string are output until the format string is exhausted or until a number format is encountered in the format string. If the format string is exhausted, then the value is printed according to the rules under PRINT (note: no spacing will occur in this case, except for a blank sign if the number is positive). Once a number format is encountered, the value is printed as specified by the format. For each character in the number format, SD BASIC prints exactly one character. A "-" sign in the number format is printed as "-" if the value is negative, otherwise as a blank. If the format contains no carets and there are leading zeros, a leading "-" sign is moved right until it is just to the left of the first significant digit. If no sign is specified in the number format, the output value must be positive or an error will result (since no space is allocated in which to print a minus sign). A "$" causes a "$" character to be printed immediately to the left of the sign character if the sign is leading and the value is negative; otherwise, it is printed where a sign character would have been printed in a normal printout. The "." is always printed as a ".", but it causes the number to have its decimal point printed in the designated place. Each "#" is printed as a digit (leading zeros to the left of the first digit to the left of the "." are replaced by blanks if not exponential form). A group of carets causes an exponent of the form "E-xxx" to be printed, where "-" is printed as "+" or "-" and xxx are exponent digits. The number of exponent digits printed is equal to the number of carets minus 2 (leaving room for the "E" and the exponent sign). They also cause the number to be printed with its most significant digit placed at the position of the leftmost hash mark in the format. If the value cannot be output using the number format, SD BASIC prints an asterisk for each character in the number format.

If the end of the PRINT USING statement is reached, and the format string has not been exhausted, then the rest of the format string is output as is, including any number formats.

Finally, a <CR> is output unless the optional trailing semicolon is present, in which case no other characters are output. This allows multiple PRINT USINGs to generate a single output line.

TAB may not be used in a PRINT USING statement.

Examples:

```
        1Ø PRINT USING "PI =-#.####",3.141593
```

prints out:

```
PI = 3.1415
```

```
        2Ø LET S$=" #.##^^^^ IS TEN PI IN E NOTATION"
        3Ø PRINT USING S$, PI*1Ø
```

prints out:

```
3.14E+Ø1 IS TEN PI IN E NOTATION
```

```
        35 LET S$= "IS NEGATIVE PI"\PRINT USING 5Ø,-PI,S$,PI+2
        5Ø FORMAT ">>>####.##- !! ####.##-IS PI+2"
```

prints out:

```
>>>   3.14-IS NEGATIVE PI !!    5.14 IS PI+2
```

```
        6Ø PRINT USING "#.## IS 75",75
```

prints out:

```
**** IS 75
```

```
        7Ø PRINT USING 8Ø, 5.93, 5.93, -5.93, -5.93
        8Ø FORMAT "$##.## : $-##.## : $-##.## : $##.##-"
```

prints out:

```
$5.93 :   $5.93 :  $-5.93 :  $5.93-
```

```
        9Ø PRINT USING 12Ø,2.92;\PRINT USING 13Ø;\PRINT USING 12Ø,9.1;
        12Ø FORMAT "##.##"
        13Ø FORMAT " IS NOT THE SAME AS "
        14Ø PRINT USING "... QED!";\PRINT\REM UNUSUAL USE OF PRINT USIN
```

prints out:

```
 2.92 IS NOT THE SAME AS  9.1Ø... QED!
```

```
        15Ø PRINT USING 16Ø
        16Ø FORMAT "THIS COULD BE A VERY LONG STRING"
        17Ø PRINT USING LEGALBUTDUMBFORMAT,26
        LEGALBUTDUMBFORMAT: FORMAT "%%"
```

prints:

```
        THIS COULD BE A VERY LONG STRING
        %%  26
```


FORMAT

The FORMAT statement is used to introduce a long or commonly used format string for PRINT USING statements. It consists simply of the word "FORMAT" and a quoted character string constant. A FORMAT statement must have a line number (or label), and must be the only statement on that line. A FORMAT statement acts like a REM statement in that executing it does absolutely nothing. The form is:

```
linenumber FORMAT string
```

Examples:

```
27 FORMAT "REMAINDER UNPAID:  ####.##"

5Ø FORMAT 'ACCOUNT BALANCE: $####.##- WITHDRAWALS: $####.##'

ACCTTOTALS: FORMAT " PAYROLL $####.## PROFIT $######.##"
```

Note: A FORMAT statement may not precede the first executable statement in a program.

LET

The LET statement is used to cause a variable to take on a new value. The form is:

```
LET variable = expression
```

If the variable is numeric, then the expression must produce a numeric value. If the variable is a string or a substring, then the expression must be a string expression. A substring specification as the variable must not exceed the current length of the string. Subscripted variables are also allowed on the left side of the equal sign. The word LET is optional.

```
1Ø LET A=5
```

causes A to take on the value 5 until a new value is assigned.

```
2Ø ZAP[2,3]=COS(ZAP[2,3])/2
```

computes a value and stores it in the second row, third column of the array ZAP.

```
3Ø LET Q[9]=B$[22]
```

sets the 9th entry in the numeric vector Q to the value of the 22nd byte of the string B$.

```
4Ø LET B$="ABC" CAT "D"
```

sets the string B$ to "ABCD". The former contents of B$ are completely lost.

```
45 LET A$="ABC"\ LET B$="ZX"\ LET A$=B$ CAT A$
```

sets A$ to "ZXABC".

```
5Ø LET B$[11,3]="ABC"
```

sets 3 characters of B$, starting in the 11th byte, to "ABC". B$ must have previously had a value whose length was 13 or more or a subscript error will result. The length of B$, and other bytes except B$[11], B$[12], and B$[13] are not affected in any way.

```
6Ø LET B$[1,2]="DEFGHI"
```

changes the first two 2 characters of B$ to "DE". The "FGHI" part is not stored.

```
65 LET B$[12,Ø]="ABC"
```

does not change B$ at all because the target substring length is zero.

```
70 LET B$[1,27]="ABCD"
```

stores "A", "B", "C" AND "D" into B$[1], B$[2], B$[3], B$[4] respectively. B$[4,23] is set to blanks. The rest of B$ is not affected.

With string assignments, only as many bytes as are specified by the minimum of the target and source string lengths are copied. Excess bytes in the source string are ignored; excess bytes in the target are blank-filled. If the target string is a string variable and not a substring, its current length is changed to the number of bytes copied. Storing into a substring does not affect the current length of the string containing the substring.

```
80 LET B$[3]=13
```

sets the third byte of B$ to an ASCII carriage-return. Note: B$[3]="ABC" is not legal since B$ in this example is a string variable; the left hand side is a numeric value, not a substring.

The current length of a string can be set to any value (less than or equal to the dimensioned string length) by writing

```
90 LET LEN(stringname$) = expression
```

This will truncate the string if the expression value is less than the current length of the string; if greater, the string is extended with garbage bytes. Extending the string in this fashion is also necessary before attempting assignment to a substring of it if the string has never previously been assigned a value.

SD BASIC is sensitive to substring overlap problems and automatically adjusts the direction of copying (first-to-last or last-to-first) in a string assignment to assure the intended result. For instance, given the statements

```
100 LET S$[2,3]=S$[3,3]
110 LET S$[3,3]=S$[2,3]
```

If S$ was "ABCDEF" before 100, it will be "ACDEEF" afterwards; if it was "ABCDEF" before 110, it will be "ABBCDF" afterwards.

Conditional expressions may be used anywhere an expression is allowed, including LET statements:

```
120 LET X= A>B OR NOT(C=D)
```

X will be set to TRUE if the condition A>B OR NOT(C=D) is true, otherwise X will be set to FALSE.

```
130 LIMERICK$[1] = "The wonderful Software Dynamics BASIC"
```

changes the first string in the LIMERICK$ string array to the given string (this is not legal if LIMERICK$ is not defined as a string array as this notation would then imply a numeric assignment to the first byte in the string LIMERICK$).

```
140 LET LIMERICK$[1](5,6) = "beauti"
```

changes "wonderful" (assigned by statement 13Ø) to "beautiful".

INPUT

The INPUT statement is used to allow user entry of values from the console into the program at execution time. The form is:

```
INPUT variable-name, variable-name, variable-name...
```

"variable-name"s can be subscripted so that input into a vector, array, or substring is possible. Only one string (or substring) reference is allowed per INPUT statement, and that must be the last variable name in the statement. Note again, if B$ is a string variable, B$[3] is a numeric variable, not a string or substring reference!

The INPUT statement causes a prompt ("? ") to be printed on the console, and BASIC then waits for a line to be entered on the same line as the prompt. The user must type a line of characters ended with a <CR> key. Editing facilities for error recovery on type-in are those provided by SDOS. BASIC interprets the typed in line as a list of numbers (in the form of numeric constants), and assigns the values found, from left to right, to the variables listed in the INPUT statement, from left to right. Values may be numeric or hex constants. Each value must be separated from its neighbor by a comma, or space(s) or tab characters. Tabs or spaces may optionally be used after the comma or before the first value. If the last variable in the input statement is a string reference, the rest of the input line, including leading spaces, is stored into the string as though a LET statement had been executed. Extra values or garbage in an input line beyond what the INPUT statement requires is ignored. If not enough values are entered, BASIC will re-prompt and ask for all the values again. Conversion errors on numbers cause an error print-out, and BASIC will re-prompt with "? ". The user must re-enter all the values required by the INPUT statement. The <CR> ending the line is not included as part of the line. The INPUT statement accepts a line of ASCII characters into the CATBUF; the size of the CATBUF determines the maximum legal input line size. An input line larger than the size of CATBUF will cause an Input Buffer Overflow error.

Examples:

The statement:

```
1Ø INPUT A
```

causes BASIC to print "? " on the terminal. If the user types "-17.2<CR>", then A is set to -17.2.

The statement:

```
20 INPUT S,B[S]
```

on entry of "3,:A2<CR>" will cause S to be set to 3, and B[3] to be set to 162.

The statement:

```
30 INPUT A$ when given "NUTS<CR>", does the same as:

30 LET A$="NUTS"
```

The statement:

```
40 INPUT A$
```

with a type-in of an empty line (just <CR>), sets A$="" (the empty string).

The statement:

```
50 INPUT B,A$[1,B]
```

with a typein of "12, HELLO<CR>", does the same as:

```
50 LET B=12\ LET A$[1,12]=" HELLO"
```

The statement:

```
60 INPUT B,A$
```

with a typein of "12,<CR>" sets B to 12 and sets A$ to the empty string.

If the programmer does not like the prompt that BASIC uses, he can force a new one for a particular INPUT statement by writing it as a string constant in the INPUT statement immediately after the keyword INPUT, which will cause BASIC to print the supplied string constant instead of the default prompt "? ":

```
70 INPUT "SOMETHING FOR X: " X
```

Since a prompt string may be empty, a variable string prompt can be forced by PRINTing the desired prompt, and then using an INPUT statement with an empty prompt string ("" or ''), as demonstrated by the following sequence:

```
80 PRINT PROMPT$;\ INPUT '' QWERTY
```

GOTO

GOTO causes BASIC to change the program flow by transferring control to the line labeled by the line number specified in the GOTO statement. The form is:

```
GOTO linenumber
```

Examples:

```
GOTO 1ØØ
GOTO TOPOFLOOP
```

The target line number must be defined. Attempting to GOTO into a blockbody (see BLOCK BODIES) from outside the blockbody is illegal. GOTO a label outside a blockbody from within the blockbody is legal.

A special form of the GOTO allows BASIC programmers to return from an error recovery routine in a simple way:

```
GOTO ELN
```

This statement specifies that control is to transfer to the last line number encountered before the last error occurred (see Error Handling). It cannot legally be executed unless an error has occurred; furthermore, an error trap routine must have been set up or the error would have aborted the program (see ON statement).

Note that GOTO ELN is purely a syntactic form, and does not imply that GOTO <exp> is legal, which it is not. Two legal variations of GOTO ELN are ...THEN ELN and ...ELSE ELN.

Example:

```
 5   REM USE SECONDARYNAME IF ENTERED FILE NAME DOESN'T EXIST
1Ø   ON ERROR GOTO 1ØØ
2Ø   INPUT "FILENAME: " FILENAME$
3Ø   OPEN #1, FILENAME$
 .
 .
 .
1ØØ FILENAME$="SECONDARYNAME"\ GOTO ELN
```


BLOCK BODIES

Many SD BASIC statements conditionally execute a statement list, or a block of statements. A statementlist (single line) form, or a multi-line form of such statements is used depending on which is desired.

A block body is a set of statement lines delimited by the keywords of a conditional BASIC statement. A block body may be used wherever a statement list may be used. BASIC always assumes that a statement list follows a conditional execution keyword, unless that keyword is followed by a <CR> alone, which signals BASIC that a block body follows. Block bodies are generally terminated by the keyword END, but ELSE, FI, UNLESS, WHEN and NEXT terminate block bodies of certain statements (see below).

The following is a list of statements and the keywords that introduce a block body, and the keyword that marks the end of the blockbody:

```
IF...THEN blockbody FI
IF...THEN blockbody ELSE...
IF...ELSE blockbody FI
REPEAT blockbody END
REPEAT blockbody UNLESS condition END
REPEAT blockbody WHEN condition END
FOR...blockbody NEXT variable
FOR...WHILE...DO blockbody END
FOR...UNTIL...DO blockbody END
FOR...DO blockbody END
WHILE...DO blockbody END
UNTIL...DO blockbody END
FOR...blockbody END
FOR...blockbody NEXT...
DEF parameterdefinitions blockbody END
SUBROUTINE parameterdefinitions blockbody END
IF ERROR WHEN blockbody THEN...
ON ERROR DO blockbody END
```

Note that a blockbody includes a <CR> as its last character, so the delimiting keyword must always be on the line following the blockbody. For specific examples, see the section on the appropriate statement.

A single-line form of these statements may be formed by replacing blockbody by a statementlist. The keyword END can be optionally dropped in the single line form if it would be followed by a <CR>.

IF (also THEN, ELSE, ELSEIF and FI)

The IF statement is used to conditionally transfer control or conditionally execute a statementlist. The form is:

```
IF condition THEN blockbody ELSE blockbody FI
```

The condition is some logical combination of relations between values (see CONDITIONAL EXPRESSIONS). The blockbody may be a linenumber or a statementlist in either the THEN or the ELSE clause; furthermore, the ELSE blockbody may be eliminated. The keyword FI is used only when it is unclear how IF statements are nested or to signal the end of a blockbody; see below.

The form:

```
IF condition THEN linenumber ELSE linenumber
```

is the same as

```
IF condition THEN GOTO linenumber ELSE GOTO linenumber
```

Control is transferred to the line specified by the THEN part if the condition is met; otherwise, control is passed to the line specified by the ELSE part. The ELSE part is optional; if not supplied, control is passed to the next statement when the IF condition is not met.

Example:

```
5 IF I<Ø THEN 4ØØ\ PRINT I
```

causes control to transfer to line 4ØØ if I<Ø; otherwise, the value of I will be printed and control will be passed to the next line (after line 5). Note that the IF statement in the example has no ELSE clause and that the PRINT statement is not part of the THEN statement list.

The form:

```
IF condition THEN thenstatements ELSE elsestatements
```

acts as though the following had been written instead:

```
IF condition THEN GOTO dummyl1
elsestatements
GOTO dummyl2 dummyl1 thenstatements dummyl2 REM
```

with dummyl1 and dummyl2 being invisible line numbers (but line numbers are not used). If the condition is true, only the statements in the THEN part are executed; otherwise, the statements in the ELSE part are executed (optional). This allows the previous example to also be written as:

```
1Ø IF I>=Ø THEN PRINT I ELSE GOTO 4ØØ
```

The THEN part may start on the physical line following the conditional part of the IF statement:

```
15 IF SALES > QUOTA
   THEN COMMISSION = COMMISSION * 1.1
```

Likewise, the ELSE part may be on the line following the last statement of the THEN statementlist:

```
2Ø IF A<2 THEN GOSUB 1ØØ\ PRINT A
          ELSE PRINT A-3
```

The statements in statementlist in the THEN and ELSE clauses are separated from each other by a "\" and an optional <CR>. If only a single statement occurs in a THEN or ELSE clause, no "\" is needed.

Example:

```
3Ø IF S>=Ø THEN PRINT SQR(S),\
                PRINT S,\
                PRINT S^2,\
                PRINT S^3\
                GOTO 7ØØ
        ELSE    PRINT " CAN'T DO SQR(";S;")" \
                GOTO 95Ø
```

The physical lines containing statements which are part of the THEN or ELSE statementlists must not have line numbers.

A blockbody in a THEN or ELSE clause allows one to write long sequences of statements without having to place \<CR> between them. Such a blockbody is introduced when the keyword THEN or ELSE is followed by a <CR> instead of a statementlist. The blockbody for a THEN clause is delimited by the keywords ELSE or FI on the line following the last line of the THEN blockbody; the blockbody for an ELSE clause is likewise delimited only by the word FI. The above example can thus be written as:

```
35 IF S>=Ø THEN
        PRINT SQR(S)
        PRINT S,S^2,S^3,
        GOTO 7ØØ
   ELSE
        PRINT "CAN'T DO SQR(";S;")"
        GOTO 95Ø
   FI
```

Unlike the statementlist case, line numbers are allowed in blockbodies (although it is illegal to branch to a line number in a blockbody from outside that blockbody). Any statement, including IF statements, may be placed in the blockbody.

When an IF statement has a blockbody in a THEN clause, but no ELSE clause, the word FI must be used to signal the end of the THEN blockbody (normally, ELSE does this). When used this way, FI must be on the line following the last line of the blockbody; no line number is allowed (on FI, THEN or ELSE). Example:

```
37 IF USERWANTSPRINTOUT
   THEN
       PRINT "RECORD FOR "; EMPLOYEE$
       PRINT "SALARY=";SALARY;"DATE OF HIRE: ";HIREDATE$
   FI
```

When IF statements are in a THEN or ELSE statementlist (or blockbody), it sometimes leads to a problem when one has statements of the form:

```
IF condition1 THEN ...\
                     IF condition2 THEN ...
                   ELSE ...
```

Which IF does the ELSE belong to? The Software Dynamics BASIC allows the word "FI" to close off an IF to prevent such a problem:

```
IF condition1 THEN ...\
                      IF condition2 THEN ... FI
                   ELSE ...
```

In this case, the ELSE belongs to the first IF; in the previous case, by convention, the ELSE belongs to the most recent unclosed IF (i.e., the second IF). IF and FI nest like left and right parentheses. When FI is used after a statementlist, it must be on the same physical text line as the last physical text line of statement list. An optional FI may be supplied after an ELSE statement list; after a blockbody, FI is required.

Examples:

```
4Ø IF B=2 THEN 49

5Ø IF B>2 OR NOT( C$=BAT$(1,4) ) THEN GOSUB 96
        ELSE PRINT B

6Ø SIGNX=1\IF X>=Ø THEN IF X=Ø THEN SIGNX=Ø FI ELSE SIGNX=-1

7Ø IF A$="***"
   THEN
      A$='???'
   ELSE
      A$='!!!'
   FI
```

A special form of the IF statement may be used as a term in an expression (see IF function).

Another special form (IF ERROR WHEN ...) can be used to handle errors (see Error Handling).

The ELSEIF keyword is especially convenient in a set of sequential IFs. It may be used anywhere that ELSE would be legal. The form

```
...ELSEIF condition THEN blockbody ELSE...
```

acts as though

```
...ELSE IF condition THEN blockbody ELSE...FI
```

had been written, thus avoiding the writing of many FIs.

Example:

```
ASKCOMMAND:
IF COMMAND$ = "QUIT" THEN EXIT
ELSEIF COMMAND$ = "UPDATE" THEN UPDATE
ELSEIF COMMAND$ = "DELETE" THEN DELETEDATA
ELSEIF COMMAND$ = "EXAMINE" THEN EXAMINERECORD
ELSE PRINT "WHAT?"\GOTO ASKCOMMAND
```


WHILE

The WHILE statement is used to perform a set of statements an indefinite number of times, WHILE a condition is true. The form is:

```
WHILE condition DO blockbody END
```

The condition is any valid conditional expression. The WHILE statement acts as if

```
dummy1 IF condition THEN statements\GOTO dummy1
```

had been written, where statements correspond to the statements in blockbody, but the line number is unnecessary. The "END" is not the end of the program, but simply terminates the WHILE loop.

Examples:

```
1Ø WHILE J<1Ø DO LET J=J+1 END

2Ø WHILE A[I]<>Ø DO
       PRINT A[I]
       I=I-1
   END

3Ø X=P-1\WHILE INT(P/X)<>P/X DO X=X-1\! FIND MAX DIVISOR OF P
```

UNTIL

The UNTIL statement is identical to WHILE except that the condition is inverted (tested for false instead of true).

Examples:

```
1Ø  UNTIL A$[I]=" " DO I=I+1 END

2Ø  UNTIL ABSERROR < 1E-1Ø DO
      REM DO NEWTON-RAPHSON ITERATION TO COMPUTE SQUARE ROOT
      X=(VALUE/X + X)/2
      ABSERROR = ABS(VALUE - X^2)
    END

3Ø UNTIL MONEYLEFT=Ø
   DO MONEYLEFT=MONEYLEFT/2+AMOUNTWONONBET(MONEYLEFT/2)
```

REPEAT

The REPEAT statement allows unconditional looping to occur. The only way out of a REPEAT loop is via a GOTO or RETURN statement. The form is:

```
REPEAT blockbody END
```

which is equivalent to

```
WHILE TRUE DO blockbody END
```

Examples:

```
1Ø REPEAT I=I+1\ IF B[I,J]^2<35Ø THEN 2ØØ\ J=J-2 END

2Ø REPEAT I=I+1\IF I>Ø THEN 1ØØØ

3Ø REPEAT
     SLOTSELECTOR=SLOTSELECTOR+1
     IF A(SLOTSELECTOR)=Ø THEN FREESLOTFOUND
     NSLOTS=NSLOTS-1
     IF NSLOTS=Ø THEN NOFREESLOTS
   END

4Ø REPEAT
      INPUT "GIMME THE ANSWER" ANSWER$
   WHEN NOTLEGALANSWER(ANSWER$) END
```

Two other forms of the REPEAT statement allow a loop to be executed one or more times (as opposed to WHILE, which allows zero or more times). These forms are:

```
REPEAT blockbody WHEN condition END

        and

REPEAT blockbody UNLESS condition END
```

The keyword END is optional for these forms of REPEAT.

REPEAT...WHEN is logically equivalent to:

```
invisiblelabel: blockbody
                IF condition THEN GOTO invisiblelabel
```

REPEAT...UNLESS is logically equivalent to:

```
invisiblelabel: blockbody
                IF NOT(condition) THEN GOTO invisiblelabel
```


FOR and NEXT

The FOR statement, in conjunction with NEXT, is used to control iterative loops in BASIC. This is useful for scanning arrays, computing a function on some set of values separated by a fixed increment, etc.

The FOR statement provides for the specification of a loop index variable, an initial value, a limit value, and STEP value. It also marks the beginning of the loop. The form is:

```
FOR variable = expl TO exp2 STEP exp3
```

where expl is an initial value expression, exp2 is a limit value expression, and exp3 is a step value expression. The variable cannot be a string, array or vector, nor may it be subscripted. The STEP part of the statement is optional; if not specified, a default step of +1 is assumed.

NEXT is required to mark the end of a loop. The form is:

```
NEXT variable
```

where the variable specified following NEXT must match that specified in the FOR statement. To be consistent with other block body forms, the word "END" may be used in place of "NEXT variable".

The set of lines:

```
1Ø FOR I=INITIALV TO LIMITV STEP STEPV
2Ø ...
3Ø ...
4Ø NEXT I
```

acts as though the following had been written:

```
1Ø LET I=INITIALV
invisiblelabell: IF STEPV>=Ø AND I>LIMITV ...
&    OR STEPV<Ø AND I<LIMITV THEN GOTO invisiblelabel2
2Ø ...
3Ø ...
4Ø LET I=I+STEPV\GOTO invisiblelabell
invisiblelabel2: REM ...
```

except that execution of the NEXT part is much quicker than the corresponding BASIC statements. The values of LIMITV and STEPV are evaluated once at loop entry and do not change during execution of the loop. The body of the loop is executed (LIMITV-INITIALV)/STEPV+l times so that

```
FOR I=1 to 1Ø
```

causes 1Ø iterations. Note that the loop body will be executed zero times if the INITIALV is "beyond" LIMITV on entry to the loop.

A FOR statement causes the body of the loop (all statements between the FOR and the NEXT) to be executed iteratively, with each iteration of the loop assigning a new value for the loop index variable (mentioned in the FOR statement). After execution of the last iteration, control will pass to the statement following the NEXT. The loop variable will be incremented past the limit value.

Example:

```
1Ø FOR I=Ø TO LEN(A)\ LET A[I]=Ø\ NEXT I
```

This zeros the vector named A.

```
15 FOR COUNT=1 TO COUNT
       LET SUM=SUM+A[COUNT]
   END
```

Note use of the index variable in limit expression.

Loops may be nested indefinitely for dealing with multi-dimensioned searches, etc.:

```
2Ø  FOR I
     ...
      FOR J
       ...
        FOR K
        ...
        NEXT K
       ...
      NEXT J
     ...
    NEXT I
```

It is illegal to allow loop bodies to cross in the following manner:

```
FOR I
   FOR J
     ...
NEXT I
     ...
   NEXT J
```


An UNTIL/WHILE option allows coding of iterative loops with a conditional exit:

```
FOR var=expl TO exp2 STEP exp3 WHILE condition
        or
FOR var=expl TO exp2 STEP exp3 UNTIL condition
```

The WHILE version is identical to

```
    FOR var = expl TO exp2 STEP exp3
    IF NOT(condition) then GOTO invisiblelabel
     ...
    NEXT var
invisiblelabel: REM ...
```

The UNTIL version is identical to

```
    FOR var = expl TO exp2 STEP exp3
    IF condition then GOTO invisiblelabel
     ...
    NEXT var
invisiblelabel: REM ...
```

where invisiblelabel is an invisible label.

The UNTIL or WHILE clause may not be on the line following the FOR statement, as they would then be treated as a conventional UNTIL or WHILE statement.

If a FOR statement ends with the optional word DO, then the block of statements executed is terminated with an END instead of NEXT var.

Example:

```
FOR I=expl TO exp2 DO blockbody END
```

A single line FOR statement can be constructed by writing a statement list immediately following the DO (the word END is optional):

```
FOR var=expl to exp2 DO statementlist
```

This is equivalent to:

```
FOR var=expl to exp2
    statementlist
NEXT var
```

Examples:

```
5Ø FOR I=1 TO ROWS(A) UNTIL A[I,1]=Ø
   NEXT I

6Ø FOR I=Ø TO 1Ø DO VECTOR(I)=Ø

7Ø FOR X= .62 TO 91 STEP .Ø2 UNTIL F(X)>.2 DO
       LET SUM=SUM+F(X)
   END

8Ø FOR INFLATION = .Ø1 TO .2Ø STEP .Ø5
       IF GNP*(INFLATION+1)>1E12 THEN FINANCIALDISASTER
       LET TAX = TAX*(1+2*INFLATION)
       PRINT TAX
   END
```

CYCLE

The CYCLE statement is used to start the next iteration of a FOR-NEXT loop. It must be nested properly within the loop which is to be cycled. The loop index variable must be specified in the statement.

Example:

```
CYCLE variable
```

The sequence:

```
FOR I=...
   ...
   CYCLE I
   ...
NEXT I
```

is logically identical to:

```
FOR I=...
   ...
   GOTO invisiblelabel
   ...
invisiblelabel: NEXT I
```

Examples:

```
1Ø FOR J=Ø TO LEN(V) DO IF V(J)=Ø THEN CYCLE J\V(J)=V(J)+1

2Ø FOR AROUNDCIRCLE=Ø TO 2*PI STEP .Ø1
       IF SIN(AROUNDCIRCLE)>.5 THEN CYCLE AROUNDCIRCLE
       LET AREA=AREA+RADIUS*CHORD
   END
```

EXIT

The EXIT statement has several purposes:

1) to EXIT a BASIC program completely
2) to EXIT a FOR-NEXT loop without doing a GOTO
3) to EXIT a block without doing a GOTO
4) to EXIT (return from) a SUBROUTINE

To exit a BASIC program, the following statement is used:

```
EXIT
```

No messages of any kind are printed. An EXIT syscall is executed, and control passes to SDOS.

To EXIT a FOR-NEXT loop with index variable "indexvariable", the following statement suffices:

```
EXIT indexvariable
```

This form of an EXIT statement must be inside the body of the FOR-NEXT block being executed. Control is passed to the statement immediately following the corresponding NEXT indexvariable (or END) statement. The value of the index variable is preserved.

To exit a labeled block, the form

```
EXIT labelname
```

is used. This is identical in function to a GOTO to the statement immediately following the end of the labeled block. This form must be textually inside the labeled block being EXITed. "Labelname:" must be on the same source line that begins the block (see line continuation in the section USING SOFTWARE DYNAMICS BASIC to get around this).

To return from a subroutine, the following is written:

```
EXIT SUBROUTINE
```

This also causes the error trap routine selected and active GOSUBs initiated since entry of the subroutine to be discarded.

Examples:

```
 ...
10 IF UPPERCASE$(COMMAND$)="QUIT" THEN EXIT
 ...
REM THE FOLLOWING LOOP HUNTS FOR AN ARRAY LOCATION
REM THAT CONTAINS ZERO OR A VALUE GREATER THAN 3
REM SUCH THAT NO ARRAY LOCATION TO THE LEFT
REM IN THE SAME ROW IS NEGATIVE.
FINDARRAYSLOT: FOR I=1 TO 10
   FOR J= 1 TO 10
      IF A(I,J)<0 THEN EXIT J
      IF A(I,J)=0 THEN EXIT FINDARRAYSLOT
      IF A(I,J)>3 THEN EXIT I
      REM A(I,J) SATISFIES THE CONDITIONS
   END
   REM "EXIT J" PASSES CONTROL TO HERE
END
REM "EXIT I" OR "EXIT FINDARRAYSLOT" PASSES CONTROL TO HERE

REM MULTIPLE LOOP EXITS
ABC: REPEAT
        ...
        IF ... THEN EXIT ABC
        ...
        IF ... THEN EXIT ABC
        ...
     END
REM CONTROL PASSES HERE WHEN "EXIT ABC" IS EXECUTED
 ...
SUBROUTINE QED (    )
   ...
   EXIT SUBROUTINE
END
```

GOSUB

The GOSUB and RETURN statements are used to implement simple subroutines. GOSUB transfers control to a subroutine, and RETURN causes control to transfer back. The form of a GOSUB statement is:

```
GOSUB linenumber
```

Control is passed to the line specified. BASIC remembers the location of the statement following the GOSUB (even if it is in the middle of a statementlist or blockbody).

RETURN

The form of a RETURN statement is simply:

```
RETURN
```

Control is passed back to the statement following the most recent not-yet-RETURNed-to GOSUB.

GOSUB calls may be nested to an arbitrary depth. An actual limit is determined by the free space available in the user area once the RTP and compiled BASIC program are loaded; it is also affected by the data space used by the BASIC program. Under normal circumstances, the GOSUB stack is deeper than any program can realistically use. It is wise to return from every subroutine called; otherwise, the GOSUB stack eventually builds up a residue and steps on something critical.

Example:

```
1Ø LET A=2\GOSUB 1ØØ\LET A=3\GOSUB 1ØØ
2Ø PRINT "DONE"
     .
     .
     .
1ØØ PRINT "ARGUMENT = ";A; "SQUARE = "; A^2\ RETURN
```

The special form,

```
RETURN SUBROUTINE
```

is identical in function to

```
EXIT SUBROUTINE
```

and is used to return control from a SUBROUTINE to the CALLer.

Another form,

```
RETURN expression
```

is used to return from a DEFined function (see DEF).

GOSUB POP

The GOSUB POP statement allows the BASIC programmer to pop the return address stack without transferring control. The form is:

```
GOSUB POP exp
```

The return stack is popped "exp" times; if exp is 3, 3 return addresses are removed from the top of the return stack. Control is passed to the statement following the GOSUB POP statement. If exp is zero, the entire return stack is popped and left empty. This is useful in error recovery routines.

If the stack is popped too many times, an error occurs and BASIC leaves the return stack empty.

Example:

```
100 ON ERROR GOTO 200
    ...
200 GOSUB POP Ø\ GOTO 300
    ...
500 GOSUB 1000
    ...
1000 LET B=C/Ø\ REM CAUSES ERROR
```

STOP

The STOP statement is used to abnormally terminate program execution. STOP causes the last line number encountered during execution to be printed. The form is:

```
STOP
```

A good example of use is:

```
    TAX = ...
    REM CHECK FOR IMPOSSIBLE CONDITION
14  IF TAX<Ø THEN STOP
```

The EXIT statement should be used (instead of STOP) when a printed line number is undesirable. A suggestion is to use STOP only when the line number printout is important for program debugging.

ON

The ON statement is used for multi-way branching. The form is:

```
ON exp GOTO linenumber1, linenumber2, linenumber3...
```

The expression must be numeric. The value is rounded to an integer. If the integer is 1, control is transferred to the first linenumber (linenumber1) in the list; if 2, to the second linenumber, 3 to the third, etc. If the integer is less than 1 or greater than the length of the list of line numbers in the ON statement, control is passed to the next statement. ELN may not be used as a linenumber in an ON statement.

Example:

```
1Ø ON A+2 GOTO 1Ø,2Ø,3Ø,4Ø
```

If A=1, control will be passed to line 3Ø.

The form:

```
ON exp GOSUB l1, l2, l3...
```

does a GOSUB to the line number specified by the expression in the same manner as ON-GOTO. RETURN causes control to pass to the statement following the ON-GOSUB statement.

Example:

```
2Ø INPUT X\ ON X GOSUB 1ØØ,2ØØ,USERREQUEST3,4ØØ\ GOTO 2Ø
```


REM

The REM statement is used to annotate the program and has no effect on its execution. However, a line number attached to a REM can serve as a branch target (of GOTO, GOSUB, etc.), provided it is not in the block of REM and DIM statements at the front of a program. The form is:

```
REM any string of characters <CR>
```

The remark includes all text after the word REM to the <CR>.

Examples:

```
1Ø REM NOW ADD 1 \ A=A+1

2Ø REMARK$="HELLO"
```

These are both entirely comments and have no effect on program execution.

The word REM can be replaced by an exclamation point, for example:

```
3Ø LET A=A+1\ ! BUMP A
```

is a valid line.

ERROR HANDLING

Building a bullet-proof program is impossible without error handling. SD BASIC provides a very general and efficient error detection, propagation, and user error handling facility.

A form of the ON statement can be used to specify an error trap routine. The form:

```
ON ERROR GOTO linenumber
```

sets up a dynamically associated error handling routine. Execution of this statement causes BASIC to remember the linenumber. If an error occurs later (in program execution), instead of issuing an error message, BASIC simply does a GOTO to the remembered line. It is expected that the line specified is the beginning of a routine to effect an error recovery. Once control has passed to an error recovery routine, the ERR function will produce a number corresponding to the error for testing, and the ELN function will produce the number of the last line executed (or in execution) when the error occurred. Error recovery, once the error type has been determined, is simple: either any corrective or diagnostic action is taken by the error routine, or an "ERROR" statement is executed, which causes BASIC to print out the error message just as if the error recovery routine had not been involved. Error recovery can be disabled by executing the statement:

```
ON ERROR GOTO Ø
```

Example:

```
        ...
100 ON ERROR GOTO 10000
        ...
200 IF B/I=4 THEN 600
        ...
10000 REM ERROR RECOVERY FOR DIVIDE BY ZERO
10010 IF ERR=14 AND ELN=200 THEN LET I=1\GOTO 200
10020 ERROR
```

This program recovers from a Division by Zero error by making the divisor in the IF statement nonzero, and transferring control back to the IF statement that failed.

The GOTO 2ØØ statement could also have been replaced by a GOTO ELN statement.

Two statement forms make coding of error trap routines more convenient. One form is:

```
ON ERROR DO statementlist END
```

Execution of this statement sets up an error trap to execute the specified statement list. Any error trap routine already established is superseded. The statement list is NOT executed at this time; control is simply passed to the next statement. When an error trap occurs after execution of an ON ERROR DO, then error trapping is disabled and the statement list is then executed. The statement list should GOTO somewhere when complete; if the END of the DO block is reached, an implied ERROR statement is executed.

Example:

```
5Ø INPUT "FILE" FILENAME$
6Ø ON ERROR DO IF ERR=1Ø11 THEN PRINT "NO SUCH FILE"\GOTO 5Ø
7Ø OPEN #1, FILENAME$
8Ø ON ERROR GOTO Ø\! DISABLE ERROR TRAP
```

Another statement form allows the programmer an easy way of specifying the range of statements over which the error trap routine should be effective.

```
IF ERROR WHEN blockbody THEN ... ELSE ...
```

Execution of this statement causes any previous error trap to be superseded, and the statement list to be executed. If an error occurs while executing blockbody, error trapping is disabled and control passes to the THEN clause. If the statementlist executes with no errors, then error trapping is disabled, and control passes to the (optional) ELSE clause. The structure of the THEN and ELSE clauses are described in the section on the IF statement.

When control reaches the next statement (via the THEN or ELSE clauses), error trapping is disabled.

Because of this interacting effect on the error trap routine, the programmer should decide either to use the ON ERROR GOTO or the ON ERROR DO/ IF ERROR WHEN style when designing a program that requires error handling. Trying to mix styles is very difficult.

Example of error handling:

```
ASKFORFILENAME: IF ERROR WHEN
                   INPUT "FILE " FILENAME$
                   OPEN #2, FILENAME$
                THEN
                   IF ERR=1 THEN OPERATORREQUESTEDATTENTION
                   ELSEIF ERR=1Ø11 THEN ASKFORFILENAME
                   ELSE ERROR
                FI
```

Another example:

```
1Ø   REM THIS PROGRAM CANNOT BE STOPPED BY AN ESCAPE IN LINES 1ØØ-19Ø
2Ø   DIM ...
     ...

7Ø   ON ERROR GOTO 1ØØØ
1ØØ ...
11Ø ...
     ...
19Ø ...
2ØØ ON ERROR GOTO Ø

1ØØØ IF ERR=1 AND 1ØØ<=ELN AND ELN<=19Ø THEN GOTO ELN 1Ø1Ø ERROR
     END
```

Note that whenever a function or SUBROUTINE is invoked, the error trap environment of the caller is saved and a new error trap environment is set up, initially with error traps disabled (see DEF, SUBROUTINE).

Error trapping must be re-enabled within the function or subroutine if needed. On exit, the caller's error trap environment is restored. An error in a function or subroutine when no error trap is set, or execution of an ERROR statement, will cause the execution of the function/subroutine to be aborted, and control is passed to the caller's error trap routine if it is enabled. Thus, an error is propagated up until some level of the program handles it, or until the main program is reached and no error trap is set, which causes the program to be aborted, and the line number and error is then printed.

An error trap occurring in an error recovery routine or Error 27 (wrong number of arguments) is treated as fatal and cannot be trapped.

ERROR

The ERROR statement is meant for use in an error recovery routine, and nowhere else. Its form is:

```
ERROR
```

At the main program level, if no error has occurred when this statement is executed, it acts like a STOP statement; otherwise, ERROR causes the message corresponding to the last error to be printed, and program execution ceases. In a subroutine or function, execution of this statement will cause processing of that subroutine (function) to cease, and the error it would have reported is caused in the calling program. In this fashion, a subroutine or function can pass an error up to its caller. See the example in the section on the "ON" statement.

Example:

```
IF ERR<>1 THEN ERROR
```

A special form of the ERROR statement may be used to cause a specific error to occur. This is useful when constructing subroutines that check parameter validity; the subroutine can generate an arbitrarily defined error if the parameters are wrong. The form is:

```
ERROR expression
```

The expression is evaluated, and an error trap occurs with an error code corresponding to the value of the expression.

Example:

```
DEF DIVIDE(DIVIDEND,DIVISOR)
   IF DIVISOR=Ø THEN ERROR 14
   RETURN DIVIDEND/DIVISOR
END
```


SUBROUTINES AND FUNCTIONS

DEF

The DEF statement is used to define a user function. A user function is convenient whenever a fixed sequence of steps is required to compute a value, and the value needs to be computed in several different places in the program. The form is:

```
        DEF fnname(paramvar1,paramvar2,...paramvarn) = expression
                 or
        DEF fnname(paramvar1,paramvar2,...paramvarn)
            DIM statements
            statementlist
        END
                 or
        DEF fnname(paramvar1,paramvar2,...paramvarn) EXTERNAL
```

"fnname" is the name of the function. Standard BASIC limits this to FNA, FNB, ... FNZ; SD BASIC allows any name not defined elsewhere in the program to be used here. If the fnname ends in a "$", then the function must compute a string result, otherwise the function must compute a numeric result.

The "paramvar"s give the names of the formal parameter variables of the function. When the function is invoked, the parameter variables are given values specified by the function invocation; the body of the function may refer to these parameter variables in the course of its execution. These parameter variable names must be unique over the entire BASIC program, i.e., once used as a parameter variable name, it may not be used again in a subsequent parameter variable declaration. The type of each parameter determines the type of the expression that must be used as a corresponding argument when the function is invoked. A parameter name ending in "$" indicates the corresponding argument must be a string; otherwise, the argument must be a numeric type. A parameter may also be followed by [*] or [*,*], meaning "vector of" or "array of" respectively. When the bracket notation is used, the parameter variable is interpreted as an array name, and may be so used in the function body. The parameter list may be empty, in which case the ()s must be dropped from the DEF statement.

The function body is a list of BASIC statements to be executed to obtain the value of the function. A function signals completion of the computation by executing a

```
RETURN expression
```

statement. The value of the expression is used as the result of the function. The form

```
DEF fnname(...)=expression
```

is identical to

```
DEF fnname(...)
    RETURN expression
END
```

DIM statements in the function (before the executable statements) allow the function to have its own "local" variables (although DIM'd variables must have names unique over the entire BASIC program). References to local DIMs from code not in the body of the function are illegal.

The word END at the end of a multiline function terminates the definition of the function, not the end of the BASIC program. END is compiled as a STOP statement, so control should not be allowed to pass to the END statement.

The EXTERNAL form notifies the compiler that the function is defined externally from this compilation module. This is discussed further under SEPARATE COMPILATION. No function body or END need be given.

It is illegal to GOTO or GOSUB outside the definition of a function (but other functions or subroutines may be called). It is also illegal to GOTO or GOSUB to a point within the definition of a function or subroutine from outside the definition.

The DEF statement defining a function must appear before any use of the function. It must be the only statement on a particular line, and it cannot be part of a THEN or ELSE clause in an IF statement. A line number is not required.

A user defined function can be used wherever an expression or subexpression is allowed, by writing:

```
...fnname(argexp1,argexp2,...argexpn)...
                   or
...fname argexp...
```

(Note that single argument functions do not need parentheses around the argument.)

Execution of this invocation occurs roughly as follows:

1) The values of each "argexp(i)" are assigned to the corresponding "paramvar(i)". WARNING: BASIC does NOT verify that the type of an argument matches that of the parameter variable; the programmer MUST guarantee this or unpredictable results will occur.

2) Control passes to the first executable line of the function.

3) Statements in the function body are executed until a RETURN expression statement is encountered.

4) The expression in the RETURN statement is evaluated, and its value is used in the invoking expression in place of the function call.

Examples:

```
1Ø DEF ROUND(VALUE)=INT(VALUE+.5)
   PRINT ROUND(Ø), ROUND (.5), ROUND (271.98)

2Ø DEF MAX(A,B)
   IF A>B THEN RETURN A
   RETURN B
END

DEF HEXBYTE$(X) = HEX$(X)[4,2]

DEF E= 2.71828182

DEF RNDØTO1Ø=1Ø*RND
```

Execution of a RETURN SUBROUTINE statement while in a function is illegal and will give unpredictable results.

During execution of a function body, a new environment for GOSUBs and error trapping is established. A new GOSUB stack is set up; the GOSUB stack of the caller is hidden until a (function) RETURN is performed. In particular, GOSUB POP Ø will clear only the function's GOSUB stack, not the caller's. A new error trap environment is also set, with error trapping initially disabled. Execution of an ERROR statement, or the occurrence of an error with no error trap set will cause the function invocation to be aborted, and the error will be triggered in the calling environment. Also, a new PRINTUSING environment is set. Note that there is only one Concatenation Buffer; it is NOT saved on function entry.

Example:

```
DEF LOG1Ø(X)
    ON ERROR GOTO HANDLEERR
    RETURN LOG(X)/LOG(1Ø)
HANDLEERR:
    IF X=Ø THEN RETURN -1E-126
    ERROR
END
```

Parameters are passed by "reference", not by value. This means that a parameter variable, if modified, will cause the value of the original argument to change (expression arguments are placed in temporary locations). A parameter may be passed as an argument to another subroutine or function; the call-by-reference will nest any number of levels.

Example:

```
DEF NEXTVALUE(VALUE)
    VALUE=VALUE+1
    RETURN VALUE
END
...
LET X=Ø
PRINT NEXTVALUE(X), NEXTVALUE(X), NEXTVALUE(X)
```

This program prints

```
  1               2               3.
```

Both numeric and string array parameters may also be passed. A parameter defined with the [] notation must be used as an array name throughout the body of the function. (Note that an array may be passed as an argument by writing its name, and excluding subscript notation). The BASIC functions LEN, ROWS and COLUMNS are useful for dealing with array parameters.

Example:

```
DIM A(1Ø,1Ø)
...
DEF DETERMINANT(Q[*,*])
    FOR I=1 TO ROWS(Q)
        FOR J=1 TO COLUMNS(Q)
        ...Q[I,J]...
        NEXT J
    NEXT I
    RETURN TheResult
END
    ...
LET DET=DETERMINANT(A)
```

Variables referenced in a function or subroutine may be parameters defined by the function/subroutine, local DIMs, or variables whose value is DIM'd, COMMON'd, or declared as a parameter variable (in a SUBROUTINE or DEF statement) by text enclosing the function/subroutine definition. Externally defined functions or subroutines cannot reference values in another main program, function or subroutine unless that value is COMMONed or passed as a parameter.

If control reaches a DEF statement in a program, it passes to the first statement beyond the end of the function definition.

Functions are NOT recursive; the following program will NOT work:

```
DEF FACTORIAL(X)
    IF X=Ø THEN RETURN 1
    ELSE RETURN FACTORIAL(X-1)*X
END
```


Locally DIM'ed variables have a static existence; their values are preserved from call to call, however, initialization of a locally DIM'ed variable will re-occur each time the subroutine/function is called. Locally DIM'd variable names must be unique over the entire program.

Example:

```
DEF SUBTOTAL(X)
    DIM SUBTOTALAMOUNT
    IF X<Ø THEN SUBTOTALAMOUNT=Ø
           ELSE SUBTOTALAMOUNT=SUBTOTALAMOUNT+X
    RETURN SUBTOTALAMOUNT
...
LET TRASH=SUBTOTAL(-1)
PRINT SUBTOTAL(5),SUBTOTAL(2.2),SUBTOTAL(9.6)
```

This program prints

```
5      7.2    16.8
```

Use of string function results can occasionally cause difficulty, due to a subtlety of the Runtime Package implementation. To enhance the performance of the system, string function results are passed by reference and not by value (numeric function results are passed by value, and so this problem cannot occur). This can lead to a problem (especially in Uniform Reference routines) if the string function computes its result by performing an assignment to a temporary string, and then returns the value of the temporary string as the result. When the results of such a string function, applied to two different argument lists, are used "simultaneously", the "value" is a pointer to the same place, and so the results of the first invocation of the string function are lost.

Example:

```
REM AN EXAMPLE OF A SUBTLE ERROR
DEF FIRSTDIGIT$(X)
    DIM DIGIT$(1)
    DIGIT$=NUMF$("#",X)
    RETURN DIGIT$
END
...
I=1\J=2\IF FIRSTDIGIT$(I)=FIRSTDIGIT$(J) THEN PRINT "OOPS"
```

Since the value of FIRSTDIGIT$(I) is the content of DIGIT$, when it is compared to FIRSTDIGIT$(J), which was also placed in DIGIT$, DIGIT$ is effectively being compared to itself, and so the equality always holds and OOPS is always printed. If this problem occurs, assignment of the result to another temporary will be required. The following example shows correct use:

```
I=1\J=2\T$=FIRSTDIGIT$(I)
IF  T$=FIRSTDIGIT$(J) THEN PRINT "OOPS"
```

USER-DEFINED SUBROUTINES

Subroutines are used to collect a set of statements for performing a commonly used sequence of operations into a package which is easily invoked. Subroutines may be defined anywhere, even following use. The definition is as follows:

```
SUBROUTINE subrname(paramvarl,...paramvarn)
           statementlist
END
        or
SUBROUTINE subrname(paramvarl,...paramvarn) EXTERNAL
```

The SUBROUTINE definition must occupy a line by itself. The END statement terminates the definition of the subroutine, and is compiled as a STOP. If control reaches a SUBROUTINE definition, it is passed to the first statement beyond the END of subroutine. "Subrname" is any name not used elsewhere in the program. It may have a trailing "$"; if so, invocation of the subroutine also requires the $. A subroutine name may not be used as a variable.

The EXTERNAL form notifies the compiler that the subroutine is defined externally from this compilation module (further discussion may be found under SEPARATE COMPILATION). No subroutine body or END need be given.

The parameter variables operate identically to parameters for functions.

The statement list contains statements to be executed to obtain the desired effect. Subroutine execution is terminated when an EXIT SUBROUTINE or RETURN SUBROUTINE statement is encountered (RETURN by itself is used to RETURN from a GOSUB). Execution of a RETURN <expression> statement in a subroutine is illegal and will cause unpredictable results.

Like functions, a subroutine may have local DIMs, and a new context is defined for the GOSUB stack, error handling, and PRINT USING formats (see DEF).

Subroutines are invoked by the CALL statement.

Example:

```
DIM X(5,5)
...
SUBROUTINE TRANSPOSE(A[*,*])
        IF ROWS(A)<>COLUMNS(A) THEN CALL PRINTERR(9)
        FOR I=1 TO ROWS(A)
            FOR J=1 TO COLUMNS(A)
               LET T=A(I,J)\LET A(I,J)=A(J,I)\LET A(J,I)=T
            NEXT J
        NEXT I
        RETURN SUBROUTINE
END

SUBROUTINE PRINTERR(ERRORNUMBER)
        REM PRINTS SDOS ERROR MESSAGE CORRESPONDING TO ERRORNUMBER
        OPEN #3, "ERRORMSGS.SYS"
        READ #3@ERRORNUMBER*3, THREEBYTES$
        READ #3@(THREEBYTES$[1]**8+THREEBYTES$[2])*256...
&       +THREEBYTES$[3],ERRORMESSAGE$
        PRINT ERRORMESSAGE$
        EXIT SUBROUTINE
END
...
CALL TRANSPOSE (X)
...
```

CALL

The CALL statement is used to invoke a subroutine written in BASIC or assembly code (for details on how to write an assembly subroutine, see ASSEMBLY LANGUAGE INTERFACE). The form is:

```
CALL name
   or
CALL name (arg1,arg2,...argn)
```

Where name is the name of the subroutine to be called, and the args are values to be used for the parameters specified, in left-to-right order, in the SUBROUTINE definition statement.

The compiler does not check for matching number of arguments or argument types. However, a BASIC SUBROUTINE will complain at execution time if the argument count is wrong; a fatal "Wrong number of Arguments" error is issued. This error cannot be trapped. Passing the wrong type of argument causes unpredictable results.

Since arguments of SUBROUTINEs (and functions) are passed by reference, the called routine could possibly modify them. For numeric scalar variables, this can be prevented by enclosing the variable in parentheses, which causes the compiler to treat it like an expression. The routine called cannot detect that argument is "protected". Array and string arguments cannot be protected against modification.

Example:

```
ABC=1
CALL MODIFY(ABC)
REM ABC=2 HERE
CALL MODIFY((ABC))
REM ABC STILL HAS THE VALUE 2 HERE
CALL MODIFY(46)
...
SUBROUTINE MODIFY(X)
     X=X+1\PRINT X,\RETURN SUBROUTINE
END
```

The example prints

```
2     3     47
```

An implied CALL statement is assumed when a subroutine name is found where a statement keyword is expected. The subroutine must have been defined (or mentioned in a CALL statement) prior to the implied CALL.

Examples:

```
        SUBROUTINE FIRETORPEDO(TORPEDONUMBER)
                ...
        END
                PRINT "Fire 1"
                FIRETORPEDO(1)
                PRINT "Fire 2"
                FIRETORPEDO(2)
                ...

        SUBROUTINE MANIPULATESTOCKMARKET(DAY) EXTERNAL
                ...
        MANIPULATESTOCKMARKET(TOMORROW)
```

UNIFORM REFERENCE

An extremely useful, but little known concept is that of uniform reference. The idea is fundamentally that the notation used to reference a data object should be identical wherever a reference to the data object occurs in a program.

The standard BASIC data objects such as strings, arrays, and simple scalar variables obey this rule; this is partly why BASIC is easy to use.

However, there are many circumstances in which BASIC does not provide an appropriate data type. Take the case of a very large array (say 1ØØ by 15Ø elements). Logically, it makes sense to build a BASIC program that uses such a large array, but memory constraints prevent us from building such a program on a microcomputer, because the array itself would occupy more than 9Ø,ØØØ bytes of storage!

A special feature of SD BASIC makes implementation of such data types easy to perform. To define a special data object, an Access Function and an Assignment Subroutine are written. The function name will be the name of the data object, and will be used by the programmer whenever the VALUE of the data object is desired. Arguments to the function are used by the function to select some sub-part of the data object, similar to array indices.

The Assignment Subroutine is used by the programmer to set the value of the data object; the arguments are likewise used to select which part of the data object is modified. The Assignment Subroutine has several constraints placed on it: the subroutine name must be SETXXX where XXX is the name of the Access Function; the number of arguments to the subroutine must be one greater than the number of arguments to the function, and the order and type of all the subroutine arguments (except the last) must be identical to the order and type of the function arguments. The last subroutine argument is the value to be assigned to the subpart of the data object selected. Definition of the subroutine must textually follow the function definition. Invocation of the subroutine must follow its definition, or the compiler will complain.

When the compiler encounters a reference to the function in an expression, it is compiled as usual (see DEF). Occurrence of the function name to the left of an "=" sign of a LET statement (or as a target of a READ or INPUT) causes the compiler to call the subroutine with the name SETXXX, with the last argument being the value of the expression to the right of the = sign (the value READ or INPUT).

If we have the following definitions:

```
DEF F(...)...
SUBROUTINE SETF(...)
...  then
F(args)=exp
```

is translated as:

```
CALL SETF(args,exp)
```

Example:

```
REM VIRTUAL ARRAY DEMO
DIM ROWSIZE/1ØØ/,COLUMNSIZE/15Ø/

DEF VIRTUALARRAY(ROWINDEX,COLUMNINDEX)
  READ #1@(ROWINDEX*COLUMNSIZE+COLUMNINDEX)*6,X
  RETURN X
END

SUBROUTINE SETVIRTUALARRAY(ROWINDEX1,COLUMNINDEX1,VALUE)
  WRITE #1@(ROWINDEX1*COLUMNSIZE+COLUMNINDEX1)*6,VALUE
  EXIT SUBROUTINE
END

  OPEN #1,"VIRTUALARRAY"
  REM FILL THE ARRAY
FOR I=1 TO ROWSIZE
  FOR J=1 TO COLUMNSIZE
      LET VIRTUALARRAY(I,J)=RND
  NEXT J
NEXT I
LOOP:
  INPUT "PICK A PLACE..."I,J
  PRINT "IT CONTAINS";VIRTUALARRAY(I,J)
  INPUT "CHANGE TO:" VIRTUALARRAY(I,J)
  GOTO LOOP
END
```

Another Example:

```
REM 16 bit integer array with 1ØØØ slots
DIM SIXTEENBITINTEGERVECTOR$(2ØØØ)

DEF SIXTEENBITS(X1)=SIXTEENBITINTEGERVECTOR$(2*X1)**8...
&    +SIXTEENBITINTEGERVECTOR$(2*X1+1)

SUBROUTINE SETSIXTEENBITS(X2,V)
   LET SIXTEENBITINTEGERVECTOR$[2*X2]=V**-8
   LET SIXTEENBITINTEGERVECTOR$[2*X2+1]=V&:FF
   EXIT SUBROUTINE
END
```

FILE I/O

The Software Dynamics BASIC supports powerful I/O facilities for dealing with random and sequential files. Statements are provided for opening and closing files, creating, renaming, deleting, reading and writing in both ASCII and binary modes to such files, file positioning, and program chaining.

Software Dynamics BASIC uses channel-directed I/O. File names are associated dynamically with a specific (I/O) channel number, and then I/O to the desired file is performed using the associated channel number instead of the file name (note that a file name must include any decimal device specification). SD BASIC supports up to 256 I/O channels, although the operating system may limit this to a smaller value (usually 8). All channels are assumed by BASIC to be both read and write.

The special channel number Ø always refers to the user's console. All simple PRINT and INPUT statements automatically direct their I/O to channel number Ø, as do all error messages. SD BASIC assumes that both read and write operations to the same file are valid; it is the responsibility of the SDOS I/O package to discover any discrepancies between this philosophy and the way a physical device operates.

SD BASIC I/O inherits many properties of the SDOS I/O philosophy; in particular, SD BASIC's view of files is that each file is a very large string of bytes. A file position indicates where in the string the next read or write will occur; performing a read or write will advance the pointer past the data read or written. Operations exist to change the current file position, so that random file access can be obtained. SD BASIC's file capabilities are limited to those provided by SDOS; it is suggested that the reader refer to the section on Device Independent I/O in the SDOS manual for finer detail. Note that any possible SDOS error may occur as a response to a BASIC I/O operation. Well-constructed application programs will be prepared to handle the most common of these errors (refer to the section on ERROR HANDLING).

OPEN

The OPEN statement is used to associate a channel number and an already existing file. The form is:

```
OPEN #exp, stringexpression
```

The exp must be numeric, and must round to an integer in the range Ø to 255. The string expression results in a file name in the form of a character string. The OPEN statement causes the operating system to open the file named and associate the channel number with all further I/O operations directed at the file. A file must be opened before I/O can occur to the file (by definition, channel Ø, the user's console, is always open). If the file cannot be opened, an error occurs.

Examples:

```
10 OPEN #2,"MYFILE"

20 INPUT FILE$\ OPEN #PAYROLLFILE,FILE$ CAT '.EXT'

30 IF ERROR WHEN OPEN #1, "DATAFILE"
   THEN IF ERR=1011 THEN PRINT "CAN'T OPEN 'DATAFILE'\EXIT
                     ELSE ERROR
```

CREATE

The CREATE statement is used to create a new file and associate a channel number with that newly created file. The form is:

```
CREATE #exp, stringexpression
```

The only difference between CREATE and OPEN is that CREATE requests the operating system to create a new file (to write into). This newly created file is then opened.

One must typically CREATE (not OPEN) a new file on a sequential-only output device such as a line printer or paper tape punch.

Example:

```
10 CREATE #47,"OUTPUT"
```

Under SDOS, CREATE will cause an already-existing disk file of the name to be implicitly deleted; the newly-created file takes its place. No error occurs.

CLOSE

CLOSE breaks the association between a file and a channel number. Once a channel has been closed, it may be re-opened for use with another file (by another OPEN or CREATE statement). The form is:

```
CLOSE #exp, #exp, #exp...
```

Each #exp specifies a channel number to be closed. All buffers for the associated file are logically written to the file if modified; other buffers are freed.

Example:

```
1Ø CLOSE #1Ø, #PAYROLLFILE
```

BASIC automatically closes all files upon program termination of any kind.

DELETE

The DELETE statement is used to delete a file. The form is:

```
DELETE stringexpression
```

The file whose name is the value of stringexpession is deleted. It is legal, but not generally a good idea to delete a file which is still open on some channel.

Example:

```
1Ø INPUT "GET RID OF: " NAME$
2Ø DELETE NAME$
```

RENAME

The RENAME statement is used to rename a file. The form is:

```
RENAME stringexp1, stringexp2
```

The file whose name is the value of stringexp1 is renamed so that its new name is the value of stringexp2.

Examples:

```
1Ø RENAME "D2:TEMP","D2:QUALITYDATA"

2Ø RENAME "JUNK2",Q$[2*J+1,6]
```

RENAME requires an I/O channel be available when executed. The I/O channel used is closed after RENAMEing is completed.

PRINT #

The PRINT to a channel statement is used to print data in an ASCII string form to a file. The form is:

```
PRINT #exp, printlist
```

For PRINT USING to a file, the form is:

```
PRINT #exp, USING formatstring, printlist
```

The formatstring can be the same as in a simple PRINT USING statement, including a line number. The #exp specifies the channel on which the output is to be printed. The PRINT # (USING) statement operates on the print list exactly the same way as a regular PRINT (USING) statement, except that all output is directed to the file previously opened on the specified channel (for tabbing purposes, each channel maintains its own column count). If the print list is null (an empty print), the comma following the channel number expression must be omitted.

Examples:

```
1Ø PRINT #3, "X: ";X, "X^2: ";X*X

2Ø PRINT #MYFILE, A$[1,19];'**'; TAB(1Ø2);\ GOTO 7ØØ

3Ø PRINT #OUTPUT

4Ø PRINT #6, USING "##.#>2", PI

5Ø PRINT #MYFILE, USING 2Ø, SALARY, PERSON$

8Ø PRINT #OUTPUT, USING "$###.##", WEEKLYPAY;\GOTO 79
```

INPUT #

The INPUT from a channel statement is used to input ASCII line data from a file. The form is:

```
INPUT #exp, variablelist
```

#exp specifies the channel from which an input line is to be taken. No prompt is issued (in contrast with the case of a simple INPUT statement where the prompt is printed on channel Ø). A line is read from the specified channel (a line is all characters up to and including a <CR> code, hexadecimal $ØD). The values on the input line are converted and placed into the variables in the variable list exactly as in a simple INPUT statement (including the operation of string input; the <CR> character is discarded). All values required by an INPUT # statement must occur within the single line read from the file, or a conversion error occurs. All values within the line not required to satisfy the INPUT request are discarded.

Examples:

```
10  INPUT #2, A, B[A]

20  INPUT #SOURCE, LINENO, LINE$
```

If a conversion error occurs while INPUTing a value, and error trapping is enabled, then the error will be trapped. If error trapping is not enabled, and input is being performed on channel zero (i.e., to the console device), then BASIC will print 'Input Error!' and ask for all values again. Otherwise, the program will be aborted by a conversion error. The input line is read into the CATBUF; if the input line is too long, an Input Buffer Overflow error occurs and the partial line read is lost.

The fact that PRINT usually places a <CR> at the end of its output, and INPUT stops reading on a <CR>, can be used to input and output variable length records. The input record size must be larger than the longest output record. One may need to trap input conversion errors when variable length lists of numbers are input in this manner.

An end-of-file condition occurs in an INPUT statement when there is not enough data remaining in the file to fill the variables in the variable list.

If End of File is encountered by an INPUT statement, the statement is aborted; the values of variables in the INPUT list is undefined. If the channel number is less than 32, control passes to the statement following the INPUT statement; no error is reported, but the End of File flag for that I/O channel is set. The programmer must check for EOF (see EOF).

Example:

```
INPUT #3,X,Y,Z
IF EOF(3) THEN NOMOREDATA
```

If the channel number is >=32 and End of File is encountered, then an End of File error trap occurs.

WRITE #

The WRITE to a channel statement is used to move binary data to a file. This is advantageous from a speed and space point of view if another program must later read the data back, or if specially formatted files must be built. The format is:

```
WRITE #exp, exp1, exp2, exp3...
```

The #exp specifies the channel on which data is to be written. WRITE causes each expression in the list to be evaluated, and the binary pattern corresponding to the value is copied to the file as a sequential byte stream.

If an expression is numeric, 6 bytes are copied (since B$[X] is considered a numeric expression when B$ is a string variable, "WRITE #2, B$[X]" also causes 6 bytes to be written). The format and content of the bytes are shown in the section on data structures. The READ command allows these numbers to be read back without knowing their format, so the format really isn't important unless something other than a BASIC program is going to process the resulting file.

If the expression is a string, the string value is copied, byte for byte, for the (current, not dimensioned) length of the string, to the channel for writing. Bytes are copied from the string to the file from left to right (i.e., from smaller subscripts to larger subscripts).

The WRITE statement does not insert blanks, <CR> marks, or any data other than the binary image of the expressions evaluated, into the byte stream written. Note that PRINT and WRITE commands can be used on the same channel without ill effects (except perhaps, what happens to the column count for that channel). WRITEing to a channel causes the COLumn count for that channel to become invalid; it can be reset to one by executing an empty PRINT on that channel.

WRITEing to a CRT device will cause that device driver to lose track of the cursor. We recommend against performing WRITEs to the screen because this ties the application program to the type of CRT being used. Use of the SDOS Virtual Terminal driver can help make applications CRT-independent.

Examples:

```
10 WRITE #2, A[I,J], S*3, B$[2]

20 WRITE #MYFILE, LLINK, RLINK, NAME$\ WRITE #2, X

30 WRITE #EMPLOYEE,COUNTY$[1,10],SALARY,PHONE,ADDRESS$

40 WRITE #FILE,LEN(X$), X$\!WRITE VARIABLE LENGTH STRING

50 WRITE #FILE,A$\PRINT #FILE,NUMF$("##.##"),X

60 WRITE #FILE,B$\PRINT #FILE,TAB(25),...\!UNDEFINED TAB ACTION

70 ONEBYTE$(1)=:C\LEN(ONEBYTE$)=1\WRITE #FILE,ONEBYTE$\
   !WRITE A SINGLE BYTE CONTAINING :C
```


READ #

The READ from a channel statement is used to read binary data from a channel (usually from a file written by a BASIC program that used WRITE statements). Trying to use an INPUT statement on binary data is a sure-fire way to cause a conversion error. The form is:

```
READ #exp, variable, variable, variable...
```

The #exp specifies the channel from which binary data is to be read.

Each variable can be anything which can appear on the left side of an equals sign in a LET statement (i.e., substring, vector element, etc.).

READ causes the specified variables to be filled by reading an appropriate number of bytes in binary mode, and storing into the variables. The variables are filled from left to right in the READ statement.

If the variable is numeric (a simple variable, vector entry or array entry), 6 bytes are read and stored into the variable, in such a way that whatever a WRITE statement wrote, the READ statement reads back correctly (see formats under the section on Assembly Language Interface). Reading into B$[X], where B$ is the name of a string variable, reads 6 bytes and stores the value into B$[X].

If the variable is a substring reference, then the number of bytes specified by the substring length are read and copied into the substring. The current string length must not be exceeded or a subscript error will occur. If the variable is a string reference (no subscripts), the READ will read "the dimensioned length of the string" number of bytes and set the current length of the string to the dimensioned length. For instance, if S$ was dimensioned with a max length of 5, and had a current length of 3, "READ # ... S$" would fill S$ with 5 bytes from the channel specified, and set the length of S$ to 5. "READ # ... S$[1,3]" would only read 3 bytes into S$, without affecting its length.

If a READ statement encounters End of File, and the channel number is less than 32, no error is given, and control is passed to the next statement (for channel numbers 32 or greater, an "End of File" error trap will occur). If EOF occurs while filling a string variable, the length of the string variable is set to the actual number of bytes read. The programmer must check for EOF.

If EOF occurs while reading on a channel number greater or equal to 32, then the READ statement is aborted (the values of the variables are left undefined), and an End of File error trap occurs.

Examples:

```
10 READ #DATA, A[I,J],S

20 READ #3, LLINK, RLINK, NAME$

30 READ #EMPLOYEE,COUNTY$[1,10],SALARY,PHONE,ADDRESS$

40 READ #FILE, LEN(X$), X$[1,LEN(X$)]\!READ VARIABLE LENGTH STR
```

POSITION # or RESTORE #

The POSITION on channel command is used to position a file for the next I/O operation (RESTORE is allowed as an alternate keyword to retain compatibility with older versions of SD BASIC, and is not recommended). The format is:

```
POSITION #exp1,exp2
```

The #exp1 specifies the channel on which positioning is to be performed. Exp2 specifies a file-dependent positioning number, usually a record or byte number within the file (see operating system interface).

Example:

```
10 POSITION #2,0\ REM THIS IS NORMALLY A "REWIND"

20 POSITION #DATA,RECORD\ READ #DATA,RECORD$
```

A convenient trick for positioning to records in a file is to define a user function (see "DEF") to compute the actual position in a file of a record. Assume that records in a file are 100 bytes long, and that the first 34 bytes of a file are to be avoided for some reason and that the first record has a logical record number of zero. The following is a program to print "HELLO" in record number 22:

```
10 DEF RECORD(X)= X*100+34
...
100 POSITION #FILE,RECORD(22)\ PRINT #FILE, "HELLO"
```

Positioning the cursor on the console CRT may be accomplished by a POSITION #Ø, DESIREDROW*256+DESIREDCOLUMN statement where the top most screen line is row number zero, and the leftmost screen column is column zero (note that the COL function returns 1 for the leftmost screen position). Proper cursor positioning requires that the SDOS I/O package be properly configured for the particular CRT being used; SDOS systems with the Virtual Terminal driver allow most terminals to be properly configured via the SDOS SET command. Note that WRITEing to a screen causes the system to lose track of the cursor location (because it cannot determine how the written bytes affect the CRT). This can be fixed by issuing a POSITION after each WRITE to the screen; after the POSITION, the system knows where the cursor is again.

IMPLIED POSITIONING

SD BASIC allows a file (or screen) position to be designated in an I/O statement directly via the "@" (at sign) notation:

```
keyword #exp1@exp2,...
        or
keyword #exp1@(exp2,exp3),...
```

The first notation specifies a file position via exp2, and is equivalent to the following:

```
POSITION #exp1,exp2\keyword #exp1,...
```

The second notation is generally used for cursor positioning on a CRT, and is equivalent to

```
POSITION #exp1,(exp2)*256+exp3\keyword #exp1,...
```

where exp2 is the screen row number and exp3 is the screen column number, origin Ø.

The @ notation may be used with channel-less PRINT or INPUT statements in a straight forward way. Note that

```
INPUT "prompt"@(exp1,exp2),varlist
```

is a convenient way to perform screen-oriented data entry.

Examples:

```
3Ø READ #2@RECORD(I),EMPLOYEE$,SALARY,SOCSECURITYNUMBER

4Ø PRINT #2@(1Ø,5),USING "##.##", COSTOFLUNCH

5Ø INPUT "CUSTOMER NAME:" @(5,1Ø),CUSTOMERNAME$

6Ø WRITE #5@KEY(5,1,EMPLOYEE$),NEWSALARY,NEWTITLE$

7Ø POSITION #Ø@(12,15)
```


CHAIN

The CHAIN statement is used to build segmented or overlay BASIC programs. The form is:

```
CHAIN stringexpression
```

The stringexpression is evaluated to produce a file name. That file is assumed to be a program and is loaded into memory. Program control passes to the first instruction in the new program. The program does not need to be a BASIC program. CHAIN (under SDOS) automatically closes all files.

Example:

```
1ØØØ CHAIN "PASSII"
```

Variables may be passed from one BASIC program to another via CHAIN by use of COMMON statements (see DIM, COMMON).

SYSCALL

The SYSCALL statement is used to perform SDOS system calls. The form is:

```
SYSCALL argumentlist
        or
SYSCALL #exp,argumentlist
        or
CALL SYSCALL(argumentlist)
```

where argumentlist consists of one to four arguments. The reader is referred to the SDOS manual for a comprehensive discussion of SDOS syscalls.

The first argument is a string expression and contains the bytes to be used for the body of a syscall block, including the syscall extension, if needed. BASIC will construct a syscall block if the specified syscall block does not have enough room for all the required syscall parameters.

The second argument, if present, is a string expression and is used as a write buffer. The address of the string is placed in WRBUF and the two byte (current) length is placed in WRLEN. For syscalls not requiring a write buffer, an empty string should be specified.

The third argument, if present, must be a string variable or a substring of a string variable, and is considered to be a read buffer. The address of the string is placed in RDBUF and the two byte length (of substring: maxlength if string) is placed in RDLEN as a ceiling upon the number of bytes to read. Upon return, RPLEN contains the number of bytes actually read, and if the argument was a string, the string's LEN is set to this value.

After all the parameters are loaded into the syscall block, the address of the block is loaded into the X register and a "JSR $FB" is executed. A "BCS IOERROR" is the next instruction after the "JSR" so that any error conditions can be signaled by a "SEC/RTS" and a success return is signaled by a "CLC/RTS". The error condition must be passed back in the X register and is made available to the BASIC program through the special function ERR.

The fourth argument, if supplied, is used as the desired length of the RDBUF instead of the value implicit in RDBUF. A subscript error will result if the desired length is larger than the RDBUF string allows.

If a channel number expression is used and has a non-zero value, then it is used as the channel number in the executed syscall instead of the channel number byte given in the syscall block.

Examples:

```
10 DIM DISMOUNT$/:E,4,Ø,:11/,STRING$(1ØØØ),READBON1$/:B,:E,1,Ø/
       ...
100 SYSCALL #1,DISMOUNT$

120 DEF FILESIZE(CHANNEL)
          REM RETURNS SIZE OF FILE OPEN ON "CHANNEL"
          DIM T$(4), GETFILESIZE$/:F,14,Ø,:3/
          SYSCALL #CHANNEL,GETFILESIZE$,"",T$
          RETURN T$[1]*256^3+T$[2]*256^2+T[3]*256+T$[4]
    END

140 SYSCALL READBON1$,"",STRING$,256\! READ ONLY 256 BYTES
```


KEYED FILE PACKAGE

The set of subroutines and functions described in this section comprise a keyed-file package. This package is not a standard part of SD BASIC but can be obtained as an option. This allows access to records by use of a "key" (or record name); a typical use would be to allow the location of a record containing data about an employee by use of his name. Location of a record in a large file using a hard disk typically occurs in under 1 second. Sequential access (KEYNEXT) is under one-half second.

A "key" structure (a B-tree) is built by the key package to help it locate records. The records and key structures are completely independent; they may both be in the same file, or in separate files.

Use of the package is very easy. The function KEY, given a character string, returns a file position where the record associated with that string is located. The statement

```
READ #DATACHANNEL@KEY(KEYCHANNEL,KEYNUMBER,DESIREDKEY$),var,var,...
```

thus reads the contents of the record (from the file selected by DATACHANNEL) whose name is DESIREDKEY$ using the key structure in the file selected by KEYCHANNEL (note: DATACHANNEL can be equal to KEYCHANNEL).

The rest of the subroutines and functions in the package exist to initialize a keyed file, and to perform other necessary support functions. The routines and their descriptions are listed below. The term "keyed file" is used to describe the file containing the key index data. This package allows multiple key indexes to be stored in one file, i.e., a customer invoice file might be keyed by both customer name and by invoice number.

KEYCHANNEL numbers select the channel number of the file that contains the key structure. The KEYNUMBER selects which key category is to be used. A customer name could be used as the first key category, and invoice number as the second key category. Note: all keys in a catagory must be unique. DESIREDKEY$ is a string containing the key desired for use in the operation. The keyed file package internally pads the specified key with ASCII nulls to fill it out to the desired size, or truncates, as needed; padding is performed on the right. Note also that the KEY package is case-sensitive: upper-case characters in key are not equivalent to lower case characters.

Each key category used in a file requires that bytes KEYCATEGORY*KEYCATEGORYHEADERSIZE through KEYCATEGORY*KEYCATEGORYHEADERSIZE+KEYCATEGORYHEADERSIZE-1 of the file be set aside exclusively for use by the keyed file package. Other space is taken from End of File as needed.

KEYINIT(keychannel#,keynumber,keysizeinbytes,branchingfactor)
is a subroutine that initializes a file for operation with the keyed file package. If KEYINIT is called on a file containing keyed data, then the old key structure is destroyed, and the space in the file used for the old key structure will be lost. This routine must be CALLed before any other keyed operation is performed. KEYINIT must be called once for each key category to be used. The branching factor specified must be larger than 4, or a "Key branch factor not large enough" error trap will occur. The branching factor controls the amount of time required to look up a key; the lookup time is roughly equal to the time for k seeks + r reads, where branchingfactor^k>=numberofkeys currently in the key category, and r is approximately (keysizeinbytes*branchingfactor)/sectorsize.

KEYINSERT(keychannel#,keynumber,desiredkey$,recordlocation)
is a subroutine that accepts a string argument containing a key and a record position, and adds information to the key structures, so the record at recordlocation may be retrieved via the KEY function when applied to the identical desiredkey$. A "Duplicate key" error trap occurs if that key already has an associated record.

KEY(keychannel#,keynumber,desiredkey$)
is a function that returns the position in the data file of the record selected by that key. If no such record, a "No Such Key" error trap occurs.

KEYDELETE(keychannel#,keynumber,desiredkey$)
is a subroutine that deletes a record from a keyed file. A "No Such Key" error occurs if the key does not exist. If a record is keyed on more than one category, KEYDELETE must be called once for each category with the proper key value for that category.

KEYNEXT(keychannel#,keynumber,desiredkey$)
is a function that returns the file position of the record whose key is the smallest key greater than "desiredkey$". Desiredkey$ is modified to contain the key of the record so found. An EOF error trap indicates the list of records in the file has been completely processed. To fetch the first record of a file, the null key (all zeros) should be used as the value for desiredkey$. Repeated use of KEYNEXT thus scans the records alphabetically.

KEYREPLACE(keychannel#,keynumber,desiredkey,newrecordlocation)
is a function that replaces the old record location of a key with a new value; its result is the value of the record location being replaced. Its effect is identical to invoking KEY, then KEYDELETE, followed by KEYINSERT except it is considerably faster. The specified key must exist or a "No Such Key" error will occur.

GETSPACE(channel#,numberofbytes)
is a function that extends the filesize by "numberofbytes" and returns the file size before it was extended. This is useful for quickly finding space for a new record.

Sample program:

```
DIM RECORDKEY$(1Ø),RECORDDATA$(8Ø),ANS$(1)
INCLUDE "KEY.BAS"
CREATE #1, "DATABASE"
CALL KEYINIT(1, 1, 1Ø, 32)
DATAENTRYLOOP:
         INPUT   "KEY: " RECORDKEY$
         IF      RECORDKEY$="" THEN UPDATEMODE
         INPUT   "DATA TO STORE: " RECORDDATA$
         RECORDLOCATION = GETSPACE(1,LEN(RECORDDATA$)+1)
         REM ...+1 in previous statement accounts for
         REM <CR> introduced by following PRINT
         PRINT #1@RECORDLOCATION,RECORDDATA$
         CALL    KEYINSERT(1,1,RECORDKEY$,RECORDLOCATION)
         GOTO    DATAENTRYLOOP
UPDATEMODE:
         INPUT   "LOOKUP: " RECORDKEY$
         IF      RECORDKEY$="" THEN PRINTSEQUENTIAL
         IF ERROR WHEN
            INPUT #1@KEY(1,1,RECORDKEY$), RECORDDATA$
         THEN
            PRINT "NO RECORD FOUND"
            GOTO UPDATEMODE
         FI
         PRINT "DATA = "; RECORDDATA$
         INPUT "CHANGE? " ANS$
         IF ANS$="" THEN UPDATEMODE
         INPUT "NEW DATA: " RECORDDATA$
         PRINT #1@KEY(1,1,KEY$), RECORDDATA$
         GOTO UPDATEMODE
PRINTSEQUENTIAL:   LET RECORDKEY$=""
PRINTSEQLOOP:
         IF ERROR WHEN
            INPUT #1@KEYNEXT(1,1,RECORDKEY$),RECORDDATA$
         THEN EXIT
         PRINT   RECORDKEY$; RECORDDATA$
         GOTO    PRINTSEQLOOP
END
```

MISCELLANEOUS STATEMENTS

DIM

The DIM statement is used to allocate space for variables used by the BASIC program, including scalars (simple numeric variables). DIM statements have no effect at run-time. The form is:

```
DIM declaration,declaration,declaration...
```

DIM statements do not allow multiple statements per line. Each declaration is either a simple numeric variable name, the name of a vector or array with its dimensions, the name of a string and its maximum dimensions or the name of a string array followed by the number of strings in the array, and the maximum dimension of any of the strings in the array.

SD BASIC always sets aside space for a "zeroth" index slot, row or column in a vector, array or string array. Strings always have a lower index of 1.

The form of each declaration is as follows:

```
Scalar:          name
Vector:          name[numberofslots]
Array:           name[numberofrows,numberofcolumns]
String:          name$[maxlenincharacters]
String Array:    name$[numberofstrings][maxlenincharacters]
```

All strings of a string array are allocated the same maxlenincharacters maximum size.

Although scalar variables need not be dimensioned, there are two reasons for doing so. First, the first 64 scalar variables mentioned in a BASIC program are assigned very small compile time code references; thus scalars used frequently in a program should be DIMed to save space. Secondly, a DIMed scalar variable may be allocated with any initial value (a compile time assignment) by writing

```
DIM name/value/
```

The value may be a hex or numeric constant. Similarly, a numeric vector can be given a set of initial values by writing

```
DIM name/value1,value2,.../
```

The dimension is implicit in the number of values given, with the first value being assigned to subscript Ø of the vector.

String variables may also be initialized similarly; if no dimension is specified then the dimension is implicit in the length of the initializing constant. A string initializing value may be a list of constant strings and/or hex values; the string is filled from left to right with the list contents. Each hex value specified occupies a single byte of storage.

Each time the program is run (or executed via CHAIN), the initialized variables are reset to the values specified in the DIM statement(s). Simple numeric variables without initial values contain garbage when the program is started, as do uninitialized strings. The current length of an uninitialized string is zeroed. This will usually cause subscripting errors if the string is used before it is set to a valid value. Arrays, vectors and string arrays cannot be initialized.

Examples:

```
        1Ø DIM A,B,I,J,DUMMY,EMPNO

        2Ø DIM VECTOR[9],ALPHA,OUCH[2,7],B$[47]

        3Ø DIM S3/7.2/, B9/-3/,QSTR$/:3,"ABC",:D,"DEF"/

        4Ø DIM FILENAME$[1Ø]/"SALES.TAX"/

        5Ø DIM ADGNLB$/" PMR    PMI",...
&          "    PMS"/
           REM LINE 5Ø SETS ADGNLB$ = " PMR    PMI    PMS"

        6Ø DIM SCREEN$(24)[8Ø]

        7Ø DIM PRIMESUNDER32/2,3,5,7,9,11,13,17,19,23,29,31/
```

Except for scalars, the compiler will complain if a variable is not mentioned in a DIM statement (or COMMONed, or declared as a parameter variable in a SUBROUTINE or function definition). Furthermore, DIM statements must be collected at the front (top) of the main program or at the beginning of a multiline function or SUBROUTINE, before any other executable statements. REM and DIM statements may be mixed in any order at the front of the program.

Local DIMs, i.e., those in SUBROUTINEs or functions, must appear immediately following the SUBROUTINE or function header line. If a DIM is in a SUBROUTINE (or function) definition, the initialization of its variables is done each time the SUBROUTINE (or function) is called. Reference to a variable mentioned in a local DIM statement is illegal outside the body of the rountine containing the local DIM.

COMMON

The COMMON statement allows the program to pass variables between CHAINed program segments. Like DIM, it is used to allocate storage space for variables (but DIM'd variables cannot be passed between program segments). Scalar variables must be specified in a COMMON statement to be passed between CHAINed segments. No initialization of variables is allowed. COMMON statements are not allowed in SUBROUTINE or function definitions.

Variables must be COMMONed in the same order, and with the same dimensions in all the program segments for this to work. Further, a DATA ORIGIN statement must be used to place data at a fixed place; the origin point must be the same in all chained program segments. Since BASIC is a compiler, no check is made when CHAIN is invoked to ensure that the order of the declared variables match, or that the types match. Failure to declare COMMON correctly can lead to unpredictable results at execution time.

```
1Ø COMMON A,B$[46]

2Ø COMMON S2, Hello, Passed Vector[1ØØ], FILEEXISTSFLAG
```

See DATA ORIGIN for example of COMMON in CHAINed program segments.

Both COMMON and DIM statements may be used in a program. The DIM statement must follow any COMMON statements used.

PROGRAM ORIGIN

The PROGRAM ORIGIN statement is used to change the location the compiler will place the program in the computer's memory. The form is:

```
PROGRAM ORIGIN hexnumber
```

A PROGRAM ORIGIN statement may not have a line number.

Normally, the compiler begins placing statements directly above the space allocated for the runtime package. The PROGRAM ORIGIN statement may be used to override this, and place the program code anywhere. If the DATA ORIGIN statement is used to control the placement of data storage, it is a good idea to use the PROGRAM ORIGIN statement to ensure that the program code does not overlap the data area. The PROGRAM ORIGIN statement must occur before a DATA ORIGIN, DIM or any executable statements and cannot be used in a function or SUBROUTINE defined in a main program. Only one PROGRAM ORIGIN statement is allowed in a program.

Example:

```
1Ø REM ANOTHER BASIC PROGRAM
   PROGRAM ORIGIN :3ØØØ
2Ø ! HERE COME THE DIM STATEMENTS
3Ø DIM Q7[47],B,J,X
4Ø DIM ...
   ...
1ØØ REM NOW FOR THE PROGRAM ITSELF
11Ø PRINT "HELLO..."
   ...
```

DATA ORIGIN

The DATA ORIGIN statement is used to change where the compiler will place the variable storage for a program. The form is:

```
DATA ORIGIN hexnumber
```

A DATA ORIGIN statement may not have a line number.

Normally, the compiler starts allocating space for variables at the end of the program. The DATA ORIGIN statement can be used to override this default, and set the allocation base to a specified hexadecimal address. The variable space is allocated as one single block, so wherever you place the DATA ORIGIN, there must be enough RAM for your variables (including machine stack and Concatenation Buffer; see DEBUGGING A COMPILED PROGRAM). This statement is normally used in a set of program segments to ensure that COMMON variables are aligned properly. The DATA ORIGIN statement must be placed in the program before any DIM statements and after a PROGRAM ORIGIN statement if used. A program may not have more than one DATA ORIGIN statement.

Example:

```
1Ø REM THIS IS "FIRST PART"
2Ø REM ...
   ...
   DATA ORIGIN :48ØØ
   ...
1ØØ COMMON Q$(46),B[12,95],...
   ...
   CHAIN "SECONDPART"
END


1Ø REM THIS IS "SECOND PART"
   ...
   DATA ORIGIN :48ØØ
   COMMON EMPLOYEE$(46), MONTHVSACCOUNT(12,95)
   REM EMPLOYEE$ GETS VALUE OF Q$ SET IN FIRST PART
   REM MONTHVSACCOUNT GETS VALUES FROM 8 SET IN FIRST PART
   ...
```

CONCATENATION BUFFER SIZE

The Concatenation Buffer Size statement is used to select the size of the concatenation buffer. The form is:

```
CONCATENATION BUFFER SIZE = decimalconstant
```

The concatenation buffer is used for two purposes: to hold a temporary result while concatenating a string, and as an input buffer for the INPUT statement. Program statements that invoke use of the concatenation buffer while it is already being used will get unexpected results. Since there is only one concatenation buffer, such programs are illegal. An example illegal program (statement) is:

```
RENAME A$ CAT "txt", A$ CAT "DOC"
```

If no Concatenation Buffer Size is used, the compiler defaults the concatenation buffer size to 256 bytes, large enough for virtually all normal use. The Concatenation Buffer Size statement, if used, must come after any PROGRAM ORIGIN and before any DATA ORIGIN statements, and before any COMMON or DIM statements. This statement may only be used in a main program.

Example:

```
REM PROGRAM THAT INPUTS LINES OF NOT MORE THAN 3ØØ BYTES
CONCATENATION BUFFER SIZE = 3Ø1
   ...
DIM LINE$(3Ø1)
   ...
INPUT #1, LINE$
```

INCLUDE

INCLUDE is a compiler directive that is used to select a new input file. The form is:

```
INCLUDE stringconstant
```

INCLUDE causes the current compiler source input file to be suspended. Input is then taken from the file specified by the string constant. When end of the included file is reached, the file is closed and compiler source input resumes from the previous file. INCLUDEs may be used to include COMMON or DIM statements, blocks of executable code, or subroutine packages.

INCLUDE statements may be nested, and the deepest nesting is determined by the number of I/O channels available minus 3 (console, output file, and original input file); generally it is limited to 4 levels. INCLUDE statements may be placed anywhere, but may not have line numbers or labels, nor may they be part of a multi-statement line.

Example:

```
REM PASS 2 OF MULTIPASS PROGRAM
  ...
INCLUDE "COMMONDIMS"
DIM A$(5Ø)
  ...
INCLUDE "KEY.BAS"
  ...
END
```

If the first significant line of a source file is an INCLUDE statement which references a file containing a set of DEF functions and/or SUBROUTINEs, BASIC will assume Separate Compilation of the first routine is what was desired. Placement of a DIM statement before the INCLUDE will convince BASIC that this is really a main program, and that the included file should be compiled in its entirety.

END

The END statement is used to mark the end of a BASIC program. It must be the last line of the program. A line number is allowed, but cannot be the target of a GOTO, GOSUB, or ON statement. The form is:

```
END
```

Example:

```
1Ø REM ADDITION PROGRAM
2Ø DIM A,B
3Ø INPUT "A,B" A,B
4Ø PRINT A+B
5Ø GOTO 3Ø
6Ø END
```

The END statement is also used to terminate blocks (see BLOCK BODIES).

ASSEMBLY LANGUAGE INTERFACE

The Software Dynamics BASIC also interfaces to assembly language routines. This section describes statements for doing so, and data formats for the information passed.

ASSEMBLY LANGUAGE SUBROUTINES

When CALLing assembly code, the subroutine name should appear in a SUBROUTINE NAME (...) EXTERNAL statement before it is used, to insure compatibility with future versions of SD BASIC.

The compiler assumes a subroutine with the specified name will be supplied to pass II. The arguments given in the CALL statement are expressions; the addresses of these expressions are pushed onto the "machine" stack to create an argument list.

The subroutine is passed a pointer to the last argument in the argument list in the X register (if the argument list was at :49A4, X would contain :49A4), and an argument count is passed in the A register. Since entry to the subroutine is made by a JSR, an exit via a RTS (with the carry reset) is expected. No other action is required by the subroutine (the argument list is automatically removed from the machine stack by BASIC). To pass an error back to the BASIC program, the error code needs to be loaded into the X register, the carry bit must be set, and an RTS performed (this is consistent with SDOS error handling). See the section on Data Structures, below, for more detail.

Each stack entry is 6 bytes in length, and can hold any of the following kinds of data:

1) Address of a floating point number
2) Address of a 16 bit positive integer
3) Numeric variable addresses
4) String descriptors
5) Address of vector or array

The compiler makes no guarantee that all CALLs will push the same number and/or type of arguments onto the value stack. The called assembly subroutine must either assume a debugged BASIC program or check the argument count and parameter types itself.

If an argument is a numeric expression, then a temporary location is allocated to hold the expression result, and the address of the temporary location is pushed onto the stack.

If the argument is a numeric variable name or a vector or array entry, the address of the 6 byte region of memory, to which that variable has been assigned, is pushed.

If the argument is a string variable or a string expression, a string descriptor is pushed onto the stack.

The user subroutine can pass data back to the program by storing a value into a numeric variable, or into a character string as specified by a string descriptor. Page zero locations Ø through 7 are available for use by the subroutine.

Assembly Subroutine Example:

```
SUBROUTINE INITIALIZEPIA(PIAADDRESS,CRACRB) EXTERNAL
       ...
       CALL INITIALIZEPIA(:F2B4,:3DØ4)
       ...
       END

* This is the assembly code for INITIALIZEPIA
* Combining this with the compiled BASIC program
* is described in the section HOW TO USE SD BASIC
*
INITIALIZEPIA  ; ASSUME ARGS ARE BOTH INTEGERS
       CMPA    #2
       BNE     INITWRONGARGCOUNT
       STX     $Ø      SAVE ARG LIST POINTER
       LDX     4,X     GET ADDRESS OF INITIALIZING CONSTANTS
       LDD     4,X     FETCH CRA,CRB VALUES
       LDX     $Ø      GET ARG LIST POINTER BACK
       LDX     1Ø,X    FETCH ADDRESS OF PIA ADDRESS
       LDX     4,X     FETCH PIA ADDRESS
       STA     Ø,X     INITIALIZE CRA OF PIA
       STB     2,X     INITIALIZE CRB OF PIA
       CLC             SIGNAL SUCCESS
       RTS
INITWRONGARGCOUNT
       LDX     #27     "WRONG ARG COUNT" ERROR
       SEC             SIGNAL FAILURE
       RTS
```


ASSEMBLY LANGUAGE FUNCTIONS

User-defined assembly language functions can also be used with SD BASIC. Any result returnable by a conventional function may be returned by an assembly function. The function name must be declared using a DEF name (...) EXTERNAL statement textually preceding any attempt to invoke the function. Invocation syntax is identical to that for invoking a conventional BASIC function.

Parameters are passed to the assembly function in exactly the format specified under Assembly Language Subroutines. Assembly functions can modify passed arguments. Page zero locations Ø-7 are available for use by the function.

Function results are returned on top of the machine stack, and must be either integer, floating point, or string descriptor values (See DATA STRUCTURES). This means the return address must be removed; the result pushed, and then control passed to the return address with the carry reset.

The runtime package handles removing the argument list from the stack after the function returns. Errors are signaled as described under Assembly Language Subroutines.

Assembly Function Example:

```
DIM POINTER$(4)
       ...  DEF CVT32BITSTOFLOAT(FOURBYTES$) EXTERNAL
       ...
       READ #6, POINTER$
       POINTER= CVT32BITSTOFLOAT(POINTER$)
       ...  END

*This is the assembly code for CVT32BITSTOFLOAT
*This function accepts a string descriptor
*The string contains 4 bytes, and represents a 32 bit binary integer
*The function returns a floating number equal in value to the integer
CVT32BITSTOFLOAT       EQU     *
       CMPA    #1      CHECK ARGUMENT COUNT
       BNE     CVT32BITBADARGCNT
       PULD            REMOVE RETURN FROM STACK...
       STD     CVT32BITSRETURN
       LDX     2,X     =ADDRESS OF STRING,-4
* We assume a 4 byte (or more) string is passed,
* so we don't check the count.  Caveat Emptor.
       LDD     4+2,X   FETCH LEAST SIGNIFICANT 16 BITS
       PSHS    D       CONSTRUCT 32 BIT INTEGER ON STACK
       LDD     4+Ø,X   FETCH MOST SIGNIFICANT 16 BITS
       PSHS    D
       JSR     FLOAT   GO FLOAT 32 BIT NUMBER
       LDX     CVT32BITSRETURN
       CLC             SIGNAL SUCCESS
       JMP     Ø,X

CVT32BITBADARGCNT
       LDX     #27     "WRONG ARG COUNT" ERROR
       SEC
       RTS

CVT32BITSRETURN
       RMB     2       PLACE TO STORE RETURN ADDRESS
```

DEBUG

The DEBUG statement is used to call the user's assembly (or whatever) language debugger via SYSCALL:DEBUG. The form is:

```
DEBUG
```

This statement can be used anywhere in the BASIC program. The debugger itself can be used for anything, but must exit as described in the SDOS manual if the BASIC program is to continue. This statement is most useful when used to help debug assembly language functions or SUBROUTINEs.

POKE

The POKE statement is used to allow the BASIC program to do direct memory stores (usually to an I/O device). The form is:

```
POKE expl,exp2
```

The value of exp2 is stored in the byte at the address specified by the value of expl. The runtime package will not poke itself.

Examples:

```
1Ø POKE 643Ø2,26

2Ø POKE RAM+7,B$[2]

3Ø POKE :4Ø67,:3

4Ø POKE IOPORT1, :41

5Ø RUSSIANROULETTE: REPEAT POKE RND*65535,Ø
```

BUILT-IN FUNCTIONS

A built-in function is a pre-defined routine to compute a value to be used in an expression. BASIC supports many built-in functions. Each function invocation is written as:

```
...name...
```

Or

```
...name arg1...
```

Or

```
...name(arg1)...
```

Or

```
...name(arg1,arg2,arg3...)...
```

Where "name" is the name of the function, and arg1, arg2,... are the values needed by the function. Each of the values can be an arbitrary expression except as noted for the particular function. Like user defined functions, if there are no parentheses surrounding the argument list, then the argument list is assumed to have one element, i.e., 3*ATN X+2 is the same as 3*ATN(X)+2.

The transcendental functions are generally accurate to 7 places. The arguments for the trigonometric functions are required to be in radians except for the arc tangent, which yields a result in radians.

SQR(arg)

produces the square root of the argument.

ATN(arg)

produces the arc tangent (in radians) of the argument. Other inverse trigonometric functions may be obtained by using the following user defined functions:

```
DEF ARCSIN(X) = ATN(X/SQR(1-X*X))
DEF ARCCOS(X) = PI/2 - ARCSIN(X)
```

SIN(arg)

produces the trigonometric sin of the argument.

COS(arg)

produces SQR(1-SIN(arg)^2) [this is not how it is implemented].

TAN(arg)

produces the trigonometric tangent of the argument value.

LOG(arg)

produces the natural logarithm of the argument.

EXP(arg)

produces 2.7182818...^arg (exponential).

RND

produces a random number >= Ø and < 1. Note: no argument is needed. The random number generator can be made to repeat its sequence by setting RND to a fixed value before using RND to generate a set of random values. The form is

```
LET RND = expression
```

Example:

```
150 LET RND=5
```

will cause the same sequence of values to be generated for each execution of the program. To make a truly non-repetitive sequence, setting RND to some value based on the current time of day is appropriate.

ROWS(arrayname)

returns the number of rows specified by the DIM or COMMON statement of the array (this is especially useful when arrayname is a parameter variable; see DEF).

COLUMNS(arrayname)

returns the number of columns specified by the DIM or COMMON statement of the array. (See ROWS function).

LEN(stringname$)

produces the current length of the string value stored in the specified string variable. (stringname$ may be a singly subscripted string array).

LEN(vectorname)

returns the DIM'ed (or COMMONed) size of the specified vector name.

LEN(stringarrayname$)

returns the number of strings DIM'ed or COMMONed for this array.

MAXLEN(stringname$)

produces the maximum (i.e., DIMensioned) length of the string variable specified.

MAXLEN(stringarrayname$)

returns the maximum (i.e., DIM'ed) length of any string in the specified string array.

MID$(stringname$,arg2,arg3)

is exactly the same as:

```
stringname$[arg2,arg3]
```

It may appear on the left side of an equals sign in a LET statement (or as a target of a READ or INPUT statement). Since MID$ is a subscripting operation, it may not itself be subscripted.

LEFT$(stringname$,arg2)

is exactly the same as:

```
stringname$(1,arg2)
```

It may appear on the left side of the equals sign in a LET statement (etc.) Since LEFT$ is a subscripting operation, it may not itself be subscripted.

RIGHT$(stringname$,arg2)

is exactly the same as:

```
stringname$(arg2,LEN(stringname$)-arg2+1)
```

It is considerably faster, however. It may appear on the left side of an equals sign in a LET statement (etc.). Since RIGHT$ is a subscripting operation, it may not itself be subscripted.

UPPERCASE$(stringexpression)

is a string function which produces a temporary string (in the concatenation buffer) that is an exact copy of the string argument except that all ASCII lowercase alphabetic characters are converted to uppercase. The string argument may only be a string name, substring, or string function. Since this function uses the concatenation buffer, the string argument may not use a concatenated expression, and the function may not be used in a concatenated expression.

Example:

```
INPUT RESPONSE$
IF FIND ("YES", UPPERCASE$(RESPONSE$))=1
THEN DoWhatHeWanted ELSE AskForAnotherCommand
```

LOWERCASE$(stringexpression)

is a string function which produces a temporary string (in the catenation buffer) that is an exact copy of the string argument except that all ASCII uppercase alphabetic characters are converted to lowercase. The string argument may only be a string name, substring, or string function. Since this function uses the catenation buffer, the string argument may not use a concatenated expression, and the function may not be used in a concatenated expression.

EOF(arg)

The argument is evaluated and used as a channel number. The function produces true if the last READ or INPUT attempted to read past the end of the file on the specified channel. Otherwise, produces false. Since READs and INPUTs cause an "End of File" error trap upon reading across the end of file for channel numbers larger than 32, an arg >=32 for EOF is unnecessary and a "Channel number is too large" error will occur.

COL(arg)

produces the current column number on the channel specified by "arg". The column number is the column in which a logical print head would be positioned at the instant of the call. The leftmost column is column number 1. If an input line is only partially read, then the COL function applied to the INPUT channel will return a value greater than the column number of the first input character.

PEEK(arg)

returns value equal to that of the memory byte whose address is specified by the value of the argument expression. Normally, the argument is a hex constant.

COM(arg)

returns an integer whose binary equivalent is the bitwise complement of the binary equivalent of the argument. If the argument is not an integer in the range Ø to 65535, an error occurs. Examples: COM(Ø)=65535, COM(17)=65518, COM(32767)=32768

NOT(arg)

returns the logical complement of the argument (true produces false and false produces true). The argument of this function must be a relational expression (such as A=B etc.) or a compound logical expression as described in the section on Conditional Expressions. Example: NOT (B>6 OR DIVISOR=Ø)

INT(arg)

produces the largest integer not greater than the argument. Example: INT(3.2)=3; but INT(-3.2)=-4.

ABS(arg)

produces the absolute value of the argument.

SGN(arg)

returns the sign of the argument. If the argument is < Ø, the function returns -1. If the argument is = Ø, the function returns Ø. If the argument is > Ø, the function returns +1.

ERR

produces a value corresponding to the most recent error encountered at runtime at this level of the program (errors trapped and handled in called subroutines or functions cannot be seen). See section on error messages.

ELN

produces a value corresponding to the last line number executed, or in execution, when the last error occurred.

PI produces the value 3.14159265

ASC(stringexpression)

This function returns the numeric value of the ASCII code of the first byte of the stringexpression specified. Examples: ASC("A") gives decimal 65, ASC("a") gives decimal 97.

CHR$(arg)

returns a single byte string containing the ASCII character corresponding to the value given by arg. Examples: CHR$(:41) gives "A", CHR$(7) gives <BELL>.

FIND(arg1$,arg2$)

searches for an occurrence of arg2$ in arg1$. Arg1$ and arg2$ may be string functions, string constant, string names, or substrings. Returns Ø if not found; otherwise returns the smallest I such that arg1$[I,LEN(arg2)]=arg2.

DATE$

returns a string corresponding to the current date (string content is defined by the operating system). Note that BASIC OPENs the CLOCK: device (on the first available I/O channel) to get the date, so an I/O channel must be available when this function is invoked; the I/Ø channel is CLOSEd immediately after use.

TIME$

returns a string corresponding to the current time of day (string content is defined by the operating system). Like DATE$, an I/O channel must be available when TIME$ is invoked.

VAL(stringexpression)

returns a numeric value equal to string expression contents interpreted as a floating point or a hex constant. Leading blanks or tabs are ignored. Any illegal characters delimit the value to be converted. The number in the string must be valid or a conversion error results.

HEX$(arg)

returns a string containing a 4 digit hexadecimal constant equivalent to the value. The first character of the string is a ":" so that the length of the result is 5 characters. An argument which is not an integer in the range Ø to 65535 will cause an error.

NUM$(arg)

returns a string whose contents are exactly what a

```
PRINT ARG;
```

statement would have printed for the argument expression, except no trailing "space" is produced.

NUMF$(stringexpression,arg2)

returns a string whose contents is what a

```
PRINT USING stringexpression,arg2;
```

statement would have printed. The stringexpression must contain only a valid "number format" (not a general format string) as defined by the section on PRINT USING; otherwise, an error results.

TRUE returns the value 1.

FALSE returns the value Ø.

IF condition THEN arg1 ELSE arg2 FI

This is known as the "IF" function. Returns the value of arg1 if the conditional expression is true, else it returns the value arg2. Arg1 is NOT evaluated if condition is false; arg2 is NOT evaluated if condition is true. For readability, a <CR> is optionally allowed after the condition or arg1, so the IF function invocation may span several physical text lines. A <CR> is NOT allowed after arg2. The final FI is required.

Examples:

```
1Ø LET A=2+IF B<>Ø THEN COS(X)/B ELSE COS(X) FI
REM NO DIV BY Ø ERROR POSSIBLE

2Ø REM MOVE CURSOR RIGHT
   LET CURSORCOL= IF CURSORCOL=SCREENWIDTH
                  THEN SCREENWIDTH
                  ELSE CURSORCOL+1 FI
```

The type of arg1 must match the type of arg2, and the types must be compatible with the expression in which the IF function is embedded.

DATA STRUCTURES

This section describes how data is stored in BASIC from an assembly language interface point of view.

Simple numeric variables use 6 bytes of storage. They may contain either a floating point number or an integer (at any instant). All scalar variables that are not parameters or COMMONed are stored sequentially in an area known as the Scalar Space.

Vectors occupy 6*(I+1)+2 bytes, where I is the dimension size (the +1 allows room for the zero subscript). The first two bytes contain the DIMensioned length of the vector, and are used to perform subscript checking. The zeroth element of the vector follows immediately. Each vector element occupies 6 bytes, and may contain either a floating point number or an integer.

Arrays occupy 6*(I+1)*(J+1)+4 bytes, where I and J are the dimension size (the +1 allows for a zero row or column subscript). The first pair of bytes contain the DIMensioned number of rows of the array, and are used for subscript bound checking. The second pair of bytes contain the DIMensioned number of columns, also used in subscript checking and array entry address computation. Each array entry uses 6 bytes, and is located by adding (I*2nd dimension+J)*6+4 to the array address, where I and J are the row and column subscripts, respectively. Each array element occupies 6 bytes and may contain an integer or a floating point number.

String variables occupy the dimension +4 number of bytes. The first two bytes are the MAXLEN (dimension) of the string. The next two bytes are the current length of the string, and are always less than or equal to the max length. The rest of the bytes hold the string, left justified.

String arrays occupy (LENgth dimension)*(MAXLEN dimension+4)+2 bytes. The first two bytes hold the number of strings in the string array. Each string in the string array has the same structure as a simple string variable.

String constants are stored with the first byte as the length; the remaining bytes are the body of the string constant.

SIMPLE NUMERIC VARIABLE

```
        !----------------!
        !     VALUE      !       6 BYTES
        !----------------!
```

VECTOR

```
        !----------------!
        !     LENGTH     !       2 BYTES
        !----------------!
        !                !
        !     VECTOR     !          6*
        !    ELEMENTS    !       (1+VECTOR
        !                !       DIMENSION)
        !                !
        !                !
        !----------------!
```

ARRAY

```
        !----------------!
        !     # ROWS     !       2 BYTES
        !----------------!
        !   # COLUMNS    !       2 BYTES
        !----------------!
        !                !
        !                !          6*
        !     ARRAY      !        (# ROWS+1) *
        !    ELEMENTS    !       (# COLUMNS+1)
        !                !
        !                !
        !----------------!
```

The number of rows and the number of columns are taken from the DIM statement for the array (i.e., DIM ARRAY[rows,columns]).

STRING ARRAY

```
!--------------------------!
!          LENGTH          !      2 BYTES
!--------------------------!
!    STRINGVARIABLE(1)     !      MAXLEN+4 BYTES
!--------------------------!
!    STRINGVARIABLE(2)     !      MAXLEN+4 BYTES
!--------------------------!
!             .            !
!             .            !
!             .            !
!--------------------------!
! STRINGVARIABLE(LENGTH)   !      MAXLEN+4 BYTES
!--------------------------!
```

STRING VARIABLE

```
  1B     1B     1B     1B     1B     1B     1B           1B
       !-----!-----!-----!-----!-----!-----!-----!       !--------!
       !    MAX     !    CUR     ! 1ST ! 2ND ! 3RD ! ... ! MAXTH  !
       !-----!-----!-----!-----!-----!-----!-----!       !--------!
           <65535       <=MAX
```

STRING DESCRIPTOR

```
!-----!-----!-----!-----!-----!-----!
!  1  !  X  !  ADDRESS  !   COUNT   !
!-----!-----!-----!-----!-----!-----!
```

String descriptors are used to temporarily represent a string or a substring in expressions; they are found in argument lists to assembly language routines.

COUNT has the range Ø<= COUNT <= 65535. If COUNT = Ø (empty string), the ADDRESS is meaningless.

If COUNT <> 65535 (substring), then ADDRESS+4 points to the first selected byte. COUNT specifies how many bytes are selected.

If COUNT = 65535 ("the entire string"), then ADDRESS points to the left byte of the max length of some string variable. The number of bytes selected is equal to MAX or CUR, depending on the operation.

DECIMAL FLOATING POINT VALUES

```
!-----!-----!-----!-----!-----!-----!
! EXP ! DIG ! DIG ! DIG ! DIG ! DIG !
!-----!-----!-----!-----!-----!-----!
```

DIG are base 1ØØ (not BCD) digits, i.e., values in the range Ø to 99 decimal. The leftmost DIG is non-zero. The most significant bit of the EXP is the number sign. The other 7 bits are the base 1ØØ exponent, biased by +64. Floating zero is defined as EXP = Ø, all DIG bytes = Ø. Otherwise, an EXP of Ø is illegal. See FLOATING POINT PACKAGE.

INTEGER VALUES

```
!-----!-----!-----!-----!-----!-----!
!  Ø  !  X  !  X  !  X  !  INTEGER  !
!-----!-----!-----!-----!-----!-----!
```

Ø <= INTEGER <= +65535

ADDRESS OF NUMERIC VALUE

```
!-----!-----!-----!-----!-----!-----!
!  Ø  !  X  !  X  !  X  !  ADDRESS  !
!-----!-----!-----!-----!-----!-----!
```

The ADDRESS points to the EXP byte.

ARGUMENT LIST FORMAT (ON "CALL" TO ASSEMBLY SUBROUTINE)

```
!----------------!
! LAST ARGUMENT  !  <----INDEX REGISTER (X)
!----------------!
!                !
!       .        !
!       .        !
!       .        !
!                !
!----------------!
! FIRST ARGUMENT !  HIGHER ADDRESS THAN (X)
!----------------!
```

On entry to the assembly language subroutine or function, register A indicates how many arguments were passed. Each argument occupies 6 bytes, with the last argument being at the lowest memory address. Register X points to the lowest address byte of the last argument. Each entry in the argument list is a string descriptor, or an address in the format described in this section.

SYSCALL PARAMETER LIST FORMAT:

```
 _____________
!    OPCODE    !   <----INDEX REGISTER (X)
!--------------!
!    WRBUF     !
!--------------!
!    WRLEN     !
!--------------!
!    RPLEN     !
!--------------!
!    RDBUF     !
!--------------!
!    RDLEN     !
!--------------!
!  EXTENSION   !
!______________!
```

After JSR $FB, (X) points to the SYSCALL parameter list.
For meaning of various opcodes, see SDOS manual.

FLOATING POINT PACKAGE

The SD floating point package for BASIC V1.4 is a fast, high precision (decimal arithmetic) software floating point package for 6800/6809 CPUs. It provides a stack-oriented environment to allow convenient evaluation of complicated expressions (in Polish notation). The package provides ADD, SUB, MUL, DIV, stack load and store, negate, floating point comparison, integer truncation, fix and float, and conversion routines to and from ASCII and floating point. Trancendentals are not included.

The floating point number format requires 6 bytes. The most significant bit of the first byte contains a sign S and the exponent (least significant 7 bits). S=0 means positive sign; S=1 means negative sign. The exponent is base 100, biased by hex 40. The range of the exponent is:

$$100^{\$40+\$3F} \quad \text{to} \quad 100^{\$40-\$3F}$$

which is

$$100^{63} \quad \text{to} \quad 100^{-63}$$

or

$$10^{126} \quad \text{to} \quad 10^{-126}$$

An exponent of zero is defined to represent a floating zero. Negative zero is not allowed. Clean floating zero is represented by 6 zero bytes.

The remaining five bytes are mantissa digits, in base 100, i.e., digits have values of 0-99. The byte contents are stored as the binary equivalent of the digit. A normalized floating point number is defined to be one in which the leading mantissa byte is non-zero. This means that up to 10 BCD digits can be stored. However, due to normalization conditions, the left-most base 100 digit may be as small as 1 which means only 9 decimal digits of precision can be guaranteed. If you are working with money amounts, note that up to $100 million can be represented accurately, to the penny.

Floating Point Number Format:

```
-------------------------------------------------------------------
! S ! 7-BIT EXP ! BASE1ØØ ! BASE1ØØ ! BASE1ØØ ! BASE1ØØ ! BASE1ØØ !
-------------------------------------------------------------------
```

Examples:

```
Ø       ØØ ØØ ØØ ØØ ØØ ØØ
1       41 Ø1 ØØ ØØ ØØ ØØ       -1              C1 Ø1 ØØ ØØ ØØ ØØ
PI      41 Ø3 ØE ØF 5C 41       -PI             C1 Ø3 ØE ØF 5C 41
.5      4Ø 32 ØØ ØØ ØØ ØØ       -.5             CØ 32 ØØ ØØ ØØ ØØ
1.5     41 Ø1 32 ØØ ØØ ØØ       -1.5            C1 Ø1 32 ØØ ØØ ØØ
1.5E-1  4Ø ØF ØØ ØØ ØØ ØØ       -1.5E-1         CØ ØF ØØ ØØ ØØ ØØ
1.5E-2  4Ø Ø1 32 ØØ ØØ ØØ       -1.5E-2         CØ Ø1 32 ØØ ØØ ØØ
1.5E-3  3F ØF ØØ ØØ ØØ ØØ       -1.5E-3         BF ØF ØØ ØØ ØØ ØØ
```

The following routines are contained in the floating point package:

```
FLOAD   Floating load
FSTORE  Floating store
FCMP    Algebraic compare two floating point numbers
FNEG    Negate floating point number
FADD    Add two floating point numbers
FSUB    Subtract two floating point numbers
FMUL    Multiply two floating point numbers
FDIV    Divide two floating point numbers
FIX16   Convert floating to 32 bit binary integer in range Ø..65
FLOAT   Convert binary integer to floating
FINT    Truncate fractional part
FIX     Convert floating to 32 bit binary integer
FCONVO  Output conversion from floating point format
FCONVI  Input conversion to floating point format
```

These routines are stack oriented; that is, they operate on numbers already in the stack, put numbers onto the stack, or take numbers off the stack.

Throughout this description, the term 'TOS' and 'TOS-1' shall mean 'the floating point value on the top of the stack' and 'the floating point value underneath the floating point value on the top of the stack', respectively.

FPTRAP is a 2 byte page zero location containing a 'floating point trap address'. Should overflow occur in the floating add, subtract, multiply or divide routines, control will be transferred to the location specified by FPTRAP. Since these routines are commonly used, setting up the error address once saves bytes, and makes the user code more readable.

A description of each routine follows. A JSR is used to enter all routines. Locations Ø-7 are used by these routines as scratch storage.

FLOAD

On entry, the X register contains a pointer to a six-byte floating point number (specifically, X points to the exponent byte of the number). This routine loads the 6 bytes onto the stack and then returns to the JSR+3.

FSTORE

On entry, the X register contains a pointer to a location to store the six-byte floating point number from the TOS. The routine pops the number off the stack and stores it into the location specified, then returns to the JSR+3.

FIX

FIX attempts to fix the floating point number on the TOS into a 32-bit (4 bytes) signed binary integer on the TOS. The number is considered unfixable and left intact on the TOS if it is less than -2^31 or greater than (2^31)-1. Returns to JSR+3 if fixable, JSR+5 if not.

FIX16

Operates identically to FIX, except the result must be in range of Ø..65535. Returns to JSR+3 if fixed properly, to JSR+5 if not.

FLOAT

FLOAT will convert a 32-bit signed integer (4 bytes) on the TOS into a 6-byte floating point number on the TOS. Returns to JSR+3.

FINT

FINT will truncate fraction bits, if any, in the floating point number on the TOS and return the largest integer not larger than the value on the TOS. For example:

```
INT(1.Ø) = 1.Ø
INT(1.2) = 1.Ø
INT(-2.Ø) = -2.Ø
INT(-1.2) = -2.Ø
```

Returns to JSR+3.

FCONVO

On entry, the TOS contains the floating point number to convert, and the X register points to the output buffer. The output buffer must be large enough to contain 17 bytes. The routine converts the number into an ASCII string in 'E-type' format:

```
(SIGN)('.')(1Ø DIGITS)('E')(ESIGN)(3 DIGITS)
```

On exit, the A register contains a number representing the number of places the decimal point in the resulting string would have to be shifted right to make the exponent zero. Example:

```
A =  Ø --> '.DDDDDDDDDD'
A = -2 --> '.ØØDDDDDDDDDD'
A =  2 --> 'DD.DDDDDDDD'
```

The B register contains the number of significant digits (1Ø - rightmost zero digits). Returns to JSR+3.

FCONVI

Converts an ASCII numeric string to the internal floating point format. On entry, register X points to the string to convert and register D (A,B) contains the string length. The result of the conversion is pushed on TOS, and (X) is returned pointing to the data byte that terminated the conversion. Leading blanks and nonsignificant zeros are automatically skipped.

Digits beyond those retainable by the conversion are skipped and ignored. The E part of the exponent may be upper or lower case, and is accepted only if followed by a valid exponent value (i.e., only 123 is accepted from a string like 123E%).

Returns to JSR+3 if conversion succeeded. Returns to JSR+5 if overflow occurs; TOS contains a properly signed version of infinity. Returns to JSR+7 if syntax error; no value is pushed onto TOS.

FNEG

FNEG negates the floating point number on the TOS. Returns to JSR+3.

FCMP

FCMP compares the floating point number on the TOS-1 to the number on the TOS. The condition codes are set according to the result of the algebraic difference of TOS-1 - TOS (S, N condition code bits are set to show the result of the compare, V is set to Ø). Both numbers are popped off the stack. This routine should be used instead of FSUB for comparisons because it pops both numbers off the stack and FSUB doesn't; FCMP sets the condition codes and FSUB doesn't, and FCMP is much faster than FSUB. Returns to JSR+3.

FADD

On entry, FPTRAP contains the error exit address. The TOS contains the addend, and the TOS-1 contains the augend. The two numbers are popped off the stack, added, and the sum is pushed onto the stack. If underflow is detected, a floating zero is returned. If overflow is detected, negative infinity (FF 63 63 63 63 63) is returned if the result sign is negative, and positive infinity (7F 63 63 63 63 63) if the sign is positive. Returns to JSR+3 if underflow or no error. Returns to the address specified in FPTRAP if overflow occurred.

FSUB

On entry, FPTRAP contains the error exit address. The TOS contains the subtrahend, and the TOS-1 contains the minuend. The two numbers are popped off the stack, subtracted, and the difference is pushed onto the stack. Overflow/underflow and exit conditions are the same as FADD.

FMUL

On entry, FPTRAP contains the error exit address. The TOS contains the multiplier, and the TOS-1 contains the multiplicand. The two numbers are popped off the stack, multiplied, and the product is pushed onto the stack. Overflow/underflow and exit conditions are the same as FADD.

FDIV

On entry, FPTRAP contains the error exit address. The TOS contains the divisor, and the TOS-1 contains the dividend. The two numbers are popped off the stack, divided, and the quotient is pushed onto the stack. Overflow/underflow and exit conditions are the same as FADD.

In Version 1.4 of the SD BASIC Runtime Package (RTP), the floating point routines can be easily accessed via a transfer vector:

```
FPTRAP  EQU     $ØØ28

FLOAD   EQU     $Ø1Ø9
FSTORE  EQU     $Ø1ØC
FCMP    EQU     $Ø1ØF
FNEG    EQU     $Ø112
FADD    EQU     $Ø115
FSUB    EQU     $Ø118
FMUL    EQU     $Ø11B
FDIV    EQU     $Ø11E
FCONVO  EQU     $Ø121
FCONVI  EQU     $Ø124
FINT    EQU     $Ø127
FIX     EQU     $Ø12A
FIX16   EQU     $Ø12D
FLOAT   EQU     $Ø13Ø
```

Note that the floating point routines use page zero heavily; other than the scratchpad locations (see SDOS manual), we recommend not using page zero in the application program.

If special hardware is considered, we recommend replacing FMUL, FDIV, FIX, and FLOAT. FADD, FSUB, FLOAD, FCMP, FSTORE, and FNEG are fast enough that replacing them with hardware won't really save any time.

EXAMPLE OF USE

Assume the following sequence needed to be coded:

```
LET RESULT = (INPUTVALUE * 1.5 + .1Ø1) / PI
IF RESULT >= 55.7 THEN GOTO LABEL1
```

Assume RESULT is a floating point variable, and INPUTVALUE is an 8-bit unsigned integer. Then the following code could be used:

```
*Floating Point Assembly Example

START       LDS     #STACK
            LDX     #TRAP
            STX     FPTRAP
*
            LDAA    INPUTVALUE CREATE A 4-BYTE INTEGER ON
            PSHA               THE STACK
            CLRA
            PSHA
            PSHA
            PSHA
            JSR     FLOAT      NOW FLOAT IT
            LDX     #F1.5      GET A 1.5
            JSR     FLOAD
            JSR     FMUL       NOW DO THE MULTIPLY
            LDX     #F.1Ø1     GET THE .1Ø1
            JSR     FLOAD
            JSR     FADD       NOW DO THE ADD
            LDX     #PI        GET PI
            JSR     FLOAD
            JSR     FDIV       NOW DO THE DIVIDE
            TSX                DUPLICATE THE TOS FOR THE COMPARE
            JSR     FLOAD
            LDX     #RESULT    SAVE A COPY IN RESULT
            JSR     FSTORE
            LDX     #F55.7     GET 55.7
            JSR     FLOAD
            JSR     FCMP       DO THE COMPARE
            BGE     LABEL1
            .
            .
LABEL1      .
            .
            .
TRAP       ;GET HERE IF OVERFLOW ERROR IN FADD, FSUB, FMUL, FDIV
            .
F1.5        FCB     $41,Ø1,5Ø,ØØ,ØØ,ØØ
F.1Ø1       FCB     $4Ø,1Ø,1Ø,ØØ,ØØ,ØØ
PI          FCB     $41,Ø3,14,15,92,65
F55.7       FCB     $41,55,7Ø,ØØ,ØØ,ØØ
INPUTVALUE                     RMB      1
RESULT      RMB     6
            RMB     5Ø
STACK       EQU     *-1
            END
```

The following subroutine will convert a string containing BASIC V1.3 floating point numbers to BASIC V1.4 floating point numbers:

```
SUBROUTINE CONVERT13TO14(NUMBER$)
   IF NUMBER$[I]=Ø THEN RETURN SUBROUTINE
   FOR I=2 TO 6
   LET NUMBER$[I]=((NUMBER$[I]**-4)*1Ø)+(NUMBER$[I]&:ØF)
   NEXT I
   RETURN SUBROUTINE
END
```

To use the subroutine, a record containing BASIC V1.3 numbers should be READ using strings instead of scalars as targets of the numeric READs for numbers. Each number read into a string should be converted by invoking the CONVERT13TO14 subroutine, then the record should be written back.

Example:

```
DIM EMPLOYEENAME$(...), SALARY$/Ø,Ø,Ø,Ø,Ø,Ø/
...
REM This record is usually read as
REM READ #FILE@RECORDLOCATION, EMPLOYEENAME$, SALARY
READ #FILE@RECORDLOCATION, EMPLOYEENAME$, SALARY$
CONVERT13TO14(SALARY$)
WRITE #FILE@RECORDLOCATION, EMPLOYEENAME$, SALARY$
```


USING SOFTWARE DYNAMICS BASIC

Since the Software Dynamics BASIC is a compiler, the procedures for using it are different from a conventional interpreter. This section includes directions on how to use the compiler to prepare a BASIC program for execution, and how to load and execute a BASIC program.

The SD BASIC system consists of three parts:

1) The compiler
2) The runtime package
3) The utility programs FIX, COMPILE, and FINDLABEL

The compiler accepts a BASIC source program and converts it to a form compatible with the assembler. This intermediate form is assembled (along with any user assembly language subroutines) to produce a binary program. Finally, the binary program is loaded with the runtime package for execution.

The first step is to create the desired BASIC program. This is done with the aid of a text editor program. The program is prepared as described in the section on program organization. Line numbers/labels are used to mark targets of GOTOs, and handy reference points within the program. SD BASIC does not require line numbers on all lines; this fact can be used to clarify the program (by removing some of the clutter of conventional BASIC programs) and allow compilation of larger programs (line numbers use up space at compile and execution time). It is a good idea to number each line until the program is nearly debugged, to aid in error diagnosis at runtime. Note that SD BASIC does not care about the order of line numbers; the text order is what counts for sequential execution. Using line number order is convenient and aids program compatibility with other BASICs.

The source form of the BASIC program consists of lines terminated with a <CR> (hexadecimal $ØD) character. Multiple spaces are treated as a single space (except in quoted character strings). Spaces may not occur in the middle of keywords, variable names, subroutine names, or in the middle of 2 character operators such as "**" or ">=". Otherwise, spaces may be used freely to improve readability. Extraneous spaces in the source program do not affect execution times.

The compiler ignores nulls; tabs are treated as spaces except in character strings. Control L and line feed are legal only after a <CR> mark. The last line of the program must be an END statement.

Upper or lower case may be used freely by the programmer; all lower case text not in a quoted string is treated as if it were upper case.

If a statement is too long for a source line, the statement may be continued on the next source line by writing ...<CR>& wherever blanks would be allowed, followed by the rest of the statement (note: This line continuation facility does not work in character strings or REM statements!).

Example:

```
          1Ø IF ARRAY(BUFFER(J),1Ø) >= ...
&         SUM THEN GOTO 75 ELSE PRINT "OOPS!"\STOP
```

If you must continue a line following a number, separate the continuation periods from the number by at least one space, or BASIC will think the first period is a decimal point and complain about the following '..'.

PASS I

Once the program is created, the next step is compilation. The compiler is very easy to use. Merely load the compiler by typing BASIC when at the SDOS command interpreter prompt. It will identify itself, and then ask for the source and output file names.

Example:

```
.BASIC
Software Dynamics BASIC Compiler V1.4h (C) 198Ø
INPUT FILE = MYPROG.BAS
OUTPUT FILE = JUNK.TMP
```

The compiler opens the source file, and creates the output file. The size of the output file that will be produced is typically three times that of the source file.

Error diagnostics are written to the console (SDOS channel #Ø). They consist of a message describing the kind of error, a printout of the line in which the error occurs, and a pointer (caret) to the problem. Each error diagnostic occupies three printed lines, and is separated from the next by a blank line to make the grouping obvious. Typical error examples are:

```
Syntax Error
  1Ø     DIM S$(12),B,7
                      ^

  Variable not DIMed as vector or array
  2Ø LET Q(I,J)=2
         ^
```

The compiler will always print "Compilation Complete" when done. This does NOT mean no errors were found.

COMPILE TIME ERROR MESSAGES

Already defined as a variable
: This name already has a definition, and so cannot be used as a label.

Assignment to a label is not allowed
: The target of a LET, READ or INPUT is the name of a label.

Can't use a parameter as a FOR loop index
: Only scalar variables that are not parameters are allowed as FOR loop indexes.

Can't use variable name for label
: A GOTO target contains the name of an object which is not a label.

Compiler Bug! XXXX
: The memory of the computer is unreliable, SDOS is unreliable, or the BASIC Compiler has an error. Please, REPORT THE ERROR TO SOFTWARE DYNAMICS, along with the source of the program, and exactly what was displayed.

Doubly-defined line number
: This line number has already been used.

Doubly-defined string variable
: There is already a definition for a string of this name.

Double subscript required
: This variable is defined as an array, and must have a double subscript in this context.

End of File hit
: The compiler expects more BASIC program, but there isn't any more. Suspect a missing NEXT, END, or FI; unfortunately the place that is missing the keyword might be almost anywhere in your program.

Function not allowed here
: This name is defined as a function name, and cannot be used in this context.

HOW MUCH STRING SHOULD I READ?
: A syntactically legal READ into a string function has been encountered, but there is no well defined meaning for it.

Memory Full: xxxx
: The program being compiled requires too much space to compile in this configuration. (xxxx indicates which compiler routine detected the error).

Missing Block END
There is a block body (REPEAT, WHILE, DO, FOR, etc.) that is incomplete and an END statement is required to complete it. (Very likely an END statement was left out several lines prior to this point.)

Missing FI
A THEN or ELSE block is incomplete and requires a FI to "seal" it. (It is fairly likely that a FI or ELSEIF is missing several lines prior to this point).

Must be loop label
An EXIT label statement specifies a label which is not the label of a block-type statement (FOR, REPEAT, DO, etc.).

No enclosing FOR with same variable
This NEXT statement specifies an index variable that is not matched by any textually preceding FOR statement.

No subscripted variables allowed here
A scalar variable name must be used here (a vector variable may not be used as a FOR-NEXT index variable).

No such file
An INCLUDE file does not exist.

Not implemented
What you are requesting appears to be legal, but the compiler cannot generate code for it.

Parameter variable not allowed here
A variable declared in a parameter list of a SUBROUTINE or DEF statement is used in an illegal context.

Single subscript required
This variable is defined as a vector, and must have only a single subscript in this context.

Source line is too long
More than 256 characters have been scanned without encountering a carriage return character. The source line will have to be split up.

String array requires subscript here
A string array requires a subscript to select the particular string desired.

String length exceeds 127 characters

A name, number or string constant has more than 127 total characters in it. Names or numbers with excessively many digits must be shortened. A string which is too long will have to be split into two or smaller strings; the line continuation technique ("...<CR>&") or multiple statements may be needed to help solve the problem.

Syntax error

The line printed is not syntactically correct. Note: since NEXT, END, THEN, ELSE and ELSEIF are not legal BASIC statements (i.e., they can only occur as part of another statement), a syntax error on these "statements" indicates that the preceding part of the statement is missing or incorrect. Example: a NEXT will yield a syntax error if there is no corresponding FOR statement.

Undefined string variable

This string variable is not defined in a DIM or COMMON statement, or in a parameter list.

Use of name incompatible with previous use

A textually preceding context defined this name in a way that is not compatible with use in this context.

Variable not DIM'ed as vector or array

The variable specified cannot be subscripted because it is a scalar.

Wrong type of value

A string result appears where a numeric expression is required, or vice versa, or an array or vector object is found where a scalar is appropriate, or vice versa.

HINTS ON HOW TO HANDLE COMPILE-TIME ERRORS

The compiler processes the entire program (unless it runs into too many ENDs), even if an error occurs. If any error message is printed, the compiler output is turned off and consequently is useless. If errors are diagnosed by the compiler, you must go back and edit the program to rid it of those errors, and recompile. Note that an error may cause other (spurious) errors; for instance, terminating a FOR block with a FI will cause the compiler to lose track of blocks that contain the FOR, and consequently complain about NEXTs and FIs.

If the compiler complains about a statement that appears legal, check the declarations of all variables referenced in that statement to make sure they have the proper type.

If an error produced by the compiler seems particularly incomprehensible, and the compiler reported another error at an earlier point in the same file, try fixing the earlier error and recompiling; many times, this will remove the source of an "error cascade", and thus the "incomprehensible" error.

Too many ENDs are not detected by the compiler; it simply stops compiling when it sees an END which does not match any unclosed block.

The compiler gives error 1ØØ to SDOS if any errors were diagnosed. This may be used in DO file to automate fetching of the editor.

PASS II

When an error free compilation occurs, then you are ready for pass II. This consists of assembling the compiler output using the Software Dynamics Assembler.

If the compiler issues any messages (other than Compilation Complete), performing an assembly is useless and any errors resulting are meaningless.

The output of the compiler contains END and ORG statements. These statements are in textual form in the output and are assembler directives. They are related to, but not the same as, the BASIC statements END, DATA ORIGIN, and PROGRAM ORIGIN. In this section, the words "END" and "ORG" refer only to the assembler directive statements END and ORG. If there are some assembly language subroutines, they should be included just prior to the END statement in the compiler's output. This can be done with an Editor, or by use of an INCLUDE command to the SD Assembler.

The ORG statement produced by the compiler defaults the BASIC program to location $2EØØ for 68ØØ, and $2AØØ for the 68Ø9. This can be changed (via PROGRAM ORIGIN or DATA ORIGIN statements) to anywhere desired in the computer provided you leave room at the bottom of memory for the runtime package (11K). The first instruction in a BASIC program is a "JSR $1ØØ"; if the program is assembled somewhere other than the standard location, it must be started by transferring control to this JSR.

Assembly time errors can occur in the form of "Undefined Symbol". The form of the undefined symbol is the key to the problem.

A symbol of the form :dddd where d's are digits means a GOTO (GOSUB, ON, etc.) target line number is not defined in the program.

A symbol of the form E:xxxx means an EXIT label statement has been used, but label xxxx is not the label of a block-type statement.

A symbol of the form xxxx (alphanumeric) means a GOTO (label) xxxx is used, but label xxxx is not defined in the program, or that a scalar variable xxxx is referenced, but has never had its value set (via READ, INPUT, or LET).

An error of the form ?Data space overlaps Program Space? Or ?Program Space Overlaps Data Space? Indicates that DATA or PROGRAM origin statements have been used, but not enough space for the data area was taken into account.

An error of the form ?Too Many Scalar Variables? Indicates that a BASIC program has used over 32Ø scalar variables. The number of scalars will have to be reduced, or the program will have to be broken into several parts with separately compiled functions or subroutines.

Other assembly time errors occur only if the compiler output file is damaged, the computer hardware is failing, or an included assembly subroutine has errors.

PROGRAM EXECUTION

A successful assembly means you are ready to run the program. Simply load both the runtime package object and the program object (from the assembler) and start it at the PROGRAM ORIGIN (if you had no PROGRAM ORIGIN statement, start it at the assembly default location).

On most SDOS systems, the runtime package and the SDOS command interpreter are placed in DEFAULTPROGRAM; in this case, all that is necessary to load and execute the compiled program is to type its name (the assembler will have set the start address to the proper value already). Otherwise, execution of a BASIC program under SDOS is accomplished by typing

```
RUN PROGRAMNAME
```

where RUN is the name of the runtime package. If your program is too big, SDOS will complain when it is loaded.

Runtime errors are printed out as

```
Line xxxxx
Text of error message
```

where xxxxx is the last line (number) encountered before the executing the line in which the error occurred. If line 5Ø invokes a user defined function which errors, the error displayed will show line 5Ø errored because of error propagation. A table of error codes detected by BASIC is given in the section on Runtime Error messages. Errors detected by SDOS can be found in the SDOS manual.

Line numbers are printed as

    Line ddddd

if a numeric line number was used. If a line label was used, the line "number" is printed as

    Line :xxxx

where xxxx is the address of the line label in the computer's memory. The label name can be determined by examining the symbol table dump produced by the assembler. A program to automate inspecting the symbol table is easily written (See FINDLABEL below).

The program can be stopped by the operator by simply pressing the "ESC" key; the message "Operator Requested Attention" will occur the next time the program executes a line number, label, or subroutine call. Note: if error trapping is enabled, an ESC will only activate the error handling part of the BASIC program.

When the program asks for input, remember to separate all typed-in numbers by commas, spaces, or tabs, and to push the <CR> key when done with the input line.

FIX, COMPILE and FINDLABEL

COMPILE is a SDOS utility program that automates the compilation and assembly steps, by creating an SDOS DO file and "DO"ing it. To use COMPILE, type COMPILE <filename><CR> or COMPILE <filename.bas><CR>. COMPILE will do the rest.

FIX is a utility program that lets one edit a BASIC program, and then automatically invokes COMPILE. Typing FIX <filename><CR> or FIX <filename.bas><CR> start FIX. FIX will invoke SEDIT if SEDIT is present on the default device, otherwise it will invoke EDIT. After exiting the editor, COMPILE will be started automatically.

If COMPILE detects an error (either in the compilation or assembly step), it will automatically invoke FIX, so that the operator may re-edit and try the compilation again (it is convenient to type ^P to SDOS after typing COMPILE to ensure that any displayed errors do not roll off a CRT screen before they are noted by the programmer).

COMPILE will also help when constructing a BASIC program that calls an assembly language module. COMPILE <filename> WITH <filename2><CR> will cause "filename" to be compiled; filename2 (which should contain the source of the assembly language routines) is appended to the compiler output (before the END statement) before the assembler is called. FIX <filename> WITH <filename2> allows filename to be edited before COMPILE ... WITH ... is invoked.

FINDLABEL is a utility program to hunt through a symbol table file (generated by the assembler from a BASIC compilation) for the name of a label printed by the runtime package as :xxxx. The program is invoked by typing FINDLABEL xxxx<CR>. It will look through ASMLOG.TMP if present; otherwise, it will ask for the name of the symbol table file. All labels with the value xxxx are shown. If no labels are shown, suspect that xxxx was mistyped, or the symbol table file used did not match the program in error.

PROGRAM DEBUGGING AIDS

SD BASIC provides some program debugging aids. The programmer can activate line number trace, single line step, or set a line number breakpoint (currently, no facility exists to examine variable contents, other than PRINT statements coded into the BASIC program itself). The SDOS I/O package defines the mechanism used to request these operations; normally, ^T (control T) is used to toggle trace mode, ^V to cause single step, ^B to request a breakpoint, and ^G to continue at full speed.

Typing ^B (at any time) on the operator's console, causes the runtime package to print:

```
Break on Line?
```

The operator enters a line number ddddd and pushes <CR>. An illegal type-in causes re-prompt. To set a breakpoint on a label, the operator must enter the address of the label as :xxxx. This address can be obtained from the symbol table dump generated by the Assembler. A type-in of zero removes a previously set breakpoint; any other value replaces the previous breakpoint line number. The runtime package automatically continues execution at full speed, until the specified line number is reached, and then prints:

```
Line ddddd (or Line :xxxx)
```

where ddddd (or :xxxx) is the specified line number. The runtime package now waits for the operator to request a new breakpoint, single step, trace, or go. Breakpointing can also be performed on subroutine and function entry points. Only one breakpoint is allowed at any time.

Typing ^V (at any time) causes the runtime package to enter single step mode. It executes the current line, and prints out the next line's number in the form:

```
Line ddddd (or Line :xxxx)
```

and waits for the operator to request a breakpoint, another single step, trace, or go.

Typing ^T (at any time) will cause the runtime package to enter line number trace mode, if not already in trace mode; otherwise, ^T causes the runtime package to exit trace mode. In trace mode, the runtime package prints out the line number of each line just before executing that line; then it will continue automatically. Calls to functions and subroutines are traced. Note that lines without linenumbers or labels cannot be traced.

Typing ^G is only valid in response to a single step or breakpoint display. This causes the runtime package to continue execution at full speed; any breakpoint which has been set is still active.

DEBUGGING A COMPILED BASIC PROGRAM

This section contains several tips on how to debug a compiled program.

Tracing and breakpointing are really very useful features; they should be used. Extra line numbers/labels should be placed in a program, especially at critical or complex calculations, so that the line number traced to or reported before an error more specifically pinpoints the problem.

Installing extra print statements in a program is also useful. A convenient way to do this is to write

```
IF DEBUGGING THEN PRINT debugginginformation
```

and to have a command, that when given to the program, sets DEBUGGING=TRUE. The debugging statements can be REM'ed out for a production compile if space is critical.

If a floating point number is printed out which contains any of the following characters:

: ; < = > ?

where one would expect digits, the program probably printed the value of a variable which was never assigned a value.

Many string subscripting errors are caused by confusion over the current LENgth of a string, and the MAXLENgth of a string. Any time a substring is specified, the string being subscripted must have a current LENgth large enough to accommodate the string bounds. MAXLEN only places a ceiling on the largest value of current LENgth; it has nothing to do with subscript checking. A string's current LENgth is set only by assignment (LET, READ or INPUT) into the name of the string, or by a LET LEN(stringname$)= statement. Assignment to a substring does NOT change the current LENgth of the string. The compiler initially zeros the current LENgth of DIM'd strings in order to make failure to set the string length more apparent.

An "error" of the form

```
Line xxxxx
```

With no error message is really only a STOP statement terminating the program's execution. If xxxxx is a hexadecimal number, it is possible that control has reached the END of a SUBROUTINE or function without executing a RETURN SUBROUTINE or RETURN <expression> statement.

It is very unusual for a compiled BASIC program to cause the computer to "crash". One of the possibilities is Stack Overflow.

It is possible for a BASIC program to overflow the computer's pushdown-stack by performing too many GOSUBs, subroutine or function calls, or by having a statement of enormous complexity. Since this problem is extremely rare, and stack limit checks are extremely expensive (in terms of overhead), no stack limit checks are performed. A program with this problem will generally just "crash" the computer, or act absolutely insane. The only cure is to add more memory to the CPU or to move the BASIC program lower in memory.

Another obscure possibility is that the program performs an assignment to a parameter variable, outside of the subroutine or function body that declared that variable. This reference is absolutely illegal! (but the compiler cannot detect it). Because parameters are call-by-reference, such an assignment causes BASIC to store the assigned value at wherever the parameter variable currently happens to point, which might be garbage. Messages such as "SDOS self-test checksum error" or "RTP self-test checksum error" are also indications of this problem.

A third possibility is execution of RETURN SUBROUTINE in a function or RETURN <expression> in a SUBROUTINE.

Passing a string as a argument to a subroutine or function which expects a numeric argument (in general, passing the wrong type of argument) can also cause some very strange effects.

SEPARATE COMPILATION

This revision of SD BASIC allows a program to be textually broken into several parts. The pieces may be separately compiled, and combined later. This facility is limited in its capability.

A program can be divided into 3 kinds of components: the main program (the traditional BASIC program), external subroutines and external functions.

Program components may refer to external subroutines or functions by declaring the target to be EXTERNAL (see DEF and SUBROUTINE statements).

A main program is distinguished by the absence of a DEF (function) or a SUBROUTINE declaration as its first line (ignoring REMarks, INCLUDE, PROGRAM or DATA ORIGIN). Typically, a main program has a DIM statement before any function or SUBROUTINE declarations (a program that has no DIMs and starts with a DEF or SUBROUTINE declaration will be interpreted as a separately compiled module). A CONCATENATION BUFFER SIZE statement marks a program component as a main program.

A separately compiled function must have a function declaration as its first source line (not counting REM, INCLUDE, PROGRAM or DATA ORIGIN). The END statement that matches the function definition signifies the end of the program text (multiple separately compiled functions may not be placed in a single text file).

Likewise, a separately compiled SUBROUTINE must have its declaration as the first source line.

Each component (except the main program) must have a PROGRAM ORIGIN statement to ensure that none of the compiled component objects overlap (note: the data space at the end of the program component must be taken into account!). Typically, the main program does not need a PROGRAM ORIGIN statement.

Since the data space for each component is allocated in one contiguous block (whether marked as COMMON or DIM), COMMON space must have enough room for the maximum data space required by all components containing a COMMON statement.

The compilation of each component is performed using the compiler as described in PASS I. When in PASS II, a set of directives must be given to the assembler, with one EQU for each separately compiled function or SUBROUTINE. Each EQU gives the name of the external subroutine or function, and the address specified in the corresponding PROGRAM ORIGIN statement. This provides a primitive linking facility.

After all modules are compiled (and assembled) error-free, then the object modules are combined using SDOSSYSGEN (for operation details, see the SDOS manual). The correct start address must be specified.

Example:

```
REM MAIN PROGRAM
    DEF ARF(X) EXTERNAL
    INPUT "VALUE to ARF?" Q
    PRINT ARF(Q)
    EXIT
END
```

The following is in a separate file:

```
PROGRAM ORIGIN :4ØØØ
DEF ARF(Z)
    RETURN SQR(Z)
END
```

An equate needs to be supplied when the main program is being assembled:

```
.ASM...
 Source File = ...
 Listing File = ...
 Object File = ...
 >ARF   EQU  $4ØØØ   AS PER PROGRAM ORIGIN
 >
```

Since ARF invokes no EXTERNAL modules, it may be compiled without giving special attention to the assembly step.

MOVING BASIC 1.4 TO SYSTEMS OTHER THAN SDOS

Due to the relatively simple structure of SDOS I/O calls, it is possible to move BASIC 1.4 to operating systems other than SDOS. Conceptually, the procedure is very simple: an SDOS simulator for that system needs to be constructed. Details on how the individual system calls operate can be found in the SDOS manual. Syscalls needing implementation are SYSCALL:OPEN, :CLOSE, :CREATE, :DELETE, :RENAME, :READA, :READB, :WRITEA, :WRITEB, :CHAIN, :EXIT, :DEBUG, :ERROREXIT, and SYSCALL:STATUS for SC:GETCOL. Since the syscall structure is regular and simple, most of the work will be invested in using the target OS facilities to simulate byte-addressable files.

To use the runtime package in a stand-alone environment, only as much of the SDOS simulator as will be used, need be coded; a simple simulator that supports only CRT I/O should occupy only a few hundred bytes.

The runtime package and the compiler are pure code and can run in an environment with interrupts enabled, providing the simulator package supplies enough room in the machine stack for interrupts. Neither the compiler nor runtime package touch the "I" bit in the processor's status byte.

CONVERTING ANOTHER BASIC PROGRAM TO SD BASIC

Programs can usually be converted from other BASIC systems without too much trouble. An 8 page CHESS program for an 8Ø8Ø BASIC was converted to SD BASIC in about 6 hours, of which 2 were spent typing in the program.

DIFFERENCES BETWEEN PROPOSED MINIMAL ANSII STANDARD AND SD BASIC

1) Multicharacter variable names.

2) No OPTION statement. All arrays and vectors have a lower bound of zero (strings have a lower bound of one).

3) Control flow is not necessarily in line number order.

4) No READ/ RESTORE/ DATA capability. A data initialization facility is provided instead.

Program transportablilty from SD BASIC to ANSI BASIC is only impaired if long names, string operations, block structure or file I/O statements are used significantly.

PERFORMANCE CHARACTERISTICS

This section details space and speed estimates for SD BASIC. The values given are only approximate and can vary from program to program.

SIZE OF VARIABLES

Each numeric variable occupies 6 bytes. Vectors and arrays occupy about 6 times their dimensioned size (in bytes). String variables use about as many bytes as their maximum dimension. Variable space can be reduced by clever use of Uniform Reference procedures.

PROGRAM SIZE

The compiled program uses approximately 3Ø% as many bytes as the source text for the program. The length of variable names has no effect on the resulting program size, so feel free to use readable names. We estimate about 15 bytes of code for each line of a long, complex BASIC program is normal.

RUNTIME PACKAGE SIZE

The runtime package occupies about 11K bytes, from location zero up. Note: this does not include the operating system package! Do not place any BASIC programs below $2EØØ.

COMPILER SIZE

The BASIC compiler needs 2ØK to perform a compilation. It will use more memory as needed, if available. In 2ØK, one can compile about a 4 page BASIC program.

MAXIMUM BASIC PROGRAM SIZE

With 11Kb runtime package and 32Kb user space (not counting SDOS), you should be able to build a 14ØØ line BASIC program (about 2Ø pages of source).

COMPILE TIME

The BASIC Compiler, including Pass I and II, processed about twenty (2Ø) source lines/second. The Benchmark Program included here took 78 seconds to compile 146 lines (using the COMPILE command) on a 2mHz 68ØØ with hard disk.

EXECUTION TIME

SD BASIC automatically uses 16 bit positive binary integers (internally) whenever possible instead of floating point numbers. This effects a significant savings at execution time when doing FOR-NEXT loops and subscripting, which comprise the bulk of BASIC programs.

A comparison with conventional BASICs indicates that a program dealing primarily with integers (array subscripts, FOR/NEXT loop indices, etc.) can run some 2-1Ø times faster on SD BASIC. Compute bound programs doing really ugly arithmetic should be some 3 to 5 times faster. I/O bound programs can actually run faster on SD BASIC since there is no interpretive overhead (all the time is spent computing or doing I/O!).

The following is a benchmark performance program and the results of that benchmark. Note the care taken to separate the times for the operation under test from the overhead of the test itself. Also note that disk I/O times will vary considerably depending on the technology of the disk drive and its controller.

The 68ØØ test was run under SDOS 1.1g with a cartridge disk and a 3ØHz clock interrupt on a 2MHz 68ØØ.

The 68Ø9 test was run under SDOS 1.1g with a Winchester disk and a 6ØHz clock interrupt on a 2mHz 68Ø9. The 68Ø9 is about 25% faster than the 68ØØ for compute-bound activities.

```
Basic 1.4 Benchmark Ø4/11/83
File to be used for test: junk.tmp
CPU chip and Clock rate: 2MHz 68ØØ, RTP14k on SDOS11g/SU with Cartridge disk
Time for Integer NEXT is 92 Microseconds
Time for Short Integer FOR-NEXT is 193 Microseconds
Time for Floating NEXT is 587 Microseconds
Time for Load and store Scalar variable is 113 Microseconds
Time for Assign Floating to Scalar variable is 165 Microseconds
Time for Assignment to Vector slot is 26Ø Microseconds
Time for Assignment to Array slot is 434 Microseconds
Time for Gosub/Return is 142 Microseconds
Time for Call/Return Subroutine with 1 argument is 83Ø Microseconds
Time for Integer Fetch and Add/Subtract/Logicalop is 116 Microseconds
Time for Integer Multiply is 232 Microseconds
Time for Floating Add Variable is 359 Microseconds
Time for Floating Subtract Variable is 39Ø Microseconds
Time for Floating Multiply by Variable is 2Ø35 Microseconds
Time for Floating Divide by Variable is 4695 Microseconds
Time for Cosine is 34649 Microseconds
Time for Small string Copy (5 bytes) is 439 Microseconds
Time for Big string Copy (1ØØ bytes) is 16Ø3 Microseconds
Time for Extend file and Write of 1ØØ Byte Record is 37572 Microseconds
Time for Write 1ØØ Byte Record is 446Ø5 Microseconds
Time for Sequential Read of 1ØØ Byte Record is 215Ø5 Microseconds
Time for Random Read of 1ØØ Byte Record is 96539 Microseconds
End of Benchmark Test
```

```
Basic 1.4 Benchmark Ø4/12/83
File to be used for test: junk.tmp
CPU chip and Clock rate: 2MHz 68Ø9, RTP14i with SDOS11g/SU and 5" Winches
Time for Integer NEXT is 75 Microseconds
Time for Short Integer FOR-NEXT is 155 Microseconds
Time for Floating NEXT is 5Ø2 Microseconds
Time for Load and store Scalar variable is 1Ø2 Microseconds
Time for Assign Floating to Scalar variable is 142 Microseconds
Time for Assignment to Vector slot is 2Ø4 Microseconds
Time for Assignment to Array slot is 317 Microseconds
Time for Gosub/Return is 1Ø8 Microseconds
Time for Call/Return Subroutine with 1 argument is 7Ø6 Microseconds
Time for Integer Fetch and Add/Subtract/Logicalop is 74 Microseconds
Time for Integer Multiply is 131 Microseconds
Time for Floating Add Variable is 323 Microseconds
Time for Floating Subtract Variable is 364 Microseconds
Time for Floating Multiply by Variable is 1554 Microseconds
Time for Floating Divide by Variable is 3264 Microseconds
Time for Cosine is 25Ø73 Microseconds
Time for Small string Copy (5 bytes) is 278 Microseconds
Time for Big string Copy (1ØØ bytes) is 7Ø5 Microseconds
Time for Extend file and Write of 1ØØ Byte Record is 16629 Microseconds
Time for Write 1ØØ Byte Record is 16112 Microseconds
Time for Sequential Read of 1ØØ Byte Record is 9896 Microseconds
Time for Random Read of 1ØØ Byte Record is 96262 Microseconds
End of Benchmark Test
```

```
    REM BENCHMARK TEST FOR BASIC 1.4

    Dim File/1/,File$(5Ø),Clock/2/
    Dim Vector[1ØØ],Array[2,5Ø]
    Dim BigSource$[1ØØ],BigTarget$[1ØØ]

Def CurrentTime
    Rem Return current time in seconds since midnite
    Dim SixBytes$(6)
    Read #Clock,SixBytes$
    Return ((SixBytes$[1]**8+Sixbytes$[2])*256+SixBytes$[3])/6Ø
End

Subroutine DisplayTime(TestName$,IterationCount,OverheadTimeperIteration)
    TimeperIteration=(CurrentTime-StartTime)/IterationCount-...
&                        OverheadTimeperIteration
    Print "Time for ";TestName$;" is";...
&         Int(TimeperIteration*1e6+.5);"Microseconds"
    Return Subroutine
End

    Print "Basic 1.4 Benchmark ";Date$
    Open #Clock,"Clock:"
    Input "File to be used for test: " File$
    Create #File,File$
    Input "CPU chip and Clock rate: " File$

    StartTime=CurrentTime
    For K=1 to 5ØØØØ\Next K
    DisplayTime("Integer NEXT",5ØØØØ,Ø)
    IntegerLoopoverhead=TimeperIteration

    StartTime=CurrentTime
    For K=1 to 1ØØØØ\For J=1 to 5\Next J\Next K
    DisplayTime("Short Integer FOR-NEXT",5ØØØØ,Ø)

    StartTime=CurrentTime
    For K=1.1 to 1ØØØØ.1\Next K
    DisplayTime("Floating NEXT",1ØØØØ,Ø)
    FloatingLoopOverhead=TimeperIteration

    StartTime=CurrentTime
    For K=1 to 5ØØØØ\Scalar=K\Next K
    DisplayTime("Load and store Scalar variable",...
&               5ØØØØ,IntegerLoopOverhead)
    ScalarLoadStoreOverhead=TimeperIteration

    StartTime=CurrentTime
    FloatingVariable=PI
    For K=1 to 5ØØØØ\Scalar=FloatingVariable\Next K
    DisplayTime("Assign Floating to Scalar variable",...
&               5ØØØØ,IntegerLoopOverhead)
    FloatingLoadStoreOverhead=TimeperIteration
```

```
    StartTime=CurrentTime
    For K=1 to 5ØØ\For J=1 to 1ØØ\Vector(J)=K\Next J\Next K
    DisplayTime("Assignment to Vector slot",5ØØØØ,IntegerLoopOverhead

    StartTime=CurrentTime
    For K=1 to 1ØØØ\For J=1 to 5Ø\Array(2,J)=K\Next J\Next K
    DisplayTime("Assignment to Array slot",5ØØØØ,IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 5ØØØØ\Gosub ToPlaceThatReturns\Next K
    DisplayTime("Gosub/Return",5ØØØØ,IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 5ØØØØ\Call OneArgumentSubroutine(K)\Next K
    DisplayTime("Call/Return Subroutine with 1 argument",5ØØØØ,...
&               IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 5ØØØØ\Scalar=K+752\Next K
    DisplayTime("Integer Fetch and Add/Subtract/Logicalop",5ØØØØ,...
&               ScalarLoadStoreOverhead+IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 516\For J=1 to 127\Scalar=K*J\Next J\Next K
    DisplayTime("Integer Multiply",516*127,...
&               ScalarLoadStoreOverhead+IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1.Ø to 1ØØØØ.Ø\Scalar=K+FloatingVariable\Next K
    DisplayTime("Floating Add Variable",1ØØØØ,...
&               FloatingLoadStoreOverhead+FloatingLoopOverhead)

    StartTime=CurrentTime
    Let Midpoint=1ØØØØ/2+PI
    For K=1.Ø to 1ØØØØ.Ø\Scalar=K-Midpoint\Next K
    DisplayTime("Floating Subtract Variable",1ØØØØ,...
&               FloatingLoadStoreOverhead+FlOatingLoopOverhead)

    StartTime=CurrentTime
    For K=1.Ø to 1ØØØØ.Ø\Scalar=K*FloatingVariable\Next K
    DisplayTime("Floating Multiply by Variable",1ØØØØ,...
&               FloatingLoadStoreOverhead+FloatingLoopOverhead)

    StartTime=CurrentTime
    For K=1.Ø to 1ØØØØ.Ø\Scalar=K/FloatingVariable\Next K
    DisplayTime("Floating Divide by Variable",1ØØØØ,...
&               FloatingLoadStoreOverhead+FloatingLoopOverhead)

    StartTime=CurrentTime
    For K=1.Ø to 1ØØØ.Ø\Scalar=COS(K)\Next K
    DisplayTime("Cosine",1ØØØ,...
&               FloatingLoadStoreoverhead+FloatingLoopOverhead)
```

```
    StartTime=CurrentTime
    Let Len(BigSource$)=5
    For K=1 to 5ØØØØ\BigTarget$=BigSource$\Next K
    DisplayTime("Small string Copy (5 bytes)",5ØØØØ,IntegerLoopOverhead)

    StartTime=CurrentTime
    Let Len(BigSource$)=1ØØ
    For K=1 to 5ØØØØ\BigTarget$=BigSource$\Next K
    DisplayTime("Big string Copy (1ØØ bytes)",5ØØØØ,IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 1ØØØ\Write #File,BigSource$\Next K
    DisplayTime("Extend file and Write of 1ØØ Byte Record",...
&                  1ØØØ,IntegerLoopOverhead)

    Position #File,Ø
    StartTime=CurrentTime
    For K=1 to 1ØØØ\Write #File,BigSource$\Next K
    DisplayTime("Write 1ØØ Byte Record",...
&                  1ØØØ,IntegerLoopOverhead)

    Position #File,Ø
    StartTime=CurrentTime
    For K=1 to 1ØØØ\Read #File,BigSource$\Next K
    DisplayTime("Sequential Read of 1ØØ Byte Record",...
&                  1ØØØ,IntegerLoopOverhead)

    StartTime=CurrentTime
    For K=1 to 1ØØØ\Read #File@1ØØ*INT(RND*1ØØØ),BigSource$\Next K
    DisplayTime("Random Read of 1ØØ Byte Record",...
&                  1ØØØ,IntegerLoopOverhead)

    Print "End of Benchmark Test"
    Exit

ToPlaceThatReturns: Return \ ! For Gosub test
Subroutine OneArgumentSubroutine(TheOnlyArgment)
    Return Subroutine
END
```

SAMPLE PROGRAMS

THE GAME OF LIFE

This program was originally designed as a bacterial growth simulation, but its properties as a digital kaleidoscope made it extremely popular among computer buffs. The program displays a "world" (think of a square Petri dish) of periods and asterisks on a screen (this program is not recommended for hardcopy terminals). When asked "WHAT NEXT?", the operator may add new life units (asterisks) by entering a row, column specification; he may run the simulation for several cycles by typing only a single number, or he may stop the program by typing "STOP". Typing "D" will cause the current world to be displayed. Simply pressing <CR> will cause a display of the next generation. Typing OUT filename will cause the output to be directed to another file; hardcopy can be obtained on a line printer.

This program uses many features of SD BASIC.

```
     REM ***** LIFE *****
     REM SIMULATES "LIFE" AS DEFINED BY THE MATHEMATICIAN JOHN CONWAY
     REM COMMANDS:
     REM     STOP        MEANS WHAT IT SAYS
     REM     D           MEANS "DISPLAY THE WORLD"
     REM     OUT file    SAYS PRINT GENERATIONS ON THE SPECIFIED FILE
     REM     <RETURN>    COMPUTE NEXT GENERATION AND DISPLAY
     REM     number      MEANS COMPUTE NEXT GENERATION number TIMES AND DISPLAY
     REM     row,col     INVERTS WHETHER THERE IS LIFE IN WORLD(row,col)

     DIM WORLD(21,21),WORLDCOPY(21,21)
     DIM DEATH/Ø/,LIFE/1/
     REM EDGES OF THE WORLD ALWAYS CONTAIN "DEATH"
     REM I.E., ROW Ø, COLUMN Ø, LAST ROW AND LAST COLUMN
     DIM GENERATIONNUMBER/Ø/
     DIM OUT/Ø/,DISPLAY$/".*"/
     DIM LINE$(8Ø)

LETTHEREBELIGHT:
     REM INITIALLY, MAKE EVERYTHING DEAD
     FOR I=Ø TO ROWS(WORLD)
         FOR J=Ø TO COLUMNS(WORLD)
             LET WORLD(I,J)=DEATH
             LET WORLDCOPY(I,J)=DEATH \ ! THIS MARKS EDGES AS "DEAD"
         NEXT J
     NEXT I

ASKFORWORK:
     INPUT "WHAT NEXT? " LINE$
     IF LINE$=""
     THEN GOSUB DOGENERATION\GOSUB DISPLAYGENERATION\GOTO ASKFORWORK
     IF UPPERCASE$(LINE$)="STOP" THEN EXIT
     IF UPPERCASE$(LINE$)="D" THEN GOSUB DISPLAYGENERATION\GOTO ASKFORWORK

     IF FIND(UPPERCASE$(LINE$),"OUT ")
     THEN
         LET LINE$=RIGHT$(LINE$,5)
         IF ERROR WHEN CLOSE #1 THEN REM WHO CARES?
         LET OUT=1
         CREATE #1,LINE$
         GOTO ASKFORWORK
     FI
     IF FIND(LINE$,",")
     THEN
         IF ERROR WHEN
             LET I=VAL(LINE$)
             LET J=VAL(RIGHT$(LINE$,FIND(LINE$,",")+1))
             LET WORLD(I,J)=NOT WORLD(I,J)
         THEN PRINT "Illegal coordinates"
         GENERATIONNUMBER=Ø
         GOTO ASKFORWORK
     FI
```

```
    IF ERROR WHEN
       COUNT=VAL(LINE$)
    THEN PRINT "Bad generation count"\GOTO ASKFORWORK
    FOR COUNT=1 TO COUNT DO GOSUB DOGENERATION
    GOSUB DISPLAYGENERATION
    GOTO ASKFORWORK

DOGENERATION: REM DO SIMULATION TO COMPUTE NEXT GENERATION
1ØØ1Ø REM FIRST, COPY THE WORLD
    FOR I=1 TO ROWS(WORLD)-1
        FOR J=1 TO COLUMNS(WORLD)-1
            WORLDCOPY(I,J)=WORLD(I,J)
        END
    END
1ØØ2Ø REM NOW PERFORM GUTS OF SIMULATION
    FOR I=1 TO ROWS(WORLD)-1
        FOR J=1 TO COLUMNS(WORLD)-1
            REM COMPUTE NUMBER OF NEIGHBORS THAT WORLD(I,J) HAS
            REM Ø,1 NEIGHBORS --> WORLD(I,J) DIES OF LONELINESS
            REM 2 NEIGHBORS --> THIS WORLD(I,J) SURVIVES UNCHANGED
            REM 3 NEIGHBORS --> WORLD(I,J) GROWS A LIFE UNIT
            REM > 4 NEIGHBORS --> WORLD(I,J) DIES OF OVERCROWDING
            ON WORLDCOPY(I-1,J-1)+WORLDCOPY(I-1,J)+WORLDCOPY(I-1,J+1)..
&              +WORLDCOPY(I,J-1)+WORLDCOPY(I,J+1)...
&              +WORLDCOPY(I+1,J-1)+WORLDCOPY(I+1,J)+WORLDCOPY(I+1,J+1).
&           GOTO DIE,SURVIVE,GROW
DIE:        WORLD(I,J)=DEATH
            CYCLE J

GROW:       WORLD(I,J)=LIFE
SURVIVE:    REM WORLD(I,J) DOESN'T CHANGE
        NEXT J
    NEXT I
    GENERATIONNUMBER=GENERATIONNUMBER+1
    RETURN


DISPLAYGENERATION: REM PRINT OUT WORLD
    PRINT #OUT, USING "GENERATION NUMBER: #####", GENERATIONNUMBER
    PRINT #OUT,"     ";
    FOR J=1 TO COLUMNS(WORLD)-1 DO PRINT #OUT, USING " ##",J;
    PRINT #OUT
    PRINT #OUT
    FOR I=1 TO ROWS(WORLD)-1
        PRINT #OUT, USING "##    ",I;
        FOR J=1 TO COLUMNS(WORLD)-1
            PRINT #OUT,DISPLAY$[WORLD(I,J)+1,1];"  ";
        NEXT J
        PRINT #OUT
    NEXT I
    RETURN

END
```

PHONEBOOK EXAMPLE PROGRAM

The following program uses many of the enhancements in SD BASIC to implement a "digital" telephone directory. It manages a database containing records that hold a person's name, title, company association, address and phone number. The program allows the operator to locate a phone number (or address) by looking up the name of the desired person, or the name of the desired company. Partial name specifications may be used if the operator does not know the complete name. Information about a person may be deleted or changed if necessary.

Commands are given to the program by typing a keyword (like FIND) followed by the desired name. The type of name required is displayed in < >s by the HELP command (i.e., filenames, person names or company names might be specified, depending on the command). Operation of the program should be self-explanatory.

The program stores only one kind of record, which has several fields (see REM PERSONRECORD below). The information fields of the record are stored as zero-padded strings so that each record is fixed size. Two special fields in each record allow that record to point to 1) another record containing an identical PERSONNAME$ field or 2) another record containing an identical PERSONCOMPANY$ field. Subroutines to read and write the entire record are used to make the rest of the program clear.

The program uses the keyed file package to allow associative storage and/or retrieval of a record by person's name or company name. The records and the key indexes are both stored in the same file. Since the key package does not allow duplicate keys, the program chains records containing identical keys together using a pointer. Each record is indexed on 2 fields: the PERSONNAME$ field and the PERSONCOMPANY$ field. The function FINDANDDISPLAYPERSON shows how an associative lookup is performed using the key package and the "SAMELINK"s. The subroutines ADDRECORD and DELETERECORD add a new record/delete an existing record from both indexes, and adjust the SAMELINKs accordingly. The function KEYREPLACE is used when adding a new record to the head of a chain of SAMELINKs.

The subroutines PAD, MODIFY and TRUNCATE all take advantage of the Call-by-reference parameter passing scheme to modify arguments. They are worth examining carefully.

The most straightforward top-level routine is DELETE1 (the word DELETE was the desired one, but it is a BASIC keyword and so could not be used). The other routines should be easy to understand once DELETE1 is understood.

The LOAD and DUMP routines are useful in any application of this sort, to provide for a backup facility, and to allow one to dump the database, so that changes in the database can be made by modifying the program and reLOADing it.

```
    REM "PHONEBOOK" PROGRAM
    REM KEEPS TRACK OF PEOPLE, THEIR ADDRESS AND PHONE NUMBER,...
    REM AND THE COMPANY FOR WHICH THEY WORK.
    REM USES KEY PACKAGE TO INDEX ON PEOPLE AND COMPANY NAMES.

    DIM CLEARSCREEN$/:C/,BACKSPACE$/:8/
    DIM COMMAND$(8Ø),PERSONKEY$(8Ø)/""/
    REM PERSONRECORD
    DIM PERSONNAMESAMELINK/Ø/,PERSONCOMPANYSAMELINK/Ø/
    DIM PERSONNAME$(25),PERSONTITLE$(2Ø)
    DIM PERSONCOMPANY$(2Ø),PERSONSTREET$(25)
    DIM PERSONCITY$(2Ø),PERSONSTATECOUNTRY$(2Ø),PERSONZIP$(9)
    DIM PERSONPHONE$(15)

    INCLUDE "KEY.BAS"

SUBROUTINE READPERSONRECORD
    READ #1@PERSONRECORD,PERSONNAMESAMELINK,PERSONCOMPANYSAMELINK,...
&                        PERSONNAME$,PERSONTITLE$,...
&                        PERSONCOMPANY$,PERSONSTREET$,...
&                        PERSONCITY$,PERSONSTATECOUNTRY$,PERSONZIP$,..
&                        PERSONPHONE$
    RETURN SUBROUTINE
END

SUBROUTINE PAD(PAD$)
    FOR PADINDEX=LEN(PAD$)+1 TO MAXLEN(PAD$) DO PAD$[PADINDEX]=Ø
    LET LEN(PAD$)=MAXLEN(PAD$)
    RETURN SUBROUTINE
END

SUBROUTINE WRITEPERSONRECORD
    PAD(PERSONNAME$)
    PAD(PERSONTITLE$)
    PAD(PERSONCOMPANY$)
    PAD(PERSONSTREET$)
    PAD(PERSONCITY$)
    PAD(PERSONSTATECOUNTRY$)
    PAD(PERSONZIP$)
    PAD(PERSONPHONE$)
    WRITE #1@PERSONRECORD,PERSONNAMESAMELINK,PERSONCOMPANYSAMELINK,...
&                        PERSONNAME$,PERSONTITLE$,...
&                        PERSONCOMPANY$,PERSONSTREET$,...
&                        PERSONCITY$,PERSONSTATECOUNTRY$,PERSONZIP$,..
&                        PERSONPHONE$
    RETURN SUBROUTINE
END
```

```
SUBROUTINE PRINTPERSONRECORD(WHERE)
    PRINT #WHERE,PERSONNAME$
    PRINT #WHERE,PERSONTITLE$
    PRINT #WHERE,PERSONCOMPANY$
    PRINT #WHERE,PERSONSTREET$
    PRINT #WHERE,PERSONCITY$
    PRINT #WHERE,PERSONSTATECOUNTRY$
    PRINT #WHERE,PERSONZIP$
    PRINT #WHERE,PERSONPHONE$
    PRINT #WHERE
    RETURN SUBROUTINE
END

DEF FINDANDDISPLAYPERSON
    REM THIS FUNCTION RETURNS FALSE IF "PERSONKEY$" CANNOT BE FOUND
    REM ELSE RETURNS TRUE AFTER DISPLAYING RECORD ABOUT PERSON
    IF ERROR WHEN
       PERSONRECORD=KEY(1,1,PERSONKEY$)
    THEN IF ERR=1Ø75 THEN NOSUCHPERSON ELSE ERROR
    PRINT CLEARSCREEN$
    READPERSONRECORD
    PRINTPERSONRECORD(Ø)
    RETURN TRUE

NOSUCHPERSON: REM CAN'T FIND THE PERSON DESIRED, TRY KEYNEXT
    PERSONNAMESAMELINK=Ø
NEXTPERSON: REM TRY FOR NEXT PERSON
    IF PERSONNAMESAMELINK<>Ø
    THEN
        REM MORE THAN ONE GUY WITH THE SAME NAME
        PERSONRECORD=PERSONNAMESAMELINK
        GOTO DISPLAYNEXTPERSON
    FI
    IF ERROR WHEN
        PERSONRECORD=KEYNEXT(1,1,PERSONKEY$)
    THEN IF ERR=1ØØ1
          THEN
              PRINT "CAN'T FIND PERSON SELECTED."
              PERSONRECORD=Ø\COMMAND$=""\RETURN FALSE
          ELSE ERROR
```

```
DISPLAYNEXTPERSON:
    PRINT CLEARSCREEN$;"PERHAPS YOU MEANT: "
    PRINT
    READPERSONRECORD
    PRINTPERSONRECORD(Ø)
    INPUT 'ENTER "YES" OR "NO", <CR> MEANS "NEXT" ' COMMAND$
    IF COMMAND$="" THEN NEXTPERSON
    ELSEIF UPPERCASE$(COMMAND$)="YES" THEN COMMAND$=""\RETURN TRUE
    ELSEIF UPPERCASE$(COMMAND$)="NO"
        THEN
            COMMAND$=""
            PERSONRECORD=Ø
            RETURN FALSE
    ELSE PERSONRECORD=Ø\RETURN FALSE
END

SUBROUTINE ADDRECORD
    REM THIS SUBROUTINE ADDS A PERSON RECORD TO THE DATABASE
    REM BY INSERTING BOTH PERSONNAME$ AND PERSONCOMPANY$ AS KEYS
    REM IN KEY INDEXES 1 AND 2, RESPECTIVELY.
    REM IF A KEY ALREADY EXISTS, THE RECORD IS SIMPLY ADDED TO A CHAIN
    REM OF RECORDS THAT HAVE IDENTICAL KEYS. THIS WAY
    REM ALL PEOPLE IN THE SAME COMPANY ARE EASILY FOUND, AS
    REM ARE ALL PEOPLE WITH THE SAME NAME.
    PERSONNAMESAMELINK=Ø \ REM ASSUME NO OTHER IDENTICAL NAMES
    PERSONCOMPANYSAMELINK=Ø \ REM ASSUME NO OTHER IDENTICAL COMPANIES
    LET PERSONRECORD=GETSPACE(1,221)
    REM ADD PERSON TO NAME INDEX
    IF ERROR WHEN
        KEYINSERT(1,1,PERSONNAME$,PERSONRECORD)
    THEN
        REM THAT NAME ALREADY EXISTS, PLACE PERSON RECORD ON CHAIN
        IF ERR=1Ø76
        THEN PERSONNAMESAMELINK=...
&               KEYREPLACE(1,1,PERSONNAME$,PERSONRECORD)
        ELSE ERROR
    FI
    REM ADD PERSON TO COMPANY INDEX
    IF ERROR WHEN
        KEYINSERT(1,2,PERSONCOMPANY$,PERSONRECORD)
    THEN
        REM THAT COMPANY ALREADY EXISTS, PLACE PERSON RECORD ON CHAIN
        IF ERR=1Ø76
        THEN PERSONCOMPANYSAMELINK=...
&               KEYREPLACE(1,2,PERSONCOMPANY$,PERSONRECORD)
        ELSE ERROR
    FI
    WRITEPERSONRECORD
    RETURN SUBROUTINE
END
```

```
SUBROUTINE DELETERECORD
    REM DELETE THE FOUND RECORD
    REM THIS UNDOES WHAT ADDRECORD DOES.
    REM THIS MAY REQUIRE SIMPLE REMOVAL FROM A CHAIN
    REM IF THE CHAIN GETS EMPTY, THE KEY MUST BE DELETED!
    REM DELETE FROM NAME KEY CHAIN FIRST
    PERSONPREVIOUS=KEY(1,1,PERSONNAME$)\! ERROR CANNOT OCCUR HERE
    IF PERSONPREVIOUS=PERSONRECORD
    THEN
        REM THIS RECORD IS THE FIRST RECORD ON A NAME CHAIN
        IF PERSONNAMESAMELINK=Ø
        THEN
            REM THIS RECORD IS ONLY RECORD WITH THIS PERSON NAME
            KEYDELETE(1,1,PERSONNAME$) \! POOF GOES THE NAME KEY
        ELSE
            REM THERE ARE OTHER RECORDS WITH THE SAME NAME
            REM REPLACE CHAIN HEAD WITH POINTER TO REST OF CHAIN
            PERSONRECORD=...
&               KEYREPLACE(1,1,PERSONNAME$,PERSONNAMESAMELINK)
        FI
    ELSE
        REM THIS RECORD IS SOMEWHERE ON A CHAIN...
        REM OF RECORDS WITH SAME NAME
FINDPREVIOUSPERSON: REPEAT
                        READ #1@PERSONPREVIOUS,PERSONNEXT
                        IF PERSONNEXT=PERSONRECORD
                        THEN EXIT FINDPREVIOUSPERSON
                        PERSONPREVIOUS=PERSONNEXT
                    END
         REM FOUND RECORD IN CHAIN WHOSE "NEXT" POINTER...
         REM SELECTS RECORD TO BE DELETED
         REM REMOVE THIS RECORD FROM THE CHAIN
         WRITE #1@PERSONPREVIOUS,PERSONNAMESAMELINK
    FI
    REM NOW DELETE FROM COMPANY KEY CHAIN
    COMPANYPREVIOUS=KEY(1,2,PERSONCOMPANY$)\! NO ERROR POSSIBLE
    IF COMPANYPREVIOUS=PERSONRECORD
    THEN
        REM THIS RECORD IS THE FIRST RECORD ON A COMPANY CHAIN
        IF PERSONCOMPANYSAMELINK=Ø
        THEN
            REM THIS RECORD IS THE ONLY RECORD WITH THIS COMPANY NAME
            KEYDELETE(1,2,PERSONCOMPANY$) \! POOF GOES THE NAME KEY
        ELSE
            REM THERE ARE OTHER RECORDS WITH THE SAME COMPANY NAME
            REM REPLACE CHAIN HEAD WITH POINTER TO REST OF CHAIN
            PERSONRECORD=...
&               KEYREPLACE(1,2,PERSONCOMPANY$,PERSONCOMPANYSAMELINK)
        FI
    ELSE
        REM THIS RECORD IS SOMEWHERE ON A CHAIN...
        REM OF RECORDS OF SAME COMPANY
```



```
FINDPREVIOUSCOMPANY: REPEAT
                        READ #1@COMPANYPREVIOUS,PERSONNEXT,COMPANYNEXT
                        IF COMPANYNEXT=PERSONRECORD
                        THEN EXIT FINDPREVIOUSCOMPANY
                        COMPANYPREVIOUS=COMPANYNEXT
                    END
         REM FOUND RECORD IN CHAIN WHOSE "NEXT" POINTER...
         REM SELECTS RECORD TO BE DELETED
         REM REMOVE THIS RECORD FROM THE CHAIN
         WRITE #1@COMPANYPREVIOUS,PERSONNEXT,PERSONCOMPANYSAMELINK
    FI
    RETURN SUBROUTINE
END

SUBROUTINE MODIFY(MODIFYTITLE$,MODIFYTARGET$)
    PRINT MODIFYTITLE$;MODIFYTARGET$;
    FOR MODIFYCOUNT=1 TO LEN(MODIFYTARGET$)...
&       UNTIL MODIFYTARGET$[MODIFYCOUNT]=Ø DO PRINT BACKSPACE$;
    INPUT '' COMMAND$
    IF COMMAND$="" THEN RETURN SUBROUTINE
    MODIFYTARGET$=UPPERCASE$(COMMAND$)
    RETURN SUBROUTINE
END

SUBROUTINE TRUNCATEBLANKS(STRINGTOBETRUNCATED$)
    FOR STRINGTOBETRUNCATEDINDEX=LEN(STRINGTOBETRUNCATED$) TO 1 STEP -1
&       UNTIL STRINGTOBETRUNCATED$(STRINGTOBETRUNCATEDINDEX)<>:2Ø
    NEXT STRINGTOBETRUNCATEDINDEX
    LET LEN(STRINGTOBETRUNCATED$)=STRINGTOBETRUNCATEDINDEX
    RETURN SUBROUTINE
END
```

```
!****************************************
!          BEGIN MAIN PROGRAM
!****************************************

BEGIN: PRINT "PHONEBOOK V1.Ø (C) 1981 SOFTWARE DYNAMICS"
    LET COMMAND$="PHONEBOOK.DATA"
OPENFILE:
    IF ERROR WHEN
        OPEN #1,COMMAND$
    THEN
        IF ERR=1Ø11
        THEN
            PRINT "CAN'T FIND ";COMMAND$
            PRINT "ENTER NAME OF PHONEBOOK FILE,"
            PRINT 'ENTER THE WORD "CREATE" TO CREATE ';COMMAND$
            INPUT "OR ENTER <CR> TO EXIT: " PERSONNAME$
            IF PERSONNAME$="" THEN EXIT
            ELSEIF UPPERCASE$(PERSONNAME$)="CREATE"
            THEN
                CREATE #1,COMMAND$
                KEYINIT(1,1,25,9) \ REM INITIALIZE "PERSON" INDEX
                KEYINIT(1,2,2Ø,9) \ REM INITIALIZE "COMPANY" INDEX
            ELSE COMMAND$=PERSONNAME$\GOTO OPENFILE
        ELSE ERROR
    FI
PRINTMENU:
    PRINT CLEARSCREEN$;"COMMANDS: "
    PRINT "DUMP <FILE> -- DUMPS ENTIRE DATA BASE TO <FILE>"
    PRINT "LOAD <FILE> -- LOADS (OR ADDS) TO DATA BASE FROM <FILE>"
    PRINT "FIND <PERSON> -- FIND A PARTICULAR PERSON"
    PRINT "NEXT -- FIND NEXT PERSON"
    PRINT "COMPANY <COMPANYNAME> -- LOCATE A COMPANY"
    PRINT "NPIC -- FIND NEXT PERSON IN SAME COMPANY"
    PRINT "FIX <PERSON> -- CHANGE INFORMATION ABOUT A PERSON"
    PRINT "ADD <PERSON> -- ADD A PERSON TO THE PHONEBOOK"
    PRINT "DELETE <PERSON> -- DELETE A PERSON FROM THE PHONEBOOK"
    PRINT "EXIT -- LEAVE THIS PROGRAM"
    PRINT "HELP -- PRINTS THIS MENU"
    PRINT "<OTHER> -- IMPLIED FIND ON <OTHER>"
ASKCOMMAND:
    INPUT "OK> " COMMAND$
INSPECTCOMMAND:
    IF LEN(COMMAND$)=Ø THEN ASKCOMMAND
    LET COMMAND$=UPPERCASE$(COMMAND$)
    IF FIND(COMMAND$,"DUMP ")=1 THEN DUMP
    IF FIND(COMMAND$,"LOAD ")=1 THEN LOAD
    IF FIND(COMMAND$,"FIND ")=1
    THEN PERSONKEY$=RIGHT$(COMMAND$,6)\GOTO FIND1
    IF COMMAND$="NEXT" THEN FINDNEXTPERSON
    IF FIND(COMMAND$,"COMPANY ")=1 THEN COMPANY
    IF COMMAND$="NPIC" THEN NPIC
    IF FIND(COMMAND$,"FIX ")=1 THEN FIX
    IF FIND(COMMAND$,"ADD ")=1 THEN ADD
    IF FIND(COMMAND$,"DELETE ")=1 THEN DELETE1
```

```
    IF COMMAND$="EXIT" THEN EXIT
    IF COMMAND$="HELP" THEN PRINTMENU
OTHER: REM TRY TO FIND THE PERSON
    LET PERSONKEY$=COMMAND$
FIND1:
    IF FINDANDDISPLAYPERSON THEN ASKCOMMAND ELSE INSPECTCOMMAND

DELETE1:
    LET PERSONKEY$=RIGHT$(COMMAND$,8)
    IF NOT FINDANDDISPLAYPERSON THEN INSPECTCOMMAND
    DELETERECORD
    GOTO ASKCOMMAND

ADD: REM ADD A NEW PERSON
    LET PERSONNAME$=RIGHT$(COMMAND$,5)
    IF ERROR WHEN
       PERSONRECORD=KEY(1,1,PERSONNAME$)
    THEN
        REM THAT NAME ALREADY EXISTS!
        IF ERR=1Ø76
        THEN
            READPERSONRECORD
            PRINT "THAT NAME IS A DUPLICATE OF: "
            PRINTPERSONRECORD(Ø)
            INPUT 'ENTER COMMAND (<CR> MEANS "ADD ANYWAY")' COMMAND$
            IF COMMAND$<>"" THEN INSPECTCOMMAND
       FI
    FI
    INPUT "TITLE:          " PERSONTITLE$
    INPUT "COMPANY:        " PERSONCOMPANY$
    INPUT "STREET/SUITE:   " PERSONSTREET$
    INPUT "CITY:           " PERSONCITY$
    INPUT "STATE/COUNTRY:  " PERSONSTATECOUNTRY$
    INPUT "ZIP:            " PERSONZIP$
    INPUT "PHONE NUMBER:   " PERSONPHONE$
    LET PERSONCOMPANY$=UPPERCASE$(PERSONCOMPANY$)
    ADDRECORD
    GOTO ASKCOMMAND

FIX: LET PERSONKEY$=RIGHT$(COMMAND$,5)
    IF NOT FINDANDDISPLAYPERSON THEN INSPECTCOMMAND
    PRINT CLEARSCREEN$;
    DELETERECORD
    PRINT "TYPE <CR> TO LEAVE OLD VALUE ALONE"
    MODIFY("NAME:           ",PERSONNAME$)
    MODIFY("TITLE:          ",PERSONTITLE$)
    MODIFY("COMPANY:        ",PERSONCOMPANY$)
    MODIFY("STREET/SUITE:   ",PERSONSTREET$)
    MODIFY("CITY:           ",PERSONCITY$)
    MODIFY("STATE/COUNTRY:  ",PERSONSTATECOUNTRY$)
    MODIFY("ZIP:            ",PERSONZIP$)
    MODIFY("PHONE NUMBER:   ",PERSONPHONE$)
    ADDRECORD
    GOTO ASKCOMMAND
```

```
FINDNEXTPERSON:
    IF PERSONRECORD=Ø
    THEN PRINT "NOBODY SELECTED, CAN'T"\GOTO ASKCOMMAND
    IF PERSONNAMESAMELINK<>Ø
    THEN
        REM MORE THAN ONE GUY WITH SAME NAME
        PERSONRECORD=PERSONNAMESAMELINK
    ELSE
        IF ERROR WHEN
           PERSONRECORD=KEYNEXT(1,1,PERSONKEY$)
        THEN IF ERR=1ØØ1
             THEN PERSONRECORD=Ø\PRINT "CAN'T"\GOTO ASKCOMMAND
             ELSE ERROR
    FI
    PRINT CLEARSCREEN$;"PERHAPS YOU MEANT: "
    PRINT
    READPERSONRECORD
    PRINTPERSONRECORD(Ø)
    INPUT 'ENTER "YES", "NO", <CR> FOR "NEXT" OR COMMAND: ' COMMAND$
    IF LEN(COMMAND$)=Ø THEN FINDNEXTPERSON
    ELSEIF UPPERCASE$(COMMAND$)="YES" THEN ASKCOMMAND
    ELSEIF UPPERCASE$(COMMAND$)="NO" THEN ASKCOMMAND
    ELSE INSPECTCOMMAND

NPIC:
    REM FIND NEXT PERSON WITHIN COMPANY
    IF PERSONRECORD=Ø
    THEN PRINT "NO COMPANY SELECTED"\GOTO ASKCOMMAND
    IF PERSONCOMPANYSAMELINK<>Ø
    THEN
        REM MORE THAN ONE GUY AT SAME COMPANY
        PERSONRECORD=PERSONCOMPANYSAMELINK
        GOTO COMPANYDISPLAY
    ELSE PRINT "NO MORE PEOPLE THERE..."\GOTO ASKCOMMAND

COMPANY:
    LET PERSONKEY$=RIGHT$(COMMAND$,9)
    IF ERROR WHEN
       PERSONRECORD=KEY(1,2,PERSONKEY$)
    THEN IF ERR=1Ø75 THEN NOSUCHCOMPANY ELSE ERROR
COMPANYDISPLAY: PRINT CLEARSCREEN$; "PERHAPS YOU MEANT: "
    PRINT
    PERSONKEY$=PERSONNAME$ \ REM IN CASE "NEXT" IS INVOKED
    READPERSONRECORD
    PRINTPERSONRECORD(Ø)
    INPUT 'ENTER "YES","NO",<CR> FOR "NPIC" OR COMMAND: ' COMMAND$
    IF LEN(COMMAND$)=Ø THEN NPIC
    ELSEIF UPPERCASE$(COMMAND$)="YES" THEN ASKCOMMAND
    ELSEIF UPPERCASE$(COMMAND$)="NO" THEN ASKCOMMAND
    ELSE INSPECTCOMMAND
```



```
NOSUCHCOMPANY: REM CAN'T FIND THE COMPANY DESIRED, TRY KEYNEXT
    PRINT CLEARSCREEN$;"CAN'T FIND COMPANY: ";PERSONKEY$
NEXTCOMPANY: REM TRY FOR NEXT COMPANY
    IF ERROR WHEN
        PERSONRECORD=KEYNEXT(1,2,PERSONKEY$)
    THEN IF ERR=1ØØ1
         THEN
             PRINT "CAN'T FIND SELECTED COMPANY."
             PERSONRECORD=Ø\GOTO ASKCOMMAND
         ELSE ERROR
    READPERSONRECORD
    PRINT "PERHAPS YOU MEANT: ";PERSONCOMPANY$
    INPUT 'ENTER "YES" OR "NO";<CR> MEANS "NEXT" ' COMMAND$
    IF LEN(COMMAND$)=Ø THEN NEXTCOMPANY
    ELSEIF UPPERCASE$(COMMAND$)="YES" THEN COMPANYDISPLAY
    ELSEIF UPPERCASE$(COMMAND$)="NO"
        THEN
            PERSONRECORD=Ø
            GOTO ASKCOMMAND
    ELSE PERSONRECORD=Ø\GOTO INSPECTCOMMAND

LOAD: REM LOAD CONTENTS OF SEQUENTIAL FILE INTO PHONEBOOK
    LET COMMAND$=RIGHT$(COMMAND$,6)
    OPEN #2,COMMAND$
    PRINT "LOADING ";COMMAND$
LOADLOOP:
    INPUT #2,PERSONNAME$
    IF EOF(2) THEN CLOSE #2\GOTO ASKCOMMAND
    IF PERSONNAME$="" THEN LOADLOOP
    TRUNCATEBLANKS(PERSONNAME$)
    INPUT #2,PERSONTITLE$
    TRUNCATEBLANKS(PERSONTITLE$)
    INPUT #2,PERSONCOMPANY$
    TRUNCATEBLANKS(PERSONCOMPANY$)
    INPUT #2,PERSONSTREET$
    TRUNCATEBLANKS(PERSONSTREET$)
    INPUT #2,PERSONCITY$
    TRUNCATEBLANKS(PERSONCITY$)
    INPUT #2,PERSONSTATECOUNTRY$
    TRUNCATEBLANKS(PERSONSTATECOUNTRY$)
    INPUT #2,PERSONZIP$
    TRUNCATEBLANKS(PERSONZIP$)
    INPUT #2,PERSONPHONE$
    TRUNCATEBLANKS(PERSONPHONE$)
    PRINT PERSONNAME$
    ADDRECORD
    GOTO LOADLOOP
```

```
DUMP: REM DUMP PHONE NUMBER FILE ALPHABETICALLY BY PERSON
    LET COMMAND$=RIGHT$(COMMAND$,6)
    CREATE #2,COMMAND$
    PRINT "DUMPING DATABASE..."
    LET PERSONKEY$=""
    PERSONNAMESAMELINK=Ø
DUMPNEXTPERSONLOOP:
    IF PERSONNAMESAMELINK<>Ø
    THEN
        REM MORE THAN ONE GUY WITH THE SAME NAME
        PERSONRECORD=PERSONNAMESAMELINK
    ELSE
        IF ERROR WHEN
            PERSONRECORD=KEYNEXT(1,1,PERSONKEY$)
        THEN IF ERR=1ØØ1
             THEN CLOSE #2\GOTO ASKCOMMAND
             ELSE ERROR
    FI
    READPERSONRECORD
    PRINTPERSONRECORD(2)
    GOTO DUMPNEXTPERSONLOOP

END
```

RUNTIME ERROR MESSAGES

Ø - Program completed normally
1 - Operator requested Attention
2 - Not used
3 - Not used
4 - Not used
5 - Not used
6 - RETURN without GOSUB
7 - Conversion Error
8 - Input Buffer Overflow
9 - Array or Vector Subscript out of range
1Ø - Runtime package self-checksum failed --> Suspect damaged RTP
11 - String Subscript out of range
12 - String subscript too large
13 - Undefined Line Number encountered
14 - Arithmetic Overflow
15 - Non-Integer operand to Logical operator (& ! XOR COM **)
16 - Concatenated String exceeds CATMAX
17 - Tab count > 255
18 - Invalid FORMAT string
19 - I can't store that value into a byte
2Ø - Illegal Argument to SIN/COS/TAN/ATN
21 - Logarithm of Ø or negative number
22 - Square root attempted on negative number
23 - PEEK or POKE address < Ø or > 65535, or not an integer
24 - POKE value < Ø or > 255, or not an integer
25 - Attempt to POKE runtime package
26 - Version number doesn't match BASIC Runtime Package
27 - Wrong number of arguments to function/subroutine
28 - Data space for BASIC program overlaps SDOS
29 - Basic Program overlaps Runtime Package
5Ø - Channel number > 255
52 - File name is too long
6Ø - File position < Ø or >= 2^31

KEYED FILE PACKAGE ERRORS

1ØØ1 - End of File encountered
1Ø75 - No such key
1Ø76 - Duplicate key
1Ø77 - Key branch Factor not large enough

COMMONLY ENCOUNTERED SDOS ERRORS

1Ø11 - No such file
1Ø23 - Filename doesn't start with A through Z or $
1Ø31 - Channel is already open
1Ø32 - Channel is closed

Other error codes can be found in the SDOS Manual.

KEYWORDS

The following words are reserved keywords in BASIC 1.4, and may not be used for variable, subroutine, function or parameter names, or be used as labels. Keywords are recognized regardless of whether the constituent characters are lower or upper case.

```
DIM COMMON REM PROGRAM DATA CONCATENATION
INCLUDE
ON ERROR GOTO ELN
GOSUB POP RETURN
FOR TO STEP CYCLE NEXT
LET
PRINT USING FORMAT TAB INPUT READ WRITE
STOP EXIT
WHILE UNTIL DO END
REPEAT UNLESS WHEN
IF THEN ELSE FI ELSEIF
POSITION RESTORE OPEN CREATE CLOSE DELETE RENAME
CHAIN
CALL POKE DEBUG SYSCALL
SUBROUTINE DEF EXTERNAL
LEN MAXLEN LEFT$ MID$ RIGHT$ ASC CHR$
COM ATN SIN COS TAN LOG EXP SQR INT ABS SGN COL VAL PEEK FIND
RND ERR PI AND OR NOT EOF ROWS COLUMNS
TRUE FALSE XOR
CAT DATE$ TIME$ COPYRIGHT$ NUM$ NUMF$ HEX$ UPPERCASE$
LOWERCASE$
```


LANGUAGE SUMMARY

STATEMENTS	FUNCTIONS
PRINT	PI
PRINT USING	SIN
FORMAT	COS
LET	TAN
INPUT	ATN
GOTO	LOG
IF-THEN-ELSE-ELSEIF	EXP
FOR-NEXT/CYCLE	SQR
GOSUB/RETURN	INT
GOSUB POP	ABS
ON GOTO/GOSUB	SGN
ON ERROR WHEN/DO/GOTO	ERR (error number)
IF ERROR WHEN	ELN (error line number)
ERROR	LEN
REM (or "!")	VAL (of string)
DEF	COM (logical complement)
END	PEEK
OPEN	EOF (end file test)
CREATE	NOT (IF cond invert)
CLOSE	FIND (string in string)
DELETE	MID$
RENAME	LEFT$
PRINT #	RIGHT$
PRINT # USING	DATE$
INPUT #	TIME$
READ # (binary)	NUM$ (unformatted conversion)
WRITE # (binary)	NUMF$ (formatted conversion)
POSITION #	HEX$ (hex conversion)
CHAIN	ASC
CALL	COL (returns column position)
SUBROUTINE	CHR$
CALL (assembly routine)	UPPERCASE$
SYSCALL (SDOS interface)	LOWERCASE$
DEBUG	IF-THEN-ELSE-FI function
DIM/COMMON	MAXLEN
POKE	ROWS
PROGRAM ORIGIN	COLUMNS
DATA ORIGIN	
INCLUDE	RND
WHILE/UNTIL DO END	TRUE
REPEAT UNLESS/WHEN END	FALSE
GOTO ELN (used in error recovery)	
EXIT/STOP	
EXIT subroutine/labelname/indexvariable	

DATA TYPES

9 digit decimal floating point
16 bit positive integers
Hex numbers
Characters strings to 65534 characters
String arrays
Numeric vectors
Numeric arrays
Byte vectors

OPERATORS

+ - * / ^
& (and)
! (or)
XOR
** (shift)
- (negate)
CAT (string concatenation)
[] (substrings)

FORMATTED OUTPUT

Money format - floating dollar / trailing minus
Exponential format
Formatted numbers available as strings (NUMF$)

I/O

Channel oriented
ASCII (print/input) variable length records
Binary (read/write) reads any file accessible by byte
Random positioning
Multi-key file access procedures

MISCELLANEOUS

Multiple statements per line
Multiple statements in THEN/ELSE clauses
Line numbers needed only as targets of GOTO/GOSUB/ON ERROR
High speed execution
Error reporting line # at runtime
Explicit pointer to compile time errors
Compiled program is ROM-able
Line number tracing, single step, and breakpoint facility
Screen/File position control integrated into I/O statements
Structured programming constructs
Error trapping
Parameterized subroutines and functions
Excellent documentation