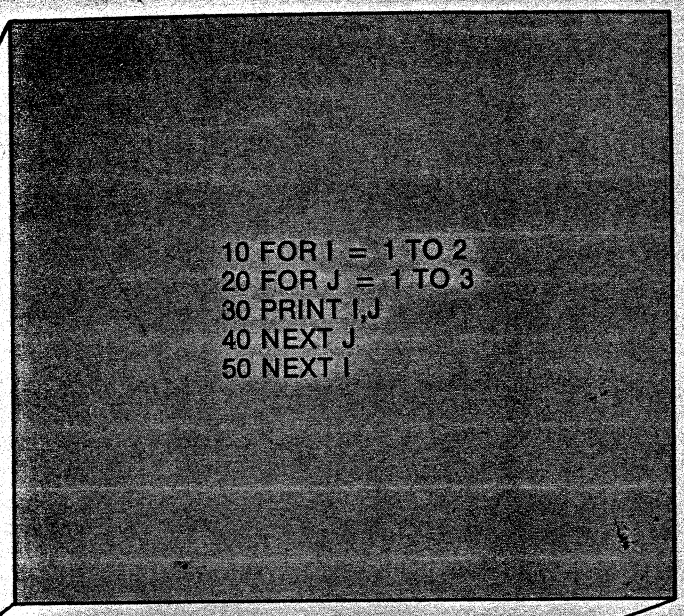
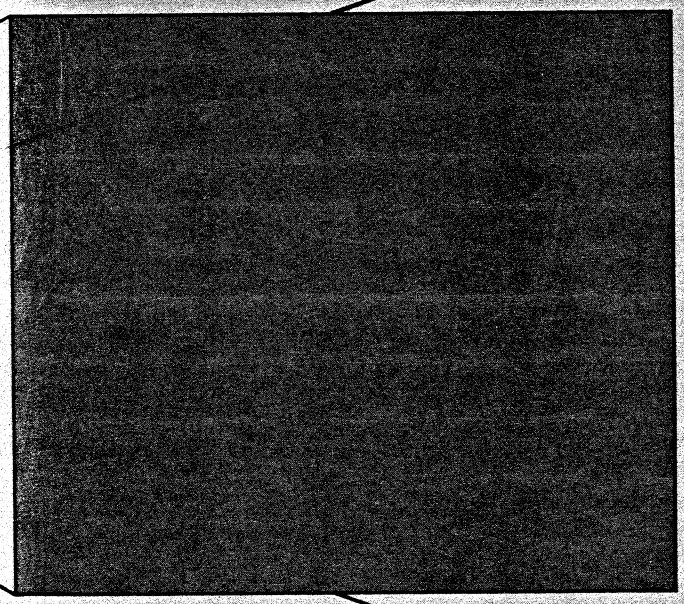
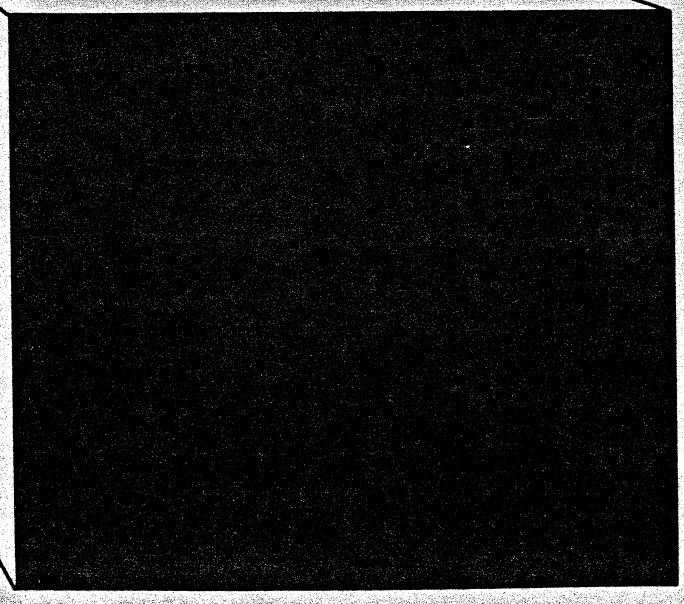


BASIC REFERENCE MANUAL

(ROM, 65D and 65U)





OHIO SCIENTIFIC SOFTWARE

$6.95

OHIO SCIENTIFIC

BASIC REFERENCE MANUAL

If you discover misprints or errors, please send a letter to the attention of: Documentation Department, Ohio Scientific Inc., 1333 S. Chillicothe Road Aurora, Ohio, 44202

CONTENTS

CHAPTER PAGE

CHAPTER 2

ELEMENTS OF BASIC

A. CONSTANTS

There are three types of constants in BASIC: numeric, string, and logical.

NUMERIC CONSTANTS

Numeric constants may have one of three forms: integer, real, and exponential. The integer form consists of a signed or unsigned string of decimal digits with no decimal point. Examples of the integer form would be 3 and −596. The real form has a decimal point. Examples of the real form would be 3.57 and −Ø.56. The exponential form is rEn where r is in real form and n is in integer form. The exponential form corresponds to scientific notation. For example, .ØØØØØ156789 in exponential form would be 1.56789E-6 and 123778641 would be 1.23778641E+8.

All three forms of numeric constants are converted internally to the exponential form. The range of values is from −1.7Ø14 E+38 to 1.7Ø14 E+38 (2 to the power 127). The smallest positive value is 2 to the −128 power or 2.9387 E-39.

If a program calculates a value greater than 1.7014 E+38 an overflow error occurs. If a program calculates a value less than −1.7Ø14 E+38 no underflow error occurs; a value of zero results.

Numeric constants have nine significant digits in the disk BASICs and six significant digits in ROM BASIC. If more digits are used (not including the decimal point or exponent) the number is truncated. For example, 98765432111 is 9.87654321E+1Ø in OS-65D.

STRING CONSTANTS

A string constant consists of a collection of up to 255 characters enclosed in quotes. Examples of string constants are "RATIO","3.9", and "". Any character in the ASCII table except " (double quotes) may be used in a string constant.

LOGICAL CONSTANTS

There are two logical constants in BASIC. They are TRUE and FALSE. TRUE is represented internally by −1 (all bits set) and FALSE by Ø (all bits reset).

B. VARIABLES

A constant can be represented by a variable. There are three types of variables in BASIC: numeric, integer, and string. Each type may be simple or subscripted. Simple variables are discussed in this chapter; subscripted variables are discussed in Chapter 6.

NUMERIC VARIABLES

A variable name consists of either a letter or a letter followed by another letter or decimal digit. Examples of numeric variables are A, XB, and S2.

A name with more than two letters can be used. Their use, however, should be watched closely because only the first two characters of a variable name are stored. Thus COVE and COUNT are considered to be the same variable (CO).

INTEGER VARIABLES

Numbers can be stored in integer form in BASIC by adding a % (percent sign) to the variable name. Examples of valid integer variable names would be A%, BA%, and S2%. The range of an integer variable in −32767 to +32767. Integer variables cannot be used with BASIC in ROM.

STRING VARIABLES

String variables are represented in BASIC by adding a $ (dollar sign) to the variable name. Examples of valid string variable names would be A$, BA$, and S2$.

C. ARITHMETIC OPERATORS AND EXPRESSIONS

The arithmetic operators are represented by the following characters:

+	addition
−	subtraction
*	multiplication
/	division
^	exponentiation

The carat representing exponentiation is entered by <SHIFT/N> on polled keyboards.

An arithmetic expression calculates a numerical value. It consists of a list of variables or constants separated by arithmetic operators or brackets. Examples of arithmetic expressions are A+B, B+3, A + 5, and (B ^2)−4*A*C.

No two operators should appear next to each other in an expression. The expression A/−2 is valid but should be written as A/(−2).

Multiplication is never implied. Thus 3(A+B) and (A+B)(A+C) are invalid and should be written as 3*(A+B) and (A+B)*(A+C).

In order to avoid ambiguity in the evaluation of expressions a preference is established among the operators. The preference is 1) brackets, 2) exponentiation, 3) negation, 4) multiplication and division, and 5) addition and subtraction. Expressions enclosed in brackets are performed first starting with the innermost pair of brackets. All operations at one level are performed before proceeding to the next level. Thus

2 + 16 / 2 ^ 3 is 4

Operations at the same level are performed from left to right. Thus

2 ^ 3 ^ 2 is (2 ^ 3) ^ 2 or 64

It is recommended that the programmer insert brackets in expressions to make them more readable and to ensure that the correct calculation is performed.

D. BIT OPERATORS AND, OR, AND NOT

The logical (or bit) operators AND, OR, and NOT operate bit-by-bit on the internal binary representations of the numbers. Each bit in the result is determined by comparing corresponding bits in the two numbers.

The operator AND sets a bit in the result to 1 if both corresponding bits in the numbers are 1. The operator OR sets a bit to 1 in the result if either one or both bits in the numbers are 1. The operator NOT operates on a single number and reverses the bits.

X	Y	X AND Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	Ø

OR

X	Y	X AND Y
1	1	1
1	Ø	1
Ø	1	1
Ø	Ø	Ø

NOT

X	NOT X
1	Ø
Ø	1

Some examples will serve to show how the logical operations work:

63 AND 16=16

$$\begin{array}{r} 111111_2 \\ \text{AND } Ø1ØØØØ_2 \\ \hline Ø1ØØØØ_2 \end{array}$$

63=binary 111111
and 16=binary 1ØØØØ
so 63 AND 16=16.

15 AND 14=14

15= binary 1111 and 14=binary 111Ø so 15 AND 14=binary 111Ø=14.

−1 AND 8=8

The two's complement representation of −1 is 1111111111111111_2 (all bits of the two-byte number set to 1). Therefore, −1=binary 1111111111111111 and 8=binary 1ØØØ, so −1 AND 8=8.

4 OR 2=6

4=binary 1ØØ and 2=binary 1Ø so 4 OR 2=binary 11Ø=6.

−1 OR −2=−1	−1=binary 1111111111111111 and −2= 1111111111111110, so −1 OR −2=−1.
NOT 0=−1	The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of −1.
NOT X=−(X+1)	the two's complement of any number is the bit complement plus one.

A typical use of logical operations is 'masking', testing a binary number for some predetermined pattern of bits. Such numbers might come from the computer's input ports and would then reflect the condition of some external device.

The action of the bit operators on the logical values −1 (TRUE) and 0 (FALSE) is the same as the familiar Boolean operators in the propositional calculus. This action is reviewed in the next section.

E. RELATIONAL OPERATORS AND EXPRESSIONS

The relational operators are represented by the following characters:

=	equal
<	less than
>	greater than
<=or=<	less than or equal
>=or=>	greater than or equal
<>or><	not equal

Just as an arithmetic expression calculates a numerical value a relational expression calculates one of the two logical values TRUE or FALSE. Examples of relational expressions are:

A < = B	Is A less than or equal to B?
(A2 − B) <> 3	Is A2 minus B unequal to 3?

The bit operators AND, OR, and NOT can be used in relational expressions. If E1 and E2 are logical constants then

E1 AND E2	is TRUE only when both E1 and E2 are TRUE, otherwise FALSE
E1 OR E2	is TRUE when either one or both of E1 and E2 are TRUE, otherwise FALSE
NOT E1	is TRUE when E1 is FALSE, otherwise FALSE

Examples would be the following expressions which are in fact equivalent:

A < 0 OR A > 3

NOT (A >= 0 AND A <= 3)

Relational operators may not be chained. The expression

1 < A < 5

is invalid and should be written

(1 < A) AND (A < 5)

The order of preference of arithmetic operators extends to relational operators and bit operators. The order of preference is 1) brackets, 2) exponentiation, 3) negation, 4) multiplication and division, 5) addition and subtraction, 6) all relational operators, 7) NOT, 8) AND, 9) OR. For example

NOT 2 * 3 > 5 is FALSE

F. COMPARING STRINGS

Strings may be compared using relational operators. The comparison is made in the same manner as a dictionary ordering. Corresponding characters in the two strings are compared moving from left to right. One character is considered less than another if it precedes it in the ASCII table. Thus "ABE" is less than "ABF", "2" is greater than "12", and "$" is less than "%". An example of a relational expression involving strings is

NOT (A$ <= B$ AND C > 5)

G. STRING EXPRESSIONS

String expressions consist of string constants, string functions, or string variables connected by the string operator +. The result of a string expression is a string. The string operator + means concatenation.

For example,

```
1Ø A$="HELLO "
2Ø B$="THERE"
3Ø C$ = A$ + B$
4Ø PRINT C$
```

results in the output:

```
HELLO THERE
```

CHAPTER 3

ASSIGNMENT, INPUT AND OUTPUT

BASIC programs usually input data from the keyboard and output data to the screen. They can, however, communicate with a variety of other devices. Tape and disk input and output are discussed in CHAPTERS 1Ø and 11.

A. LET STATEMENT

The assignment of values to variables is performed by the LET statement. The forms are:

```
N  LET A = B
N  A = B
```

where N is a line number, A is a variable, and B is an expression. The keyword LET is optional. The expression B is evaluated and its value is assigned to A. Examples of valid assignment statements are:

```
1Ø LET X = 3.14159
2Ø A$ = "YES" + B$
3Ø Z = 2 < 3
4Ø N% = 3.999
```

Numeric and integer variables may be assigned either numeric or logical values. If a numeric value is assigned to an integer variable it is truncated. The value assigned to N% above is the integer 3. The value assigned to Z above is −1, representing the logical value TRUE. String variables may be assigned only string values. Attempts to assign values to the wrong type of variable results in a type-mismatch error.

Chaining assignment statements as in the statement

```
1Ø A = B = C
```

will evaluate A to a logical value. The value of A will be −1 (TRUE) if B is equal to C, or Ø (FALSE) if B is not equal to C. So, the expression

```
1Ø A=B=C=Ø
```

would be evaluated from left to right as

```
1Ø A=((B=C)=Ø)
```

B. INPUT STATEMENT

Input is obtained using the INPUT statement. The forms are:

```
N INPUT S
N INPUT#M, S
```

where N is a line number, S is a list of variables separated by commas, and M is a device number.

An example of the first form is:

```
1Ø INPUT A,B
```

When this INPUT statement is executed, a question mark appears on the screen. Values are entered from the keyboard separated by commas.

The INPUT statement allows a comment to be printed along with the question mark. For example:

```
1Ø INPUT "YES OR NO";A$
```

displays

```
YES OR NO?
```

on the screen. The response from the keyboard is assigned to the string variable A$.

The second form is discussed under DEVICE SPECIFIED INPUT AND OUTPUT (Section H).

C. READ AND DATA STATEMENTS

The READ and DATA statements are always used together. The READ statements "read" the values in the DATA statements.

The DATA statement has the form

```
N DATA S
```

where N is a line number and S is a list of constants separated by commas. For example,

```
1Ø DATA 1.5, "HI", HI, −66
```

Strings may appear either quoted or unquoted. If unquoted, leading blanks are ignored and trailing blanks are included.

The values appearing in DATA statements are combined into a list in the order in which they appear. Thus the statement

```
1Ø DATA 2,3,5
```

is equivalent to the two statements

```
1Ø DATA 2,3
2Ø DATA 5
```

The READ statement has the form

```
N READ S
```

where N is a line number and S is a list of variables separated by commas. Each READ statement assigns values to the variables in its list by accessing the DATA list. The next READ statement proceeds in the DATA list where the previous READ statement left off. For example,

```
1Ø READ A,B$,C$
2Ø READ C
3Ø DATA 1.5,TYPE
4Ø DATA 4Ø,5Ø
```

is equivalent to

```
1Ø LET A=1.5
2Ø LET B$="TYPE"
3Ø LET C$="4Ø"
4Ø LET C=5Ø
```

Numeric values may be read into string variables. However, if an attempt is made to read a string into a numeric variable, a syntax error occurs in the line containing the string.

If there are more items in the DATA list than are read, the rest are ignored. On the other hand, if the DATA list contains too few items, then an out-of-data error occurs and the program is terminated.

D. RESTORE STATEMENT

The RESTORE statement resets the pointer in the DATA list to the first DATA item. The RESTORE statement has the form:

```
N RESTORE
```

where N is a line number. For example:

```
1Ø DATA 1Ø,2Ø
2Ø READ A
3Ø READ B
```

assigns A the value 1Ø and B the value 2Ø. While

```
1Ø DATA 1Ø,2Ø
2Ø READ A
3Ø RESTORE
4Ø READ B
```

assigns both A and B the value 1Ø.

E. PRINT STATEMENT

The PRINT statement is used for output. The forms are:

```
N PRINT S
N PRINT#M,S
```

where N is a line number, S is a list of expressions, and M is a device number.

The second form is discussed under DEVICE SPECIFIED INPUT AND OUTPUT (Section H).

The following example of a PRINT statement:

```
1Ø LET A=3.15
2Ø LET B$="TOTAL IS"
3Ø PRINT B$;A
```

results in

```
TOTAL IS 3.15
```

appearing on the screen.

A question mark can be used instead of PRINT when entering a program. The following examples are equivalent:

```
1Ø ?"THE VALUE IS ";B
1Ø PRINT"THE VALUE IS ";B
```

The question mark only appears when the program is first typed, and is replaced by PRINT when the program is listed.

F. VERTICAL SPACING OF OUTPUT

Vertical spacing is accomplished by using the PRINT statement without an output list. This, in effect, prints a blank line. For example,

```
1Ø PRINT"LINE ONE"
2Ø PRINT
3Ø PRINT"LINE TWO"
```

results in the output:

```
LINE ONE

LINE TWO
```

Using a colon to allow multiple statements on a line and using ? for PRINT, three lines are skipped by this example:

```
1Ø ?:?:?
```

Because the question mark is replaced by PRINT when the program is listed, the programmer should be careful of overrunning the end of a line if he uses a lot of question marks for PRINTS.

The following example will skip 32 lines or clear the screen.

```
1Ø FOR I=1 to 32: PRINT:NEXT
```

G. HORIZONTAL SPACING OF OUTPUT

BASIC has several features that can be used to control horizontal spacing: zoned output, compressed output, the TAB function, the SPC function, and the POS function.

ZONED FORMAT

Each line of output is divided into 14-space zones. The use of commas in the output list specifies zoned format. For example,

```
1Ø PRINT"123456789Ø123456789Ø123456789Ø"
2Ø A=1.2:B=-5
3Ø PRINT A,B
4Ø PRINT A,,B
```

results in the output:

```
123456789Ø123456789Ø123456789Ø
 1.2           -5
 1.2                         -5
```

All values are left-justified in their zones. Positive numerical values have a space in the first position instead of a plus sign.

If a PRINT statement ends with a comma, the next PRINT statement outputs to the next zone instead of the next line. The statement:

```
1Ø PRINT A,B
```

is equivalent to the two statements:

```
1Ø PRINT A,
2Ø PRINT B
```

If a value will not fit into the 14 spaces allowed for a zone (for example, a long string), the next zone is skipped.

COMPRESSED FORMAT

The use of a semicolon in the output list of a PRINT statement specifies compressed format. String values are printed next to each other. Numeric values are printed with a trailing blank. Positive numeric values also have a leading blank instead of a plus sign. For example,

```
1Ø B=-40:A=3.5
2Ø C$="THE ANSWERS "
3Ø D$="ARE "
4Ø E$=" AND "
5Ø PRINT C$;D$;A;E$;B
```

results in the output:

```
THE ANSWERS ARE 3.5 AND -4Ø
```

If a PRINT statement ends with a semicolon, then the next statement outputs to the same line instead of the next line. For example,

```
1Ø PRINT A;
2Ø PRINT B
```

and

```
1Ø PRINT A;B
```

are equivalent.

TAB FUNCTION

The TAB function is used in the same way as the TAB key on a typewriter. The general form is

```
TAB(X)
```

where X is a arithmetic expression whose value is one less than the position where the next value is to be printed. An example:

```
1Ø PRINT A;TAB(3*X);B
```

Semicolons should be used with the TAB function. If followed by a comma, printing begins in the next zone. Note the effect of commas in lines 5Ø and 6Ø of this example:

```
1Ø PRINT"123456789Ø123456789Ø123456789Ø"
2Ø A=12.3:B=-5
3Ø A$="A"
4Ø PRINT A;TAB(8);B
5Ø PRINT A,TAB(8);B
6Ø PRINT A$,TAB(8),B
```

results in the output:

```
123456789Ø123456789Ø123456789Ø
 12.3    -5
 12.3            -5
A                        -5
```

SPC FUNCTION

The SPC function is used to print spaces in output. The general form is

```
SPC(X)
```

where X is a numerical expression whose value is the number of spaces to be printed. For example,

```
1Ø PRINT"123456789Ø123456789Ø"
2Ø PRINT"A";SPC(5);"B"
3Ø PRINT"A",SPC(5);"B"
```

results in the output:

```
123456789Ø123456789Ø
A     B
A                B
```

Note that the comma in line 3Ø produces spacing within the zone.

POS FUNCTION

A PRINT statement can print a sequence of up to 132 characters in length. The position function returns (as an integer between Ø and 132) the position in the sequence of the last character printed. Its form is

```
POS(X)
```

where X is a dummy argument. The value of X is ignored. For example,

```
1Ø PRINT"Ø1234";POS(X)
```

results in the output:

```
Ø1234 5
```

There may be a difference between the position of a character in the output sequence and its position on the screen. This is because a video screen displays either 32 or 64 characters per line; the output sequence, which can be as long as 132 characters, may extend over several lines. Thus, when POS(X)=64 the cursor is at the left margin of the screen.

H. DEVICE SPECIFIED INPUT AND OUTPUT

The disk BASICs, OS-65D and OS-65U, allow a device to be specified in PRINT, INPUT, and LIST statements. A device is specified by typing a pound sign followed by the device number. Some examples:

```
INPUT #8,D$
PRINT #4, "LINE PRINTER"
LIST #6
```

Input and output can be routed from or to various devices on the system including a terminal, modem or cassette at the serial port, video display, 43Ø board based UART, memory buffer, line printer, two disk buffers, 16 port serial board and a null device. The following table lists the device numbers:

65D INPUT DEVICES

1. Serial Port (ACIA)
2. Keyboard on 44Ø/54Ø Board
3. UART on 43Ø Board
4. Null
5. Memory
6. Disk Buffer 1
7. Disk Buffer 2
8. 55Ø Board Serial Port
9. Null

65D OUTPUT DEVICES

1. Serial Port (ACIA)
2. Video on 44Ø/54Ø Board
3. UART on 43Ø Board
4. Line printer
5. Memory
6. Disk Buffer 1
7. Disk Buffer 2
8. 55Ø Board Serial Port
9. Null

For example, to store a program on cassette that exists on disk, the user calls that program into memory and types LIST#1 or LIST#3 depending on which port his cassette interface is connected to. This

lists that program on that device. To output to a printer, the user types PRINT#4 and the output will be routed to the line printer. Memory output, device #5, is useful for various experimenter situations such as directly displaying information on the 54Ø video screen without scrolling.

Device #6 and device #7 are memory buffers for use with disk files.

Care must be taken not to route input or output to non-existent or turned-off peripheral devices since this will cause the computer system to "hang" and will require a reset which may destroy data in memory. For 65u device numbers, please refer to the manual.

I. MONEY-MODE OUTPUT

Money-mode output of numeric variables is available in OS-65U BASIC. Any numeric variable output in the money-mode is automatically truncated to two digits after the decimal point. For example, 3.149 would be output as 3.14. Rounding up can be accomplished by adding .ØØ5 to the number to be output. The money-mode also inherently provides left or right justification of the output in one of the 14-space output zones. A variable to be output in money-mode is preceded by either $R or $L, depending on whether it is to be left or right-justified in its field. Values are printed with a leading and following blank. When right-justified, values end two spaces inside the right edge of the field. For example:

```
1Ø X=1.429
2Ø Y=2.222
3Ø PRINT"123456789Ø123456789Ø123456789Ø"
4Ø PRINT $L,X
5Ø PRINT $R,Y
```

results in the output:

```
123456789Ø123456789Ø123456789Ø
 1.42
                2.22
```

BASIC turns on the money-mode when it encounters either $L or $R in and output list. The next numeric variable encountered in the output list is printed in money-mode, and then money-mode is turned off. String variables should not be used in money-mode because they do not turn it off.

CHAPTER 4

PROGRAM CONTROL

Normally, program execution proceeds sequentially. The order of execution can be altered by the control statements described in this chapter.

A. GOTO STATEMENT

The GOTO statement is an unconditional transfer statement. It has the form

```
N GOTO M
```

where N and M are line numbers. In OS-65U the directive M can also be a variable. Because blanks are ignored in BASIC, it can also be written GO TO.

When the GOTO is executed control transfers to line M, rather than to the next statement. For example,

```
1Ø GOTO 3Ø
2Ø PRINT "LINE 2Ø"
3Ø PRINT "LINE 3Ø"
```

results in the output

```
LINE 3Ø
```

When used in the immediate mode (see CHAPTER 5.B), GO TO M starts execution of the program in the workspace at line M.

B. IF . . . GOTO STATEMENT

The IF . . . GOTO statement is a conditional transfer statement. It has the general form:

```
N IF X GOTO M
```

where N and M are line numbers and X is a relational or arithmetic expression. In OS-65U the directive M can also be a variable. If the value of X is TRUE then the next statement executed is line number M; if X is FALSE control transfers to the line following N. For example,

```
1ØØ IF A < = 5 GOTO 1Ø
```

results in control passing to line 1Ø whenever A is less than or equal to 5 and to the line following 1ØØ whenever A is greater than 5.

If X is an arithmetic expression, the value of X is treated as FALSE whenever X is zero. If the value of X is nonzero then X is treated as TRUE. For example,

```
1ØØ IF SIN(A) GOTO 3ØØ
```

and

```
1ØØ IF SIN(A) < > Ø GOTO 3ØØ
```

are equivalent.

The difference between statements and lines (which can contain several statements) becomes very important when using the IF . . . GOTO statement. It should never be followed by a second statement on the same line; the second statement is never executed. Regardless of whether X is TRUE or FALSE, control always passes to a different *line*.

C. IF . . . THEN STATEMENT

The IF . . . THEN statement is a conditional transfer statement. It occurs in two forms:

```
N IF S THEN M
N IF S THEN R
```

where N and M are line numbers, S is a relational or arithmetic expression and R is a statement. If S is an arithmetic expression, the value of S is treated as FALSE whenever S is zero. If the value of S is nonzero then S is treated as TRUE.

The first form is equivalent to the IF . . . GOTO statement described above. For example,

```
1Ø IF Z < > Ø THEN 3ØØ
```

and

```
1Ø IF Z THEN 3ØØ
```

both transfer control to line 3ØØ whenever Z is non-zero.

With the second form, the statement R is executed whenever S is TRUE. If S is FALSE, R is ignored and control passes to the following line. These statements, for example,

```
1Ø INPUT "ENTER X";X
2Ø IF X > Ø THEN PRINT"X IS POSITIVE"
3Ø X<= Ø THEN PRINT"X IS NOT POSITIVE"
```

will cause one, but not both, of the phrases "X IS POSITIVE" or "X IS NOT POSITIVE" to be printed.

Multiple statements can appear in the place of statement R. If S is FALSE *all* of the statements following THEN are ignored. For example,

```
IF Z > Ø THEN PRINT"Z IS POSITIVE":GOTO 3ØØ
```

prints "Z IS POSITIVE" and transfers control to line 3ØØ when Z is positive. If Z is not positive, control passes to the next line in the program.

D. ON . . . GOTO STATEMENT

The ON . . . GOTO statement is a conditional transfer statement having the general form:

```
N ON S GOTO L
```

where N is a line number, S is an arithmetic expression, and L is a list of line numbers separated by commas. In OS-65U each line number in the list can be represented by a variable. The expression S is evaluated and truncated. Control then passes to the S-th line number in the list. For example,

```
2Ø ON Z GOTO 1ØØ,2ØØ,3ØØ
```

transfers control to line 100 if Z is 1, to line 2ØØ if Z is 2, and line 3ØØ if Z is 3. If Z is less than 1 or greater than 3 then the ON . . . GOTO statement is ignored and control passes to the following statement.

E. FOR-NEXT STATEMENTS

It is often desirable to repeat a segment of a program. Looping back over a portion of a program is usually accomplished in BASIC with a FOR-NEXT loop. The FOR and NEXT statements are used together to form the loop.

The FOR statement has the forms:

```
N FOR V = X TO Y
N FOR V = X TO Y STEP S
```

where N is a line number, V is a single numeric or integer variable, and X and Y are arithmetic expressions. The value of X is called the *initial value* assigned to the *index variable* V, the value of Y is the *limit* of V, and the value of S is the *increment*.

The NEXT statement has two forms:

```
N NEXT
N NEXT V
```

where N is a line number and V is the same index variable appearing in the FOR statement. The index variable is optional in the NEXT statement for a single loop, but should appear if loops are nested.

The FOR statement is the first statement in the program loop. The NEXT statement is the last statement in the loop. The collection of statements between the FOR statement and the NEXT statement is called the *body* of the loop and comprises the block of statements that are repeated.

The following actions take place with the first form of the FOR statement. When the FOR statement is executed, the index is assigned the initial value. Then the body of the loop is executed. Two actions, increment and check, take place when the NEXT statement is executed. First, the index variable is incremented by adding one to its value. Second, the value of the index variable is now compared to the limit. If the value of the index variable exceeds the limit, control transfers to the statement following NEXT. If the index variable is less than or equal to the limit, control transfers to the first statement in the body of the loop. For example,

```
1Ø FOR I = 1 TO 5
2Ø PRINT I
3Ø NEXT
```

causes the numbers from 1 to 5 to be printed in a column.

Because the index is not compared to the upper limit until the end of the loop, the body is always executed at least once.

The expressions X, Y and S are evaluated only once, when the FOR statement is executed. Thus the looping is unaffected if the variables comprising these expressions are assigned new values within the body of the loop. Looping may be affected if the value of the index variable is changed within the body of the loop.

Control may transfer out of the body of the loop. Transfer into the body with a statement other than a RETURN from a GOSUB, may lead to unexpected results.

STEP CLAUSE

The second form of the FOR statement contains the STEP clause. In the first form, the index variable is incremented by 1 on each pass through the loop. In the second form, the index variable is incremented by the value of S. If S is positive, then control passes out of the loop to the statement following NEXT when the index variable exceeds the limit. If the increment is negative, control passes out of the loop when the index variable is less than the limit. If the increment is zero, no check is made and the loop repeats indefinitely. Consider these examples:

STATEMENT	VALUES OF X
FOR X=1 TO 2 STEP .5	1,1.5,2
FOR X=1 TO 5 STEP 1Ø	1
FOR X=1Ø TO 1 STEP −1	1Ø,9,8,7,6,5,4,3,2,1
FOR X=1 to 1Ø STEP Ø	1,1,1, . . .

NESTED LOOPS

Loops may be nested. For example,

```
1Ø FOR I=1 TO 2
2Ø FOR J=1 TO 3
3Ø PRINT I,J
4Ø NEXT J
5Ø NEXT I
```

results in the output

```
1        1
1        2
1        3
2        1
2        2
2        3
```

Note that the inner loop is completed with each step of the outer loop.

Care must be taken to be sure the loops are properly nested and not overlapped as would occur if lines 4Ø and 5Ø above were reversed. Lines 4Ø and 5Ø can also be written in a shorthand form as " 4Ø NEXT J,I ".

Exiting in the middle of FOR-NEXT loops and then reusing the same loop variables as loop variables can create unexpected NEXT without FOR errors. Such errors can be avoided by using different loop variables.

F. STOP STATEMENT

Program execution is halted with a STOP statement. Its form is:

```
N STOP
```

where N is a line number. There may be more than one STOP statement in a program. When execution is halted a BREAK message with the line number is printed. For example,

```
1Ø PRINT "HERE"
2Ø STOP
```

results in the output:

```
HERE
BREAK IN 2Ø
```

A program that has been halted by a STOP statement can be restarted where it left off by the CONT command (see page 15).

G. END STATEMENT

The END statement is often used as the last statement in a program. Like the STOP statement, it terminates execution. It has the form:

```
N END
```

where N is a line number. The END statement is optional. If used, it need not be the last statement in the program, and there may be more than one END statement. In contrast to the STOP statement, no BREAK message is printed and the program cannot be restarted where it left off.

CHAPTER 5

PROGRAMMING

A. BASIC PROGRAMS

A BASIC program can be entered into the computer whenever the BASIC prompt

```
OK
```

appears on the screen. A BASIC program is composed of lines. Each line begins with a number (called the line number) followed by a list of BASIC statements separated by colons. For example,

```
1Ø A = 6Ø
2Ø B = A/2 : C = A/3
3Ø PRINT B, C
```

Any integer from 1 to 63999 can be used as a line number.

The lines of the program are typed one at a time. A line can hold 71 characters. A statement is not allowed to overlap two lines. (Since a line on the screen contains either 32 or 64 characters, a statement may overlap lines on the screen.)

After each line is typed the <RETURN> key is depressed.

After all of the program has been entered it is executed by typing the command

```
RUN
```

without a line number and followed by <RETURN>. Normally, the statements in a program are executed starting with the lowest line number and then to the next lowest, and so on. Statements on the same line are executed in order from left to right.

If the program above were run, the screen would appear as follows:

```
OK
1Ø A = 6Ø
2Ø B = A/2 : C = A/3
3Ø PRINT B, C
RUN
 3Ø                    2Ø

OK
```

The program is entered after the OK prompt. The RUN command is given and execution proceeds as follows:

Line 1Ø: The value of 6Ø is assigned to the variable A.

Line 2Ø: The value of A is divided by 2 and assigned to the variable B (now 3Ø). Then the value of A is divided by 3 and assigned to the variable C (now 2Ø).

Line 3Ø: The values of B and C are printed.

The output consists of the two numbers 3Ø and 2Ø. The OK prompt reappears after execution is completed. The computer is ready for another instruction.

The program is stored in a region of memory referred to as the workspace. The program will remain in the workspace until erased by the NEW command, replaced by a program loaded from tape or disk, or lost by unplugging the computer.

B. IMMEDIATE MODE

As each line is typed it is stored in a memory location referred to as a buffer. If the line begins with a line number it is added to the program in the workspace. If the line does not begin with a number it is executed immediately. This immediate execution feature is called the immediate mode or calculator mode of BASIC. Most BASIC statements can be used in the immediate mode. Examples:

```
PRINT SIN(.315)
```

and

```
FOR I=1TO1ØØ:PRINT I^3:NEXT
```

C. COMMANDS

A command is an instruction that is usually used in the immediate mode as apposed to a statement which is an instruction that usually appears within a program.

NEW COMMAND

If a program resides in the workspace any new lines that are entered with a line number will be added to it. In order to create new programs the workspace must be reset by the NEW command. It has the form

```
NEW
```

Because the NEW command returns from a BASIC program to the immediate mode, it is not as useful as other commands when used within a program. But the NEW command can be used within a program for read protection by using NEW in place of an END commands.

RUN COMMAND

The RUN command has the following forms:

```
RUN
RUN M
RUN"N
```

where M is a line number and N is the name of a program stored on disk or a track number. The first form starts execution of the program in the workspace at the lowest line number. The second form starts execution of the program in the workspace at the line numbered M. The third form causes the program named N to be loaded from disk and executed starting at the lowest line number. If N is a number then the program on track number N will be loaded and executed starting at the lowest line number. Examples:

```
RUN 135
RUN"ACCOUNT
RUN"23
```

All of the forms of the RUN command can be used within a program. When used in a program, the first form simply restarts the program; the second form acts in a manner similar to the GOTO statement except that the variable table is cleared; the third form is the most useful. The third form can be used to "chain" programs so that they are executed one after the other. Programs are chained by having the last statement in each program be a RUN command giving the name of the next program to be executed.

LIST COMMAND

The LIST command causes a segment of the program in the workspace to be printed, usually on the screen. It has the forms:

```
1) LIST
2) LIST F
3) LIST#D
4) LIST#D,F
```

The first form lists the entire program on the screen.

The letter F represents one of the following forms which are illustrated by example using the second form above:

LIST 1Ø	lists only line 1Ø
LIST −1Ø	lists from the beginning to line 1Ø
LIST 1Ø−	list from line 1Ø to the end
LIST 1Ø−2Ø	lists from line 1Ø to line 2Ø

The letter D is an output device number. If a hard-copy printer were device number 1, then the entire program would be printed by

```
LIST#1
```

and lines 1Ø through 2Ø would be printed by

```
LIST#1, 1Ø−2Ø
```

Device numbers are discussed in Chapter 3 under heading DEVICE SPECIFIED INPUT AND OUTPUT (Section H).

It is often desirable to "page through" a program by stopping and restarting the LIST command. In ROM BASIC listing can be halted by depressing <CTRL/C>. Listing can be restarted on line number N by LIST N−. This procedure can also be used in the disk BASICs, but there is an easier procedure: listing can be halted by <CTRL/S> and restarted where it left off by <CTRL/Q>.

The output of LIST to the screen can be toggled in OS-65U BASIC by <CTRL/O>. This differs from the features described above in that the listing continues; it simply doesn't appear on the screen.

Since LIST returns from a BASIC program to the immediate mode it is not very useful as a program statement.

CONT COMMAND

The continue command has the following form:

```
CONT
```

The continue command can only be used in the immediate mode.

When a program is halted by a <CTRL/C> or STOP statement a pointer is set in the program at the point of interruption. The program can be restarted where it left off by the CONT command. Restarting the program need not take place immediately. For example, the immediate mode can be used for LISTing and PRINTing without disturbing the pointer. This provides the programmer with a very useful debugging procedure:

1) Place STOP statements at convenient points within the program.
2) RUN the program.
3) When a STOP is executed a BREAK message with the line number will be printed. The program segment that was just executed can be listed using the LIST command, and the present values of variables can be determined by using the PRINT statement in the immediate mode. For example,

```
LIST 1ØØ-2ØØ
PRINT A,B,C$
```

NOTE: If a new line of text is added to the program, these pointers are cleared and a continue error will be given if CONT is used.

D. INTERRUPTING EXECUTION

A program that is running can be halted by depressing <CTRL/C>. It can be restarted where it left off with the CONT command.

The keyboard is continually checked during execution to see if a <CTRL/C> has been depressed. This feature can be disabled by one of the following POKE statements; it must be disabled when a program polls the keyboard.

A <BREAK>, or on some systems <RESET>, will also halt program execution. In ROM BASIC a <BREAK> followed by a warm start <W> will return to BASIC with the program in the workspace intact. In 65D and 65U-BASIC the program must be reloaded from disk following a <BREAK> or <RESET>.

The LIST command can also be halted; see LIST COMMAND above.

E. EDITING A PROGRAM

Corrections can be made in a line as it is being typed. A <SHIFT/O> will backspace and delete the last character. Multiple deletions can be made by repeating the <SHIFT/O>. In ROM BASIC the characters will still appear on the screen with cursor marks after them. The line will appear in corrected form after a LIST command.

A line can be deleted as it is being typed by entering a "commercial at" symbol, @. On polled keyboards @ is entered by <SHIFT/P>.

If a new line is entered with the same line number as a previous line it will replace the previous version. Thus a line can be removed from a program by simply typing its line number followed by <RETURN>.

F. DOCUMENTATION

Remarks can be placed in BASIC programs with the REM statement. These comments can often be very useful to a person reading the program. They are ignored by the computer when the program is executed.

REM STATEMENT

The remark statement has the form:

```
N REM R
```

where N is a line number and R is a remark. For example:

```
1ØØ REM SUBROUTINE TO FIND RATIO
25Ø X=T/D:REM X IS THE RATIO
```

As shown by the above examples a remark may appear as the only statement on a line or follow other statements. However, another statement should not follow a REM statement on the same line. It would not be executed; everything after REM is ignored on execution.

DISABLE <CTRL/C>	ENABLE <CTRL/C>	BASIC USED
POKE 53Ø,1	POKE 53Ø,Ø	ROM
POKE 2Ø73,96	POKE 2Ø73,173	OS-65D
POKE 2Ø73,96	POKE 2Ø73,76	OS-65D

G. SAVING SPACE

Writing a procedure so that it will fit into the available workspace can be a significant programming difficulty. BASIC provides some features that can be of help.

FRE FUNCTION

The amount of workspace available to the programmer can be determined by the free function. The free function returns the number of bytes of memory in the workspace that are unused. It has the form:

```
FRE(X)
```

where X is a dummy variable. A programmer who wishes to expand an existing program should run the program before using the free function; additional memory is required during execution for the variable table. After the program is executed, the following line can be entered in the immediate mode:

```
PRINT FRE(X)
```

If more than 32K of memory is available the FRE function returns a value that has cycled negative. That is, values increase in the order 1,2,. . ., 32767,−32768,−32767,. . . . When FRE(X) is negative the number of available bytes can be determined by

```
PRINT 65536+FRE(X)
```

If the FRE function causes the computer to "hang," it should be preceded by the CLEAR statement. Since the FRE function may cause the computer to "hang," the programmer should save the program in the workspace on tape or disk before using FRE.

CLEAR STATEMENT

As variables are encountered in a program they are put in a variable table along with their values. The clear statement clears the variable table and RESTOREs the DATA pointer. It has the form:

```
N CLEAR
```

where N is a line number. The CLEAR statement can be used to reduce the amount of memory that a program requires by removing variables that are no longer needed.

The following example illustrates the effect of the CLEAR statement. The FRE function is used to determine the amount of workspace remaining unused.

```
 1Ø PRINT FRE(X)
 2Ø A=2:A$="X"
 3Ø PRINT FRE(X)
 4Ø PRINT A,A$
 5Ø CLEAR
 6Ø PRINT FRE(X)
 7Ø PRINT A,A$
OK
RUN
 31923
 31911
 2             X
 31923
 Ø
```

SUGGESTIONS

The first place a programmer can look for additional space is the overall design of the program. After that, some simple fixes can be tried. For example:

1) Use subroutines for repeated code and functions for repeated calculations.
2) Remove blanks:
   `1Ø FOR I = 1 TO 1Ø`
   and
   `1Ø FORI=1TO1Ø`
   are equivalent.
3) Remove REM statements.
4) Remove line numbers by putting more statements per line.
5) Reuse variable names.
6) Use smaller names such as A for A1.
7) Put variables in arrays; an array of 1Ø elements uses less space than 1Ø different variable names.
8) Integer arrays use less space in memory than real arrays.

CHAPTER 6

ARRAYS

Large quantities of data can be handled in BASIC by organizing the data into arrays. Arrays can be one or multi-dimensional. The elements of an array are subscripted variables.

A. SUBSCRIPTED VARIABLES

Variables may be simple or subscripted. Subscripted variables have the form

```
N(L)
```

where N is an arithmetic, integer, or string variable; and L is a list of arithmetic expressions, called subscripts, separated by commas. Examples:

```
N1(2)
V$(5.6,4*X)
RA%(S,T,W)
```

The arithmetic expressions used as subscripts are evaluated and then truncated to integer values. Subscripts can have values between Ø and 2Ø, inclusive. Larger subscript values are allowed if the array is dimensioned in a DIM statement.

B. DIM STATEMENT

The dimension statement has the form

```
N DIM L
```

where N is a line number and L is a list of subscripted variable names. For example,

```
1Ø DIM A(2Ø),B$(1,2)
2Ø DIM X1(N*2)
```

The array A is a one-dimensional arithmetic array having twenty-one elements: A(Ø), . . . , A(2Ø). The array B is a two-dimensional string array having the six elements: B$(Ø,Ø), B$(Ø,1), B$(Ø,2), B$(1,Ø), B$(1,1), and B$(1,2).

Arrays can have variable subscripts. For example,

```
1Ø INPUT"WHAT IS THE DIMENSION OF M";N
2Ø DIM M(N)
```

Dimension statements are usually placed together at the beginning of the program. However, dimension statements can occur anywhere in a program. Space is allocated as they are encountered. The dimension statement must be executed before the array is used. A double dimension error occurs if an array is encountered in a DIM statement after one of the elements of the array has been encountered. A double dimension error also occurs if an array is encountered in more than one DIM statement.

CHAPTER 7

FUNCTIONS

A. MATHEMATICAL FUNCTIONS

The following mathematical functions are supplied in each version of BASIC. In general, the arguments of these functions may be any arithmetic expression. Exceptions are noted in the discussion of each function.

ABS FUNCTION

The absolute value function returns the absolute value of its argument: ABS(X) is equal to X if X is greater than or equal to zero and ABS(X) is equal to −X if X is less than zero. For example, ABS(−7.5) is 7.5.

EXP FUNCTION

The exponential function returns e=2.71828. . . raised to the power of its argument. For example, EXP(1) is e. The argument of EXP must be less than 88.Ø296919.

INT FUNCTION

The integer function returns the greatest integer less than or equal to its argument. For example, INT(6.6) is 6 and INT(−3.2) is −4.

LOG FUNCTION

The logarithm function returns the natural logarithm (log to the base e) of its argument. The argument must be positive. The log to another base, say B, of X is LOG(X)/LOG(B).

RND FUNCTION

The random number generating function returns a number between Ø and 1. This function is usually used to generate a sequence of pseudo-random values. For example, in ROM BASIC this program:

```
 5 X = RND(−1)
1Ø FOR I=1 TO 5
2Ø PRINT RND(1);
3Ø NEXT
```

results in the output:

```
163997    .56961    .865247    .3236Ø2    .412642
```

If the argument is positive, it is a dummy argument. That is, its value is not important; RND only checks to see if it is positive. As long as the argument remains positive, RND will generate the next number in the sequence using the last value returned. The random number sequences are periodic. The example above repeats after 1861 calls to RND.

If the argument is negative, RND will start a new sequence with a new period based on the value of the argument. Thus negative arguments serve as seeds. The same sequence is generated if the same negative seed is used.

If the argument is zero, RND will return the previous value again.

If the programmer wishes to have a program generate a different random number sequence each time the program is run, he should devise a procedure for choosing the seeds. Such a procedure might be based on PEEKing various memory locations.

A random number N between two numbers A and B (A<N<B) can be obtained by N = A+RND(X)*(B−A).

SGN FUNCTION

The sign function returns the sign of the argument. Plus one is returned for positive arguments, minus one for negative arguments, and zero is returned if the argument is zero.

SQR FUNCTION

The square root function returns the square root of its argument. For example, SQR(4) is 2. The argument must be positive.

TRIGONOMETRIC FUNCTIONS

The trigonometric functions require their arguments to be in radians. To convert degrees to radians: radians = .Ø174532925 * degrees.

The sine, cosine, tangent, and arctangent functions are supplied by BASIC. They are called by the following forms: SIN(X), COS(X), TAN(X), and ATN(X); where the argument is an arithmetic expression. The value of the argument of ATN(X) must be between −1 and 1.

The following functions, while not intrinsic to BASIC, can be calculated using the existing BASIC functions as follows:

```
SEC(X) = 1/COS(X)
CSC(X) = 1/SIN(X)
COT(X) = 1/TAN(X)
ARCSIN(X) =ATN(X/SQR(-X*X+1))
ARCCOS(X) = -ATN(X/SQR( -X*X+1)) +1.57Ø8
ARCSEC(X) = ATN(SQR(X*X-1))
ARCCSC(X) = ATN(1/SQR(X*X-1)) +(SGN(X)-1)*1.57Ø8
ARCCOT(X) = ATN(1/X)
SINH(X) = (EXP(X)-EXP(-X))/2
COSH(X) = (EXP(X)+EXP(-X))/2
TANH(X) = EXP(-X)/(EXP(X)+EXP (-X)) *(-2)+1
SECH(X) = 2/(EXP(X)+EXP(-X))
CSCH(X) = 2/(EXP(X)-EXP(-X))
COTH(X) = EXP(-X)/(EXP(X)-EXP (-X))*2+1
ARCSINH(X) = LOG(X+SQR(X*X+1))
ARCCOSH(X) = LOG(X+SQR(X*X-1))
ARCTANH(X) = LOG((1+X)/(1-X))/2
ARCSECH(X) = LOG((SQR(-X*X+1) +1)/X)
ARCCSCH(X) = LOG((SGN(X)*SQR(X*X+ 1)+1/)X
ARCCOTH(X) = LOG((X+1)/(X-1))/2
```

B. STRING FUNCTIONS

A string function is either a function whose argument is a string or a function which returns a string. String functions may return either numeric values or strings. Those that return strings have names ending with a dollar sign.

ASC FUNCTION

The ASCII function returns the ASCII value in decimal of the first character in the argument. It has the form

```
ASC(X$)
```

where X$ is a string expression. For example, ASC("BIG") is 66.

CHR$ FUNCTION

The character function returns a one-character string. The character returned is the one whose decimal ASCII value is the argument. It has the form

```
CHR$(X)
```

where X is an arithmetic expression whose value is between Ø and 255. The character function is essentially the opposite of the ASC function. For example, CHR$(66) IS "B".

LEFT$ FUNCTION

The left function returns a left-most substring of a string. It has the form

```
LEFT$(X$,Y)
```

where X$ is a string expression and Y is a positive arithmetic expression. The Y left-most characters of X$ are returned. For example, LEFT$("123456",3) is "123". If Y exceeds the length of the string, the string is returned.

LEN FUNCTION

The length function returns the length of a string. It has the form

```
LEN(X$)
```

where X$ is a string expression. For example, LEN("OUT") is 3.

MID$ FUNCTION

The middle function returns a middle substring of a string. It has the two forms

```
MID$(X$,Y)
MID$(X$,Y,Z)
```

where X$ is a string expression, Y is a positive arithmetic expression and Z is a nonnegative arithmetic expression. The first form returns the substring of X$ starting in the Y-th position to the end of the string. For example, MID$("123456",3) is "3456". The second form returns a substring of length Z starting in the Y-th position. For example, MID$("12345",3,2) is "34". If Y exceeds the length of the string, the string of length zero, "", is returned. If Z goes past the end of the string, the substring starting in the Y-th position to the end of the string is returned.

RIGHT$ FUNCTION

The right function returns a right-most substring of a string. It has the form

```
RIGHT$(X$,Y)
```

where X$ is a string expression and Y is a positive arithmetic expression. The right-most Y characters of X$ are returned. For example, RIGHT$("VALUE",3) is "LUE". If Y exceeds the length of the string the string is returned.

STR$ FUNCTION

The string function returns the value of the argument as a string. It has the form

```
STR$(X)
```

where X is an arithmetic expression. For example, STR$(12.3) is " 12.3", a string of length 4. For positive numbers, a leading blank instead of a plus sign is returned. The results are the same as when X is PRINTed; except that no trailing blank is included in the string. Some forms are converted; for example,

```
1Ø PRINT"12345678Ø1234567Ø"
2Ø A(1)=15.1
3Ø A(2)=-25
4Ø A(3)=12.ØE+2
5Ø A(4)=1ØØØØØØØØØØ
6Ø FOR I = 1 TO 4
7Ø A$=STR$(A(I))
8Ø PRINT A$, LEN(A$) : NEXT
```

results in the output

```
12345678Ø1234567Ø
 15.1            5
-25              3
 12ØØ            5
 1E+11           6
```

VAL FUNCTION

The value function returns the numeric value of a string. It is the opposite of the STR$ function. Its form is

VAL(X$)

where X$ is a string expression representing a number. For example, VAL("Ø.ØØ35") is 3.5E−Ø3. If X$ does not represent a number the value Ø is returned.

CHAPTER 8

SUBPROGRAMS

A calculation that needs to appear more than once in a program can be written as a subprogram. The subprogram can be called each time it is needed; thus avoiding the necessity of rewriting the calculation. There are two types of subprograms in BASIC: statement functions and subroutines. A statement function consists of a one-statement calculation. A subroutine can be a self-contained program.

A. STATEMENT FUNCTIONS

In addition to the functions supplied by the system, the user can create functions called statement functions. Statement functions are defined by the DEF statement.

DEF STATEMENT

The define function statement has the form

```
N DEF FNX(A) = E
```

where N is a line number, X and A are simple numeric variables, and E is an arithmetic expression. The name of the function consists of the letters FN followed by a variable name. The variable A is called the dummy variable. The expression E may reference other functions; including those defined by DEF statements. The use of the variables X and A in the function does not affect their use elsewhere in the program. The define function statement cannot be used in the immediate mode.

Consider the following program:

```
1Ø DEF FNS(P) = P + P ^ 2
2Ø X = 2
3Ø PRINT FNS(X+1)
```

The output is

```
12
```

The function FNS is defined in line 1Ø to be P plus the square of P. In line 3Ø, the programmer has replaced the dummy argument P by an arithmetic expression, X + 1. At this point in the program the value of X + 1 is 3, so 3 is substituted for each occurence of P in the defining expression. The result, 3 + 3 ^ 2 = 12, is assigned to FNS(X+1).

B. SUBROUTINES

A subroutine consists of a program segment ending with a RETURN statement. A GOSUB statement calls the subroutine by transferring control to the first line of the subroutine. When the return statement at the end of the subroutine is encountered, control returns to the statement following the GOSUB statement.

GOSUB AND RETURN STATEMENTS

The form of the GOSUB statement is

```
N GOSUB M
```

where N and M are line numbers. In 65U the directive M can also be a variable.) Control is transferred to line number M. The form of the RETURN statement is

```
N RETURN
```

where N is a line number. For example,

```
1Ø PRINT "START"
2Ø GOSUB 5Ø
3Ø PRINT "OUT OF SUBROUTINE"
4Ø END
5Ø PRINT "IN SUBROUTINE"
6Ø RETURN
```

results in the output

```
START
IN SUBROUTINE
OUT OF SUBROUTINE
```

Subroutines can call other subroutines including themselves. Subroutines may have logical branches each of which ends in a RETURN statement.

It is convenient to picture the transfer of control as follows: As the GOSUBs are encountered they are stacked one on the other; when a RETURN statement is encountered one of the GOSUBs is peeled off the top of the stack. Control then passes to the next statement following the GOSUB that was on top of the stack.

If a RETURN statement is encountered with no GOSUB on the stack, a RETURN-without-GOSUB error occurs. For this reason, subroutines are placed after a STOP or END statement denoting the end of the main logical sequence in the program.

ON . . . GOSUB STATEMENT

The ON . . . GOSUB statement is a conditional transfer statement similar to the ON . . . GOTO statement. It has the form

```
N ON S GOSUB L
```

where N is a line number, S is an arithmetic expression, and L is a list of line numbers separated by commas. The expression S is evaluated and truncated. Control then passes to the S-th line number in the list L. When a RETURN is encountered, control returns to the statement following the ON . . . GOSUB statement. For example,

```
20 ON Z GOSUB 1ØØ,2ØØ,3ØØ
```

transfers control to statement 1ØØ if Z = 1, to statement 2ØØ if Z = 2, and to statement 3ØØ if Z = 3. If Z is less than 1 or greater than 3 then the ON . . . GOSUB statement is ignored and control passes to the following statement. (In 65U each line number can also be represented by a variable.)

CHAPTER 9

DIRECT MEMORY CONTROL

The following features of BASIC can be very useful to the experienced programmer. Care must be exercised with these statements and functions because they manipulate the memory of the computer directly. An improper operation with any of these commands can cause a system crash, wiping out BASIC and the user's programs.

The function of each memory location varies with the computer's configuration. The programmer should consult his operating systems manual for a "memory map" and a listing of the most useful parameters used by PEEK, POKE, and WAIT.

A. POKE STATEMENT

The POKE statement stores a value into a memory location. It has the form

```
N POKE I, J
```

where N is a line number, and I and J are arithmetic expressions whose values are integers. The value of I is a memory location expressed in decimal and the value of J is placed in location I. The value of J must be between Ø and 255 inclusive. For example,

```
1Ø FOR I = 14822 TO 14828
2Ø POKE I, A(I)
3Ø NEXT
```

stores the array A in the memory locations 14822 to 14828.

The expression J cannot contain the PEEK function. No error results; the value is not POKED.

A value cannot be POKED into a ROM memory location.

B. PEEK FUNCTION

The PEEK function reads a location memory. It functions as the opposite of the POKE statement. It has the form

```
PEEK(I)
```

where I is a memory location or I/O location expressed in decimal. For example,

```
1Ø FOR I = 14822 TO 14828
2Ø A(I) = PEEK(I)
3Ø NEXT
```

assigns the values in memory locations 14822 to 14828 to the array A. A memory location can store one byte; the value returned by the PEEK function is therefore an integer between Ø and 255.

If a write-only memory location is PEEKed the value returned may not be the actual value in the location.

C. WAIT FUNCTION

The WAIT function halts program execution, or causes the program to "wait", until a particular bit in memory is set or reset. It has the forms

```
WAIT I, J
WAIT I, J, K
```

where I is a memory location expressed in decimal, and J and K are integers between Ø and 255. The WAIT function in the first form reads the status of memory location I then ANDs the result (see THE BIT OPERATORS AND, OR, AND NOT) with J until a nonzero result is obtained.

The bit operator OR compares two binary numbers bit-by-bit and sets a bit in the result to 1 if one or both of the corresponding bits in the two numbers is 1. The logical operation "exclusive" OR is similar, but sets a bit to 1 in the result if exactly one of the corresponding bits in the two numbers is 1.

The WAIT function in the second form reads the status of memory location I, exclusive ORs that value with K, and then ANDs that result with J until a nonzero result is obtained.

The WAIT function is used for fast service of input status flags. For example,

```
WAIT X, 1
```

will cause the execution of a BASIC program to halt and then continue when bit zero of memory location X goes low.

The WAIT function is not available on some specialized disks. To determine if WAIT is available on a disk, boot up the disk and PEEK four consecutive memory locations starting at 713 in OS-65D (starting at 9Ø29 in OS-65U). If WAIT is available, the values returned will be 87, 65, 73, and 212. These numbers are the ASCII values of W, A, I, and T plus 128.

CHAPTER 10

TAPE INPUT AND OUTPUT

This chapter discusses the commands provided in ROM BASIC for input and output to cassette tape.

A. LOAD COMMAND IN ROM BASIC

The load command switches input from the keyboard to serial input port 1. It has the form

LOAD

The LOAD command can be executed in the immediate mode or as part of a stored program.

When the BASIC interpreter encounters a LOAD command, it switches input from the keyboard to serial input port 1. Input continues from this port until the user depresses the space bar on the terminal or a program modifies a flag in memory by the statement POKE 515,Ø (POKE 515,255 also turns on LOAD). Serial port 1 is normally connected to an audio cassette interface.

To LOAD programs which are stored on tape into the computer, proceed as follows:

1. Rewind the tape.
2. Cold start the machine or type NEW <RETURN>.
3. Type LOAD but not <RETURN>.
4. Start the tape in play-back mode.
5. As soon as the leader passes over the tape head, depress <RETURN>.
6. Upon completion of a LOAD, turn off the tape recorder type <SPACE BAR> and then <RETURN>.

B. SAVE COMMAND IN ROM BASIC

The SAVE command causes output to be routed to both the video screen and serial port 1. It has the form

SAVE

The SAVE command can be executed in the immediate mode or as part of a stored program.

When the BASIC interpreter encounters the SAVE command, it routes output to both the video screen and serial port 1. This mode of operation continues until a LOAD command is encountered which automatically clears the SAVE condition. The serial port is normally connected to the audio cassette output interface so that the SAVE command can normally be used for saving programs and storing data in cassette files.

To SAVE a program which is in the workspace on cassette tape, proceed as follows:

1. Rewind the tape.
2. Type SAVE <RETURN>. It is optional, but good practice, to now type NULL 8.
3. Type LIST but not <RETURN>.
4. Start the recorder in the record mode.
5. As soon as the leader passes over the tape head, depress <RETURN>.
6. When the listing is complete, turn off the tape recorder and optionally type LOAD <RETURN> <SPACE BAR> <RETURN> to revert to normal operation.

C. NULL COMMAND

The NULL command inserts zeros at the beginning of each line as it is stored on tape. It has the form

NULL M

where M is an integer from Ø to 8. The integer M is the number of zeros to be inserted at the beginning of each line. When the NULL command is used, the programmer should avoid extra long lines in the program to be SAVEd. Lines of 64 characters or less are generally safe.

CHAPTER 11

DISK INPUT AND OUTPUT

The disk BASICs, OS-65D and OS-65U, contain complete disk operating systems. These systems differ in their commands and procedures. Below is a brief description of the data and program file commands available in each system. The programmer should consult his operating systems manual for a more detailed discussion.

A. OS-65D COMMANDS

This system permits accessing BASIC programs, assembler source files and data files by name.

I/O DEVICES #6 & #7

These devices are the CHANNELS under which DISK file I/O operates. Device #6 permits random (in any order) access file operation while devices #6 and #7 may be used in conjunction with sequential (one after the other) files.

OPEN COMMAND

The open command permits the OPENing of a data file for sequential or random access. The format for the OPEN command is

```
DISK OPEN, device number, "filename"
```

CLOSE COMMAND

The CLOSE command closes a file to permit the opening of another file. Typing

```
DISK CLOSE,X
```

where X is a specific device number, will close that device.

DISK! COMMAND

The DISK! command permits the programmer to send commands to the OS-65D disk operating system. This command may be entered in the immediate mode or used in a program. Examples include:

```
DISK!"PUT PROG1"
DISK!"LOAD PROG1"
DISK!"IO ,03"
```

GET COMMAND

The GET command brings a specific record from the disk to the workspace and sets the INPUT and PRINT pointers (device #6) to the beginning of the record. Typing

```
DISK GET,N
```

will find record number N, and set the I/O pointers.

PUT COMMAND

The PUT command places a specific record on the disk. Typing

```
DISK PUT
```

will cause the contents of the disk buffer (#6) to be placed in the disk file.

EXIT COMMAND

The EXIT command will transfer control to OS-65D Disk Operating System. Just type

```
EXIT
```

to enter the OS-65D DOS.

B. OS-65U COMMANDS

OPEN COMMAND

The OPEN command is used to OPEN a data file for access. The command has two possible formats:

```
OPEN "filename","password",channel number
     and
OPEN "filename",channel number
```

The channel number must be an integer between one and eight. After OPENing a file, the file is referred to by the channel it was opened under rather than by its name. If a file is opened with the correct password, the system may read or write to the file regardless of that file's access rights. If a file is opened without the correct password, the system may only access the file in the manner defined by its access rights (e.g. read only, write only, none, etc.).

CLOSE COMMAND

The CLOSE command permits a file to be closed. Closing a file frees the channel the file was opened under. Simply typing

```
CLOSE
```

closes all the channels currently open. Typing

```
CLOSE X
```

where X is a specific channel number, closes only that channel.

INDEX COMMAND

The INDEX is a reserved system variable. This command takes two forms the first being

```
INDEX (channel number)
```

An example of the program usage would be

```
X=INDEX(1)
```

After the execution of this statement, X would equal the value of the INDEX for channel 1. This INDEX value is simply a relative pointer into the file opened under a particular channel. When a file is opened, the INDEX of the channel the file was opened under is set to zero.

The second form of the INDEX function is

```
INDEX (channel number) = expression
```

This form permits "bumping" the INDEX to point anywhere within the data file. Note the power of the INDEX function in permitting the user to define the file formats to be whatever is desired.

PRINT % COMMAND

This command has the form:

```
PRINT% channel number, variable expression
```

where the variable expression may be string or numeric. This statement directs the variable to be printed into the data file opened under the channel *"channel number"*. The variable(s) will be printed into the file starting at the current position of the INDEX of that channel.

INPUT % COMMAND

This command has the form:

```
INPUT% channel number, variable(s)
```

This command INPUTS data from the file opened under *"channel number"* and assigns the data to the variables(s) that appear in the INPUT% statement. The data is INPUT form the file starting at the current position of the INDEX. The INPUT% statement terminates upon the receipt of a carriage return.

FIND COMMAND

The FIND command has the form

```
FIND "string expression," channel number
```

The FIND command executes a high speed search for the string expression in the file opened under "channel number". The search starts from the current position of the INDEX for that channel. If the string is found, the INDEX for that channel points at the string in the file. If the string is not found, the INDEX for that channel is set to 1,000,000,000. Note

that the FIND command may be used to find subsets, e.g.,

```
FIND "ABCD",1
```

would find the subset "ABCD" in the string "ABCDEFG". The FIND command also permits "don't care" characters within the string it is searching for. That is,

```
FIND "AB&HI",1
```

would find "ABCHI" and "ABZHI".

FLAG COMMAND

The FLAG command permits the user to tailor his system toward a specific application. The FLAG command has the form:

```
FLAG N
```

where N is the flag number of the option desired. The options and their flag numbers are presented in the OS-65U operating system manual.

DEV COMMAND

The device command specifies which device is to be the current mass storage unit. It has the form:

```
DEV U
```

where U is the unit. Unit specifications are A, B,C or D for floppy disks, E, F, G or H for hard disks and K through Z for machines connected in a network.

CHAPTER 12

INDIRECT FILES

A BASIC program can be moved in memory to a location other than the workspace. Memory holding a program in this manner is called an indirect file. An indirect file can be used to merge two programs and to transfer programs between disks having different systems such as OS-65D and OS-65U. A procedure for performing each of these operations is given below. The specific commands necessary to perform these operations vary according to the system used, so the procedures are discussed separately from the commands.

A. MERGING PROGRAMS

If a program needs a lot of corrections or the addition of substantial amounts of code, such as subroutines, a separate file can be created containing the additions. This file can then be merged into the program. Three procedures for merging files are discussed in this section. The first can be used with BASIC-in-ROM; the second can be used with OS-65D; and the third can be used with OS-65D and OS-65U.

The following procedure can be used to merge two BASIC-in-ROM programs.

(1) Store PROG1 onto a cassette,
(2) Load PROG2 into the workspace, and
(3) Load PROG1 into the workspace without entering NEW.

If each of the programs has a line with the same number the line in PROG1 will be the one that appears in the merged program.

The following procedure can be used to merge two programs in OS-65D. Start with both programs, say PROG1 and PROG2, stored on a diskette.

1) Load PROG1 into the workspace:

```
DISK!"LOAD PROG1"
```

Enter

```
EXIT
```

The number of tracks necessary to hold PROG1 will be displayed, say N tracks. Return to BASIC by entering

```
RE BA
```

2) Run the disk utility CREATE and create a file PROG3 to hold the merged programs. If PROG2 already has enough space the merged program can be stored as PROG2.

3) The number, N, of tracks necessary to store PROG1 was determined in step 1). Run CREATE again and make a file called "DATA" with 4 times N tracks for a 5 inch diskette and 6 times N tracks for an 8 inch diskette. Answer NO to the query about pages per track. Specify 4 pages per track.

4) Load PROG1 into the workspace:

```
DISK!"LOAD PROG1"
```

5) Enter the following POKEs to create a 4 page buffer and to disable the scrolling of the screen (the screen will hold the buffer).

```
POKE 8998,Ø
POKE 8999,2Ø8
POKE 9ØØØ,Ø
POKE 9ØØ1,212
POKE 977Ø,Ø
```

6) Enter on *a single line:*

```
DISK OPEN,6,"DATA":DISK!"IO, 22":LIST
```

A listing of the workspace will appear on the screen while PROG1 is being stored in the file DATA.

7) When the listing is finished, reset the I/O pointers and close the file by entering:

```
DISK!:"IO 02,02":DISK CLOSE,6
```

8) Load PROG2 into the workspace by entering:

```
DISK!"LOAD PROG2"
```

9) Reopen the file DATA and merge PROG1 into PROG2 by entering:

```
DISK OPEN,6,"DATA":DISK!"IO 2Ø"
```

1Ø) Reset the I/O pointers, close the file, and enable scrolling by entering:

```
DISK!"IO Ø2,Ø2":DISK CLOSE, 6
POKE 977Ø,64
```

11) Store the merged file by entering:

```
DISK!"PUT PROG3"
```

12) Clean house by rebooting the system.

If each of the programs has a line with the same number, the line in PROG1 will be the one that appears in the merged program.

To merge two BASIC programs using indirect files:

1) determine the starting page number N of the indirect file,
2) load one program into the workspace,
3) move this program to the indirect file,
4) load the second program into the workspace,
5) move the first program back from the indirect file to the workspace.

If each of the programs has a line with the same number the line in the first program will be the one that appears in the merged program.

B. MOVING PROGRAMS BETWEEN INCOMPATIBLE DISKS

To transfer a program between incompatible disks:

1) determine the starting page number N of the indirect file,
2) boot up BASIC and load the program into the workspace,
3) move the program to the indirect file using the POKEs for the system on this disk,
4) boot up BASIC on the other disk; clear the workspace with NEW,
5) move the program from the indirect file to the workspace using the POKEs for the system on this new disk,
6) PUT the program on the new disk.

C. STARTING PAGE NUMBER OF INDIRECT FILE

The starting page number N of an indirect file can usually be set at 128 in OS-65D and 144 in OS-65U. If the program is quite large these values may not work. The indirect file must fit into memory above the program in the workspace. A value for N is given by:

```
N = highest page in memory—pages unused
in memory
```

the highest page in memory can be obtained by:

```
?PEEK(133)
```

and the number of pages unused in memory can be obtained by

```
?INT(FRE(X)/256)      ,or
```

if FRE(X) is negative, by:

```
?INT(65536+FRE(X))/256)
```

The starting page of the workspace is approximately

page 5Ø (317E) for OS-65D on an 8 inch disk,
page 51 (327E) for OS-65D on a 5 inch disk, and
page 96 (6ØØØ) for OS-65U.

The number of pages used by the program is:

```
highest page—starting page—pages left.
```

If the number of pages used exceeds the number of pages left there is not enough memory available to put this program in an indirect file.

D. FROM WORKSPACE TO INDIRECT FILE

To move a program from the workspace to an indirect file:

1) enable the indirect file function with the following POKES, where N is the starting page number.

```
POKE 9554,N                          for OS-65D

POKE 14646,91 and
 POKE 11667,N                        for OS-65U
```

2) LIST the program between square brackets as follows: With the program in the workspace, type

```
LIST[ <RETURN>
(wait for listing to end)
] <@> <RETURN>
```

If the keyboard is a polled keyboard use these commands instead:

```
LIST <SHIFT/K> <RETURN>
(wait for listing to end)
<SHIFT/M> <@> <RETURN>
```

The first bracket "[", <SHIFT/K> will not appear on the video screen. The second bracket appears twice as "]]".

If the end of the listing appears garbled the indirect file was not placed high enough in memory and the end of the program in the workspace has been overwritten.

E. FROM INDIRECT FILE TO WORKSPACE

To move a program from an indirect file to the workspace:

1) enter the appropriate POKEs, where N is the starting page number of the indirect file

POKE 9368,N	for OS-65D
POKE 14721,24 and POKE 11667,N	for OS-65U

2) enter the command:

<CTRL/X> <RETURN>

A listing of the program in the indirect file will appear ending with the bracket closure "]]". On some systems there will be a harmless error message before or after the listing. To see the contents of the workspace enter the command LIST.

CHAPTER 13

LINKING PROGRAMS TO MACHINE LANGUAGE ROUTINES

The USR function permits leaving a BASIC program, executing a machine language routine, and then returning to the BASIC program.

A. USR FUNCTION

The USR function has the form

```
USR(X)
```

where X is an arithmetic expression. The value of X can be sent to the machine language routine and a single value can be returned as USR(X).

If no paramenters are passed the function is used in the form

```
N  Y = USR(X)
```

where N is a line number, and X and Y are dummy variables. Control passes to the machine language routine at line N and then returns to the next line. It is often more convenient to use the second form and pass parameters by PEEK and POKE rather than to use the parameter passing feature of the USR function. If no parameters are passed, Y is assigned the value of X.

Before the machine language routine can be called by the USR function its starting address must first be POKEd into memory. The location depends upon the version of BASIC that is used. Letting LO denote the decimal value of the low byte of the starting address and HI denote the decimal value of the high byte, one of the following POKEs must be used:

```
POKE11, LO and
 POKE 12, HI      for ROM BASIC

POKE 574, LO and
 POKE 575, HI         for OS-65D

POKE 8778, LO and
 POKE 8779, HI        for OS-65U
```

For example, if the routine starts at $4ØØØ then 4Ø is the high byte and ØØ is the low byte. Converting to decimal, HI is 64 and LO is Ø.

PASSING PARAMETERS

The machine language routine begins by calling a routine whose starting address is a $ØØØ6. This routine converts the argument X into a 16 bit two's complement number which is then stored. The storage location of this number depends upon the BASIC used; as follows:

HIGH BYTE	LOW BYTE	BASIC USED
$ØØAE	$ØØAF	ROM BASIC
$ØØB1	$ØØB2	65D and 65U

The value of X is now available for the machine language routine.

The machine language routine ends by placing the value to be returned to the BASIC program in the accumulator (high byte) and the Y register (low byte); then calling a subroutine that starts at $ØØØ8. This subroutine will pass the value to the BASIC program as USR(X) and then return control to the BASIC program.

EXAMPLE

An example is given in this section of a program in 65D BASIC and a machine language routine that are linked by and have parameters passed by the USR function. In the example, the argument of the USR function is an integer H between Ø and 255. The value of H is passed to the machine language routine which then returns as USR(H) the number of times the character whose ASCII value is H appears on the video screen.

The BASIC program:

```
1Ø POKE 574,Ø
2Ø POKE 575,64
3Ø INPUT "ENTER CHARACTER";A$
4Ø H=ASC(A$)
5Ø N=USR(H)
6Ø PRINT N
7Ø END
```

The machine language routine:

```
 1Ø                          ;passing parameters to USR function
 2Ø                          ;N=USR(H)
 3Ø                          ;H=character number Ø<=H<=255
 4Ø                          ;N=count of how many times the character
 5Ø                          ;  appears on the screen
 7Ø 3FFC                     *=$3FFC
 8Ø 3FFC 6CØ6ØØ CALL         JMP  (6)
 9Ø 4ØØØ                     *=$4ØØØ
11Ø 4ØØØ 2ØFC3F  START      JSR CALL                    integerize H
17Ø 4ØØ3 A5B2               LDA $B2                     the result
18Ø 4ØØ5 A2DØ               LDX #$DØ
19Ø 4ØØ7 8E194Ø             STX COMP+2                  screen addr (hi)
2ØØ 4ØØA A2ØØ               LDX #Ø
21Ø 4ØØC 8E184Ø             STX COMP+1                  screen addr (lo)
22Ø 4ØØF 8E364Ø             STX COUNT
23Ø 4Ø12 8E374Ø             STX COUNT+1                 initialize counter
24Ø 4Ø15 AØØ8               LDY #8                      this many pages per screen
25Ø 4Ø17 DDFFFF COMP        CMP $FFFF,X                 dummy address
26Ø 4Ø1A DØØ8               BNE END
27Ø 4Ø1C EE374Ø             INC COUNT+1                 count it
28Ø 4Ø1F DØØ3               BNE END
29Ø 4Ø21 EE364Ø             INC COUNT                   do this if lo half rolls over
3ØØ 4Ø24 E8       END       INX
31Ø 4Ø25 DØFØ               BNE COMP
32Ø 4Ø27 EE194Ø             INC COMP+2
33Ø 4Ø2A 88                 DEY
34Ø 4Ø2B DØEA               BNE COMP
35Ø 4Ø2D AD364Ø             LDA COUNT
36Ø 4Ø3Ø AC374Ø             LDY COUNT+1
37Ø 4Ø33 6CØ8ØØ             JMP (8)
38Ø 4Ø36 ØØ       COUNT     .BYTE Ø,Ø
38Ø 4Ø41 ØØ
```

These two programs can be combined into the following one; the machine language routine is directly POKED into memory after converting each hex instruction to its decimal equivalent.

```
 2 FOR I=Ø TO 2
 4 READ V
 6 POKE 1638Ø+I,V
 8 NEXT
1Ø FOR =Ø TO 55
2Ø READ V
3Ø POKE 16384+I,V
4Ø NEXT
5Ø POKE 574,Ø
6Ø POKE 575,64
7Ø INPUT"ENTER CHARACTER";A$
8Ø H=ASC(A$)
9Ø N=USR(H)
1ØØ PRINT N
11Ø DATA 108,6,Ø
12Ø DATA 32,252,63,165,178,162,2Ø8
13Ø DATA 142,25,64,162,Ø,142,24,64
14Ø DATA 142,54,64,142,55,64,16Ø,8
15Ø DATA 221,255,255,2Ø8,8,238,55
16Ø DATA 64,2Ø8,3,238,54,64,232,2Ø8
17Ø DATA 24Ø,238,25,64,136,2Ø8,234
18Ø DATA 173,54,64,172,55,64
19Ø DATA 108,8,Ø,Ø,Ø
```

APPENDIX 1

ASCII CHARACTER CODES

CODE	CHAR	CODE	CHAR	CODE	CHAR
ØØ	NUL	2B	+	56	V
Ø1	SOH	2C	,	57	W
Ø2	STX	2D	–	58	X
Ø3	ETX	2E	.	59	Y
Ø4	EOT	2F	/	5A	Z
Ø5	ENQ	3Ø	Ø	5B	[
Ø6	ACK	31	1	5C	/
Ø7	BEL	32	2	5D	]
Ø8	BS	33	3	5E	^
Ø9	HT	34	4	5F	_
ØA	LF	35	5	6Ø	
ØB	VT	36	6	61	a
ØC	FF	37	7	62	b
ØD	CR	38	8	63	c
ØE	SO	39	9	64	d
ØF	SI	3A	:	65	e
1Ø	DLE	3B	;	66	f
11	DC1	3C	<	67	g
12	DC2	3D	=	68	h
13	DC3	3E	>	69	i
14	DC4	3F	?	6A	j
15	NAK	40	@	6B	k
16	SYN	41	A	6C	l
17	ETB	42	B	6D	m
18	CAN	43	C	6E	n
19	EM	44	D	6F	o
1A	SUB	45	E	7Ø	p
1B	ESC	46	F	71	q
1C	FS	47	G	72	r
1D	GS	48	H	73	s
1E	RS	49	I	74	t
1F	US	4A	J	75	u
2Ø	SP	4B	K	76	v
21	!	4C	L	77	w
22	"	4D	M	78	x
23	#	4E	N	79	y
24	$	4F	Ø	7A	z
25	%	50	P	7B	{
26	&	51	Q	7C	}
27	'	52	R	7D	!
28	(	53	S	7E	÷
29	)	54	T	7F	DEL
2A	*	55	U		

APPENDIX 2

ERROR CODES

DISK BASIC	ROM BASIC	
BS	B	Bad subscript: Matrix outside DIM statement range, etc.
CN	C	Continue Errors: Attempt to inappropriately continue from BREAK or STOP.
DD	D	Double Dimension: Variable dimensioned twice Remember subscripted variables default to demension 1Ø.
FC	F	Function Call Error: Parameter passed to function out of range.
ID	I	Illegal Direct: INPUT and DEFIN statements cannot be used in direct mode.
LS	L	Long String: String longer than 255 characters.
NF	N	NEXT without FOR.
OD	O	Out of Data: More reads than data.
OM	O	Out of Memory: Program too big or too many GOSUBs, FOR-NEXT loops or variables.
OV	O	Overflow: Result of calculation too large.
RG	R	RETURN without GOSUB.
SN	S	Syntax Error: Typo, etc.
ST	S	String Temporaries: String expression too complex.
TM	T	Type Mismatch: String variable mismatched to numeric variable.
UF	U	Undefined Function.
US	U	Undefined Statement: Attempt to jump to nonexistent line number.
/Ø	/	Division by Zero.

The following messages are not available in ROM BASIC.

OS	Out of String Space: Same as OM.
DV	Device Error. Only available in OS-65U. See operators manual for list of Dsik Error Codes.
FS	Full Stack: Stack overflow. Only available in OS-65U.

APPENDIX 3

KEYWORD INDEX WITH EXAMPLES

PAGE	NAME	EXAMPLES	BASICS
19	ABS	ABS(X)	ALL
3	AND	IF B>Ø AND B<5 THEN 1ØØ	ALL
2Ø	ASC	ASC(X$)	ALL
2Ø	ATN	ATN(X)	ALL
21	CHR$	CHR$(I)	ALL
17	CLEAR	CLEAR	ALL
28	CLOSE	DISK CLOSE,6	65D
29		CLOSE	65U
		CLOSE 3	65U
15	CONT	CONT	ALL
2Ø	COS	COS(X)	ALL
6	DATA	DATA 4,78,"BIG"	ALL
23	DEF	DEF FNA(X)=X*SIN(X)	ALL
3Ø	DEV	DEV A	65U
18	DIM	DIM A(2Ø,5Ø),B$(2,4,5)	ALL
28	DISK	DISK CLOSE,5	65D
28	DISK!	DISK!"PU PROG1"	65D
13	END	END	ALL
28	EXIT	EXIT	65D
19	EXP	EXP(X)	ALL
29	FIND	FIND "LOAN",2	65U
3Ø	FLAG	FLAG Ø3	65U
23	FN	DEF FNA(X)=X*SIN(X)	ALL
12	FOR	FOR I=1 TO 1Ø	ALL
17	FRE	FRE(X)	ALL
28	GET	DISK GET ,1	65D
23	GOSUB	GOSUB 15Ø	ALL
24		ON X GOSUB 100,2ØØ	ALL
11	GOTO	GOTO 1ØØ	ALL
11		ON X GOTO 1ØØ,2ØØ	ALL
11	IF	IF X<5 GOTO 27Ø	ALL

PAGE	NAME	EXAMPLES	BASICS
11		IF S>B THEN PRINT"TOO BIG"	ALL
29	INDEX	X = INDEX(1)	65U
29	INDEX<	INDEX<1> = 3 + N	65U
6	INPUT	INPUT"ENTER YOUR NAME";N$	ALL
9		INPUT#8,D$	65D,65U
29	INPUT%	INPUT% 2,X,Y	65U
19	INT	INT(X)	ALL
21	LEFT$	B$ = LEFT$(A$,5)	ALL
21	LEN	X = LEN(A$)	ALL
6	LET	LET X = Z + COS(Y)	ALL
15	LIST	LIST	ALL
		LIST 10Ø	ALL
		LIST -10Ø	ALL
		LIST 10Ø-	ALL
		LIST 10Ø-20Ø	ALL
15		LIST#1,1Ø-2Ø	65D,65U
27	LOAD	LOAD	ROM,65U
28		DISK!"LOAD PROG1"	65D
19	LOG	LOG(X)	ALL
21	MID$	A$=MID$(B$,2,3)	ALL
		A$=MID$(B$,2)	ALL
15	NEW	NEW	ALL
12	NEXT	NEXT	ALL
		NEXT I	ALL
3	NOT	NOT(A<5 AND B=Ø)	ALL
27	NULL	NULL 8	ALL
12	ON	ON X GOTO 10Ø,20Ø	ALL
24		ON X GOSUB 10Ø,20Ø	ALL
28	OPEN	DISK OPEN,6,"FILE3"	65D
28		OPEN"FILE2","PASS",3	65U
28		OPEN"FILE2",3	65U
3	OR	IF A<R OR A>S THEN 29Ø	ALL
25	PEEK	PEEK(23456)	ALL
25	POKE	POKE 32456 ,76	ALL
9	POS	POS(X)	ALL
7	PRINT	PRINT X,Y;"TOO LARGE"	ALL
9		PRINT#4, "LINE PRINTER"	65D,65U
29	PRINT%	PRINT% 2,X	65U
28	PUT	DISK PUT	65D

PAGE	NAME	EXAMPLES	BASICS
29		DISK!"PUT PROG1"	65D
6	READ	READ S,T	ALL
16	REM	REM DETERMINE THE RATIO	ALL
7	RESTORE	RESTORE	ALL
23	RETURN	RETURN	ALL
21	RIGHT$	A$=RIGHT$(B$,3)	ALL
19	RND	RND(1)	ALL
15	RUN	RUN	ALL
		RUN 200	ALL
		RUN"PROG2	65D,65U
27	SAVE	SAVE	ROM,65U
19	SGN	SGN(X-5)	ALL
20	SIN	SIN(X)	ALL
9	SPC(	SPC(4)	ALL
20	SQR	SQR(X)	ALL
13	STEP	FOR X=1 TO 2 STEP .5	ALL
13	STOP	STOP	ALL
21	STR$	A$=STR$(X)	ALL
8	TAB(	TAB(12)	ALL
20	TAN	TAN(X-B)	ALL
11	THEN	IF A<B THEN GOSUB 200	ALL
12	TO	FOR I=1 TO 5	ALL
34	USR	USR(X)	ALL
22	VAL	VAL(A$)	ALL
25	WAIT	WAIT I,J	ALL
		WAIT I,J,K	ALL

INDEX