BASIC-E/65

VERSION 2.1

180 Ridge Road
Cimarron, CO 81220

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

This manual describes the Naval Postgraduate School BASIC Language (BASIC-E) as modified for and implemented on the 6502 operating system DOS/65. BASIC-E/65 consists of two subsystems, the compiler (COMPILE.COM) and the run-time interpreter (RUN.COM). The compiler checks the syntax of a BASIC-E/65 source program and produces a pseudo-machine code file which may then be executed by the run-time interpreter.

BASIC-E/65 is intended to be used in the interactive mode with a CRT or printing terminal. It includes most features of the proposed ANSI standard BASIC [1] as well as extensive string manipulation and file input/output capabilities. BASIC-E/65 uses DOS/65 to handle all input/output operations and disk file management. The source program is an ASCII text file which is created and edited using the DOS/65 editor EDIT.COM.

The BASIC-E/65 compiler consists of a table-driven parser which checks statements for correct syntax and generates code for the BASIC-E/65 pseudo-machine. The code is executed by the run-time interpreter. The BASIC-E/65 pseudo-machine is a zero address stack computer. Floating point numbers are represented in 32 bits with an 8 bit exponent (including sign) and a 24 bit (including sign) mantissa. This provides from 6 to 7 decimal digits of significance. Variable length strings and N dimensional arrays are both dynamically allocated.

Section 2 of this manual describes the language elements. Section 3 provides operating instructions for BASIC-E/65. The appendices list the bibliography and provide examples of BASIC-E/65 programs.

SECTION 2 - DESCRIPTION OF THE BASIC-E/65 LANGUAGE

Elements of BASIC-E/65 are listed in alphabetical order in this section of the manual. A synopsis of the element is shown, followed by a description and examples of its use. The intent is to provide a reference to the features of this implementation of BASIC and not to teach the BASIC language. References [2,3] provide a good introduction to BASIC programming.

A program consists of one or more properly formed BASIC-E/65 statements. An END statement, if present, terminates the program and causes all following statements to be ignored. The entire ASCII character set is accepted, but all statements may be written using the common 64 character uppercase subset. Section 3 provides information on source program format.

In this section the synopsis presents the general form of the element. Square brackets [ ] denote an optional feature while pairs of &'s indicate that the enclosed section may be repeated zero or more times. Terms enclosed in !'s are either non-terminal elements of the language, which are further defined in this section or are terminal symbols. All special characters and capitalized words are terminal symbols.

ABS predefined function

SYNOPSIS:

ABS(!expression!)

DESCRIPTION:

The ABS function returns the absolute value of the expression. The argument must evaluate to a floating point number.

EXAMPLES:

```
a=ABS(X)
x=ABS(X*Y)
```

ASC predefined function

SYNOPSIS:

ASC(!expression!)

DESCRIPTION:

The ASC function returns the ASCII numeric value of the first character of the expression. The argument must evaluate to a string. If the length of the string is 0 (null string) an error will occur.

EXAMPLES:

```
a=ASC(A$)
char.num=ASC("X")
seven.count=ASC(RIGHT$(A$,7))
```

ATN predefined function

SYNOPSIS:

ATN(!expression!)

DESCRIPTION:

The ATN function returns the arctangent of the expression. The argument must evaluate to a floating point number.

EXAMPLES:

```
angle=ATN(X)
x.angle=ATN(SQR(SIN(X)))
```

PROGRAMMING NOTE:

All other inverse trigonometric functions can be computed from the arctangent using simple identities.

CALL predefined function

SYNOPSIS:

CALL(!expression!)

DESCRIPTION:

The CALL function executes the machine language routine at the address corresponding to the value of the expression. The expression must evaluate to a floating point number and is converted to an integer modulo 65536. The value returned by the function must be a 16 bit integer contained in the A (low half of number) register and the Y register (high half of the number). All parameters required by the routine must have been stored using POKE statements at locations determined by the programmer. The machine language routine must execute a 6502 RTS to return control to BASIC-E/65.

EXAMPLES:

```
VOLTAGE=CALL(A.TO.D.CONVERTER)*SCALE.FACTOR
DUMMY=CALL(USER.IO.LOCATION)
```

PROGRAMMING NOTE:

Since CALL is a function it must appear as part of an expression and not as a stand-alone statement even if there is no meaningful return value from the machine language routine. In such a case a dummy variable should be used and its value ignored.

CHR$ predefined function

SYNOPSIS:

CHR$(!expression!)

DESCRIPTION:

The CHR$ function returns a character string of length one consisting of the character whose ASCII equivalent is the expression converted to an integer modulo 128. The argument must evaluate to a floating point number.

EXAMPLES:

```
PRINT CHR$(A)
A$=CHR$(12)
S.CHAR$=CHR$(A+B/C)*SIN(X))
```

PROGRAMMING NOTE:

CHR$ can be used to send control characters such as a linefeed to the output device. The following statement would accomplish this:

```
PRINT CHR$(10)
```

CLOSE statement

SYNOPSIS:

[!line number!] CLOSE !expression!&,!expression!&

DESCRIPTION:

The CLOSE statement causes the file specified by each expression to be closed. Before the file may be referenced again it must be reopened using a FILE statement. An error occurs if the specified file has not previously appeared in a file statement.

EXAMPLES:

```
    CLOSE 1
150 CLOSE I,K,L*M-N
```

PROGRAMMING NOTE:

On normal completion of a program all open files are closed. If the program terminates abnormally it is possible that files created by the program will be lost.

constant

SYNOPSIS:

[!sign!]!integer!.[!integer!][E!sign!!exp!]
["]!character string![“]

DESCRIPTION:

A constant may be either a numeric constant or a string constant.

- All numeric constants are stored as floating point numbers. Strings may contain any ASCII character. Numeric constants may be either signed or unsigned integers, decimal numbers, or numbers expressed in scientific notation. Numbers up to 31 characters in length are accepted but the floating point representation of the number maintains approximately seven significant digits (1 part in 16,777,216). The largest magnitude that can be represented is approximately 1.7 times ten to the 38th power. The smallest non-zero magnitude that can be represented is approximately 5.9 times ten to the minus 39 power.
- String constants may be up to 255 characters in length. Strings entered from the console, in a data statement, or read from a disk file may be either enclosed in quotation marks or delimited by a comma. Strings used as constants in the program must be enclosed in quotation marks.

EXAMPLES:

```
10
-100.75639E-19
"THIS IS THE ANSWER"
```

COS predefined function

SYNOPSIS:

COS(!expression!)

DESCRIPTION:

COS is a function which returns the cosine of the expression. The argument must evaluate to a floating point number expressed in radians.

EXAMPLES:

```
cos.angle.b=COS(B)
unk.cos=COS(SQR(X-Y))
```

DATA statement

SYNOPSIS:

[!line number!] DATA !constant!&,!constant!&

DESCRIPTION:

DATA statements define string and floating point constants which are assigned to variables using a READ statement. Any number of DATA statements may occur in a program. The constants are stored consecutively in a data area as they appear in the program and are not syntax checked by the compiler. Strings may be enclosed in quotation marks or optionally delimited by commas.

EXAMPLES:

```
10 DATA 10.0,11.72,100
   DATA "XYZ",11.,THIS IS A STRING
```

DEF statement

SYNOPSIS:

[!line number!] DEF !function name!(!dummy argument list!) =
!expression!

DESCRIPTION:

The DEF statement specifies a user defined function which returns a value of the same type as the function name. One or more expressions are passed to the function as arguments and are used in evaluating the expression specified for the function. The passed values may be in string or floating point form but must match the type of the corresponding dummy argument. Recursive calls are not permitted. The expression in the DEF statement may reference variables other than the dummy arguments, in which case the current value of the variable is used in evaluating the expression. The type of the function name must match the type of the expression.

EXAMPLES:

```
10 DEF FNA(X,Y)=X+Y-A
   DEF FNB$(A$,B$)=A$+B$+C$
   DEF FN.COMPUTE(A,B)=A+B-FNA(A,B)+D
```

DIM statement

SYNOPSIS:

[!line number!] DIM !identifier!(!subscript list!)
&,!identifier!(!subscript list!)&

DESCRIPTION:

The DIM statement dynamically allocates space for floating point or string arrays. String array elements may be of any length up to 255 bytes and change in length dynamically as they assume different values. Initially all floating point array elements are set to zero and all string array elements are null strings. An array must be dimensioned explicitly -- no defaults are provided. Arrays are stored in row major order. Expressions in subscript lists are evaluated as floating point numbers and rounded to the nearest integer when determining the size of the array. All subscripts have an implied lower bound of zero. When array elements are referenced a check is made to ensure that the element resides in the referenced array.

EXAMPLES:

```
DIM A(10,20), B(10)
DIM B$(2,5,10), C(I+7.3,N),D(I)
DIM X(A(I),M,N)
```

PROGRAMMING NOTE:

A DIM statement is an executable statement and each execution will allocate a new array.

END statement

SYNOPSIS:

[!line number!] END

DESCRIPTION:

An END statement indicates the end of the source program. It is optional and, if present, terminates reading of the source program. If any statements follow the END statement they are ignored.

EXAMPLES:

```
10 END
   END
```

EXP predefined function

SYNOPSIS:

EXP(!expression!)

DESCRIPTION:

The EXP function returns the result of raising e (the base of natural logarithms) to the power defined by the argument. The argument must evaluate to a real number.

EXAMPLES:

```
x=EXP(1.397)
rate=1.2*EXP(data.value)
```

expression

DESCRIPTION:

Expressions consist of algebraic combinations of variables, constants, and operators. The hierarchy of operators is:

1. ()
2. ^
3. *, /
4. +, -, concat (+), unary +, unary -
5. relational ops <, <=, >, >=, =, <>, LT, LE, GT, GE, EQ, NE
6. NOT
7. AND
8. OR, XOR

Relational operators result in a 0 if false and a -1 if true. NOT, AND, and OR are performed on 24 bit two's complement representation of the integer portion of the variable. The result is then converted to a floating point number. String variables may be operated on by relational operators and concatenation only. Mixed string and numeric operations are not permitted.

EXAMPLES:

X+Y

A$+B$

(A<=B) OR (C$>D$)/(A-B AND D)

FILE statement

SYNOPSIS:

[!line number!] FILE !variable![(!expression!)]
&,!variable![(!expression!)]&

DESCRIPTION:

A file statement opens files for use by the program. If the file does not exist, it will be created as an empty file. The numbers used to reference the files in READ, PRINT, CLOSE, and IF END# statements are determined by the order in which the files are opened. The value assigned to the first file is 1, the second is 2, and so forth. There may be any number of FILE statements in a program, but no more than 20 files can be open at any one time. The variable must be a non-subscripted string variable that contains the name of the file. The name must comply with DOS/65 UFN requirements and can include a drive prefix. The optional expression designates the logical record length of the file. If no length is specified, the file is an unblocked file and if the record length is present the file is a blocked file. More data on disk file I/O can be found in section 3.3.

EXAMPLES:

```
input$="e:username.dat" : output$=default.file$
FILE INPUT$, OUTPUT$(N*3-J)
```

PROGRAMMING NOTE:

The run-time interpreter will always assign the lowest available (not already assigned) number to the file being opened. Thus if files are closed and others opened it is possible that number assignment will vary with program flow. The safest approach is either to open all files at once or to make sure that none are closed except at program termination.

FOR statement

SYNOPSIS:

[!line number!] FOR !index!=!expression! TO
!expression! [STEP!expression!]

DESCRIPTION:

Execution of all statements between the FOR statement and its corresponding NEXT statement is repeated until the indexing variable, which is incremented by the STEP expression after each iteration, reaches the exit criteria. If the step is positive, the loop exit criteria is that the index exceeds the value of the TO expression. If the step is negative, the index must be less than the TO expression for the exit criteria to be met. The index must be an unsubscripted variable and is initially set to the value of the first expression. Both the TO and STEP expressions are evaluated on each loop, and all variables associated with the FOR statement may change within the loop. If the STEP clause is omitted, a default value of 1 is assumed. A FOR loop is always executed at least once. A step of 0 may be used to loop indefinitely.

EXAMPLES:

```
FOR I=1 TO 10 STEP 3
FOR INDEX=J*K-L TO 10*SIN(X)
FOR I=1 TO 2 STEP 0
```

PROGRAMMING NOTE:

If a step of 1 is desired the step clause should be omitted.
Execution will be faster since fewer run-time checks must be made.

FRE predefined function

SYNOPSIS:

FRE

DESCRIPTION:

The FRE function returns the number of bytes of unused space in the free storage area.

EXAMPLE:

```
print FRE
```

function name

SYNOPSIS:

FN!identifier!

DESCRIPTION:

Any identifier starting with FN refers to a user-defined function. The function name must appear in a DEF statement prior to being used in an expression. There must not be any spaces between the FN and the identifier.

EXAMPLES:

```
d=FNA
answer$=FN.BIGGER.$(input.$)
```

GOSUB statement

SYNOPSIS:

[!line number!] GOSUB !line number!
[!line number!] GO SUB !line number!

DESCRIPTION:

The address of the next sequential instruction is saved on the run-time stack and control is transferred to the subroutine labeled with the line number following the GOSUB or GO SUB.

EXAMPLES:

```
10 GOSUB 300
   GO SUB 100
```

PROGRAMMING NOTE:

The max depth of GOSUB calls allowed is controlled by the size of the run-time stack which is currently set at 12.

GOTO statement

SYNOPSIS:

[!line number!] GOTO !line number!
[!line number!] GO TO !line number!

DESCRIPTION:

Execution continues at the statement labeled with the line number following the GOTO or GO TO.

EXAMPLES:

```
100 GOTO 50
    GO TO 10
```

identifier

SYNOPSIS:

!letter!&!letter!or!number!or.&[$]

DESCRIPTION:

An identifier begins with an alphabetic character followed by any number of alphanumeric characters or periods. Only the first 31 characters are considered unique. If the last character is a dollar sign the associated variable is of type string, otherwise it is of type floating point.

EXAMPLES:

```
A
B$
XYZ.ABC
PAY.RECORD.FILE.NUMBER.76
```

PROGRAMMING NOTE:

All lowercase letters appearing in an identifier are converted to uppercase unless compiler toggle D is set to off.

IF statement

SYNOPSIS:

[!line number!] IF !expression! THEN !line number!
[!line number!] IF !expression! THEN !statement list!
[!line number!] IF !expression! THEN !statement list!
ELSE !statement list!

DESCRIPTION:

If the value of the expression is not zero the statements which make up the statement list are executed. Otherwise the statement list following the ELSE is executed if present or the next sequential statement is executed. In the first form of the statement if the expression is not equal to zero an unconditional branch to the label occurs.

EXAMPLE:

```
IF A$ B$ THEN X=Y*Z
IF (A$ B$) AND (C OR D) THEN 300
IF B THEN X=3.0 : GOTO 200
IF J AND K THEN GOTO 11 ELSE GOTO 12
```

IF END statement

SYNOPSIS:

[!line number!] IF END #!expression! THEN !line number!

DESCRIPTION:

If an end of file is detected during a read from the file specified by the expression, control is transferred to the statement labeled with the line number following the THEN.

EXAMPLES:

```
IF END #1 THEN 100
IF END #FILE.NUMBER-INDEX THEN 700
```

PROGRAMMING NOTE:

On transfer to the line number following the THEN the stack is restored to the state prior to execution of the READ statement which caused the end of file condition.

INPUT statement

SYNOPSIS:

[!line number!] INPUT [!prompt string!;] !variable!&,!variable!&

DESCRIPTION:

The prompt string, if present, is printed on the console. A line of input data is read from the console and assigned to the variables as they appear in the variable list. The data items are separated by commas and/or blanks and terminated by a carriage return. Strings may be enclosed in quotation marks. If a string is not enclosed in quotation marks the first comma terminates the string. If more data is requested than was entered, or if insufficent data items are entered a warning is printed on the console and the entire line must be reentered.

EXAMPLES:

```
10 INPUT A,B
   INPUT "Size of Array"; N
   INPUT "Values"; A(I),B(I),C(A(I))
```

PROGRAMMING NOTE:

Trailing blanks in the prompt string are ignored. One question mark and a blank are always supplied by the system.

INT predefined function

SYNOPSIS:

INT(!expression!)

DESCRIPTION:

The INT function returns the largest integer less than or equal to the value of the expression. The argument must evaluate to a floating point number.

EXAMPLES:

```
print #1;INT(AMOUNT/100)
a=INT(3*X*SIN(Y))
```

LEFT$ predefined function

SYNOPSIS:

LEFT$(!expression!,!expression!)

DESCRIPTION:

The LEFT$ function returns the n leftmost characters of the first expression, where n is equal to the integer portion of the second expression. An error occurs if n is negative. If n is greater than the length of the first expression than the entire expression is returned. The first argument must evaluate to a string and the second to a floating point number.

EXAMPLES:

```
file.name$=LEFT$(A$,3)
dummy$=LEFT$(C$+D$,I-J)
```

LEN predefined function

SYNOPSIS:

LEN(!expression!)

DESCRIPTION:

The LEN function returns the length of the string expression passed as the argument. Zero is returned if the argument is the null string.

EXAMPLES:

```
ffggg=LEN(A$)
for.var=LEN(C$+B$)
print LEN(LASTNAME$+","+FIRSTNAME$)
```

LET statement

SYNOPSIS:

[!line number!] [LET] !variable!=!expression!

DESCRIPTION:

The expression is evaluated and assigned to the variable appearing on the left side of the equal sign. The type of the expression may be either string or floating point but must match the type of the variable.

EXAMPLES:

```
100 LET A=B+C
    X(3,A)=7.32*Y+X(2,3)
73  W=(A B) OR (C$ D$)
    AMOUNT$=DOLLARS$+"."+CENTS$
```

line number

SYNOPSIS:

!digit!&!digit!&

DESCRIPTION:

Line numbers are optional on all statements and are ignored by the compiler except when they appear in a GOTO, GOSUB, or ON statement. In these cases the line number must appear as the label of one and only one statement in the program. Line numbers may contain any number of digits but only the first 31 are considered significant by the compiler.

EXAMPLES:

```
100
4635276353
```

LOG predefined function

SYNOPSIS:

LOG(!expression!)

DESCRIPTION:

The LOG function returns the natural logarithm of the expression. The argument must evaluate to a non-zero, positive floating point number.

EXAMPLES:

```
z=LOG(X)
temperature=LOG((A+B)/D)
LOG10=LOG(X)/LOG(10)
```

MID$ predefined function

SYNOPSIS:

MID$(!expression!,!expression!,!expression!)

DESCRIPTION:

The MID$ function returns a string consisting of the n characters of the first expression starting at the mth character. The value of m is equal to the integer portion of the second expression while n is the integer portion of the third expression. The first argument must evaluate to a string and the second and third arguments must be floating point numbers. If m is greater than the length of the first expression a null string is returned. If n is greater than the number of characters left in the string then all characters from the mth character are returned. An error occurs if m or n is negative.

EXAMPLES:

```
lower.case.char$=MID$(A$,I,J)
print MID$(B$+C$,START,LENGTH)
```

NEXT statement

SYNOPSIS:

[!line number!] NEXT [!identifier!&,!identifier!&]

DESCRIPTION:

A NEXT statement denotes the end of the closest unmatched FOR statement. If the optional identifier is present it must match the index variable of the FOR statement being terminated. The list of identifiers allows matching multiple FOR statements. The line number of a NEXT statement may appear in an ON or GOTO statement, in which case execution of the FOR loop continues with the loop variables assuming their current values.

EXAMPLES:

```
10 NEXT
   NEXT I
   NEXT I,J,K
```

ON statement

SYNOPSIS:

(1) [!line number!] ON !expression! GOTO
!line number!&,!line number!&

(2) [!line number!] ON !expression! GO TO
!line number!&,!line number!&

(3) [!line number!] ON !expression! GOSUB
!line number!&,!line number!&

(4) [!line number!] ON !expression! GO SUB
!line number!&,!line number!&

DESCRIPTION:

The expression is rounded to the nearest integer value and is used to select the line number at which execution will continue. If the expression evaluates to 1 the first line number is selected and so forth. In the case of an ON ... GOSUB statement the address of the next instruction becomes the return address. An error occurs if the expression after rounding is less than one or greater than the number of line numbers in the list.

EXAMPLES:

```
10 ON I GOTO 10,20,30,40
ON J*K-M GO SUB 10,1,1,10
```

PEEK predefined function

SYNOPSIS:

PEEK(!expression!)

DESCRIPTION:

The PEEK function returns the value of the byte at the absolute memory address corresponding to the integer portion of the expression. The expression must evaluate to a positive floating point number between 0 and 65535.

EXAMPLES:

```
contents=PEEK(MEM.ADDRESS)
data=peek(X*3)
```

POKE statement

SYNOPSIS:

[!line numbe!] POKE !expression!,!expression!

DESCRIPTION:

The integer portion of the second expression is stored at the absolute address corresponding to the integer portion of the first argument. Both arguments must evaluate to positive floating point numbers. The first expression must be between 0 and 65535 and the second expression must be between 0 and 255.

EXAMPLES:

```
100 POKE 3,10
POKE MEM.ADDR,NEXT.CHAR
```

POS predefined function

SYNOPSIS:

POS

DESCRIPTION:

The POS function returns the value of the output line pointer. This value will range from 1 to 132 and designates the position at which the next character output by the system will appear.

EXAMPLE:

```
PRINT TAB(POS+3);X
```

PRINT statement

SYNOPSIS

(1) [!line number!] PRINT #!expression!,!expression!;
!expression!&,!expression!&

(2) [!line number!] PRINT #!expression!;!expression!&,!expression!&

(3) [!line number!] PRINT !expression!!delim!&!expression!!delim!&

DESCRIPTION:

A PRINT statement sends the value of the expressions in the expression list to either a disk file (type 1 or 2) or the console (type 3). A type 1 PRINT statement sends a random record specified by the second expression after the # to the disk file specified by the first expression after the #. An error occurs if there is insufficient space in the record for all values. A type 2 PRINT statement outputs the next sequential record to the file specified by the expression following the #. A type 3 PRINT statement outputs the value of each expression to the console. A space is appended to all numeric values and if the numeric item exceeds the right margin then a carriage return - linefeed is output before the item is printed. The delim between the expressions may be either a comma or a semicolon. The comma causes automatic spacing to the next tab position (14, 28, 42, etc). A semicolon indicates no spacing between printed values. If the last expression is not followed by a delim, a carriage return - linefeed is output and the print position is set equal to one. A carriage return - linefeed is automatically output anytime the print position exceeds 132.

EXAMPLES:

```
100 PRINT #1;A,B,A$+"*"
    PRINT #FILE,WHERE;A/B,D,"END"
    PRINT A,B,"The Answer is ";X
```

RANDOMIZE statement

SYNOPSIS:

[!line number!] RANDOMIZE

DESCRIPTION:

A RANDOMIZE statement initializes the random number generator.

EXAMPLES:

```
10 RANDOMIZE
RANDOMIZE
```

READ statement

SYNOPSIS:

(1) [!line number!] READ !variable!&,!variable!&

(2) [!line number!] READ #!expression!,!expression!;!variable!
&,!variable!&

(3) [!line number!] READ #!expression!;!variable!&,!variable!&

DESCRIPTION:

A READ statement assigns values to variables in the variable list from either a disk file (type (2) or (3)) or from a DATA statement (type (1)). Type (2) reads a random record specified by the second expression after the # from the disk file specified by the first expression after the # and assigns the fields in the record to the variables in the variable list. Fields may be floating point or string constants and are delimited by a blank or a comma. Strings may optionally be enclosed in quotes. An error occurs if there are more variables than fields in the record. The type (3) READ statement reads the next sequential record from the file specified by the expression following the # and assigns the fields to the variables as described above. A type (1) READ statement assigns values from DATA statements to the variables in the list. DATA statements are processed sequentially as they appear in the program. A attempt to read past the end of the last DATA statement will result in an error.

EXAMPLES:

```
100 READ A,B,C$
200 READ #1,I;PAY.REG,PAY.OT,HOURS.REG,HOURS.OT
    READ #FILE.NO;NAME$,ADDRESS$,PHONE$,ZIP
```

REM statement

SYNOPSIS:

[!line number!] REM [!remark!]
[!line number!] REMARK [!remark!]

DESCRIPTION:

A REM statement is ignored by the compiler and compilation continues with the statement following the next carriage return - line feed. The REM statement may be used to document a program. REM statements do not affect the size of the intermediate code file. An unlabeled REM statement may follow any statement on the same line. The line number of a REM statement may be used as the object of a GOTO, GOSUB, or ON statement.

EXAMPLES:

```
10 REM This is a remark
   REMARK THIS IS ALSO A REMARK
   LET X=0  REM Initial value of X
```

reserved word list

SYNOPSIS:

!letter!&!letter!&[$]

DESCRIPTION:

The following words are reserved in BASIC-E/65 and may not be used as identifiers:

```
ABS       AND  ASC  ATN     CALL   CHR$   CLOSE
COS       DATA DEF  DIM     ELSE   END    EQ
EXP       FILE FOR  FRE     GE     GO     GOSUB
GOTO      GT   IF   INPUT   INT    LE     LEFT$
LEN       LET  LOG  LT      MID$   NE     NEXT
NOT       ON   OR   PEEK    POKE   POS    PRINT
RANDOMIZE READ REM  RESTORE RETURN RIGHT$ RND
SGN       SIN  SINH SQR     STEP   STOP   STR$
SUB       TAB  TAN  THEN    TO     VAL
```

Reserved words must be preceded and followed by either a special character or a space. Spaces may not be embedded within reserved words. Unless compiler option D is set, lowercase letters are converted to uppercase prior to checking to see if an identifier is a reserved word.

RESTORE statement

SYNOPSIS:

[!line number!] RESTORE

DESCRIPTION:

A RESTORE statement repositions the pointer into the data area so that the next value read with a READ statement will be the first item in the first DATA statement. The effect of RESTORE statement is to allow rereading of the DATA statements.

EXAMPLES:

```
   RESTORE
10 RESTORE
```

RETURN statement

SYNOPSIS:

[!line number!] RETURN

DESCRIPTION:

Control is returned from a subroutine to the calling routine. The return address is maintained on the top of the run-time interpreter stack. No check is made to insure that the RETURN follows a GOSUB statement.

EXAMPLES:

```
130 RETURN
    RETURN
```

RIGHT$ predefined function

SYNOPSIS:

RIGHT$(!expression!,!expression!)

DESCRIPTION:

The RIGHT$ function returns the n rightmost characters of the first expression. The value of n is equal to the integer portion of the second expression. If n is negative an error occurs. If n is greater than the length of the first expression then the entire expression is returned. The first argument must be of type string and the second must be of type floating point.

EXAMPLES:

```
RIGHT$(X$,1)
RIGHT$(NAME$,LNG.LAST)
```

RND predefined function

SYNOPSIS:

RND

DESCRIPTION:

The RND function generates a uniformly distributed random number between 0 and 1.

EXAMPLE:

```
xyz=RND
```

SGN predefined function

SYNOPSIS:

SGN(!expression!)

DESCRIPTION:

The SGN function returns 1 if the value of the expression is greater than 0, -1 if the value is less than zero, and 0 if the value is equal to zero. The argument must evaluate to a floating point number.

EXAMPLES:

```
print SGN(X)
if SGN(A-B+C) then a=345
```

SIN predefined function

SYNOPSIS:

SIN (!expression!)

DESCRIPTION:

SIN is a predefined function which returns the sine of the expression. The argument must evaluate to a floating point number in radians.

EXAMPLES:

```
X=SIN(Y)
```

sine.angle=SIN(A-b/c)

SINH predefined function

SYNOPSIS:

SINH (!expression!)

DESCRIPTION:

SINH is a predefined function which returns the hyperbolic sine of the expression. The argument must evaluate to a floating point number.

EXAMPLES:

```
west.height=SINH(Y)
test.function=sinh(b c)
```

special characters

DESCRIPTION:

The following special characters are used by BASIC-E/65:

^	circumflex
(	open parenthesis
)	close parenthesis
*	asterisk
+	plus
-	minus
/	slant
:	colon
;	semicolon
<	less-than
>	greater-than
=	equal
#	number-sign
,	comma
cr	carriage return
\	back-slant

Any special character in the ASCII character set may appear in a string. Special characters other than those listed above will generate an error if they appear outside a string.

statement

SYNOPSIS:

[!line number!] !statement list!!cr!
[!line number!] !IF statement!!cr!
[!line number!] !DIM statement!!cr!
[!line number!] !DEF statement!!cr!
[!line number!] !END statement!!cr!

DESCRIPTION:

All BASIC-E/65 statements are terminated by a carriage return – line feed pair.

statement list

SYNOPSIS:

!simple statement!&:!simple statement!&

where a simple statement is one of the following:

FOR statement
NEXT statement
FILE statement
CLOSE statement
GOSUB statement
GOTO statement
INPUT statement
LET statement
ON statement
PRINT statement
READ statement
RESTORE statement
RETURN statement
RANDOMIZE statement
POKE statement
STOP statement
empty statement

DESCRIPTION:

A statement list allows more than one statement to occur on a single line.

EXAMPLES:

```
LET I=0 : LET J=12 : LET K=23
x=y+z/w : return
:::::::: PRINT "This is ok also!"
```

STR$ predefined function

SYNOPSIS:

STR$ (!expression!)

DESCRIPTION:

The STR$ function returns the ASCII string which represents the value of the expression. The argument must evauate to a floating point number.

EXAMPLES:

print #file.number;STR$(x)
pi$=STR$(3.14159)

subscript list

SYNOPSIS:

!expression!&,!expression!&

DESCRIPTION:

A subscript list may be used as part of a DIM statement to specify the number of dimensions and the extent of each dimension of the array being declared or as part of a subscipted variable to indicate which element of an array is being referenced. There may be any number of expressions but each must evaluate to a floating point number. A subscript list as part of DIM statement may not contain a reference to the array being dimensioned.

EXAMPLES:

```
var=X(10,20,20)
x$=Y$(I,J)
dim COST(AMT(I),PRICE(I))
```

SQR predefined function

SYNOPSIS:

SQR (!expression!)

DESCRIPTION:

SQR returns the square root of the expression. The argument must evaluate to a non-negative floating point number.

EXAMPLES:

```
print SQR(y)
hyp=SQR(X 2 + Y 2)
```

STOP statement

SYNOPSIS:

[!line number!] STOP

DESCRIPTION:

Upon encountering a STOP statement program execution terminates and all open files are closed. The print buffer is emptied and control returns to DOS/65. Any number of STOP statements may appear in a program. A default STOP statement is appended to all programs by the compiler.

EXAMPLES:

```
10 STOP
STOP
```

TAB predefined function

SYNOPSIS:

TAB (!expression!)

DESCRIPTION:

The TAB function positions the output line pointer to the position specified by the integer value of the expression rounded to the nearest integer. If the value of the rounded expression is less than the current print position, a carriage return - linefeed is output and the line pointer is set as described above. If the value of the rounded expression is 0 only a carriage return is output and then the output line pointer is set to one. Errors will result if the value of the rounded argument is greater than 132 or is negative. The TAB function may occur only in PRINT statements.

EXAMPLES:

```
print x;TAB(10);vb
print TAB(I + 1)
```

TAN predefined statement

SYNOPSIS:

TAN (!expression!)

DESCRIPTION:

TAN returns the tangent of the expression. The argument must be a floating point number and is assumed to be in radians. An error may occur if the argument is a multiple of pi/2 radians.

EXAMPLES:

```
dft=TAN(A)
x.tan=TAN(X-3*cos(y))
```

VAL predefined function

SYNOPSIS:

VAL (!expression!)

DESCRIPTION:

The VAL function converts the number in ASCII passed as the argument into a floating point number. The expression must evaluate to a string. Conversion continues until a character is encountered that is not part of a valid number or until the end of the string is encountered.

EXAMPLES:

```
IO.value=VAL(A$)
comp=VAL("3.789" + "E-7" + "This is ignored")
```

variable

SYNOPSIS:

!identifier![(!subscript list!)]

DESCRIPTION:

A variable in BASIC-E/65 may either represent a floating point number or a string depending on the type of the identifier. Subscripted variables must appear in a DIM statement before being used as a variable.

EXAMPLES:

```
X
Y$(3,10)
ABS.AMT(X(I),Y(I),S(I-1))
```

SECTION 3 - OPERATING INSTRUCTIONS

Execution of BASIC-E/65 programs under DOS/65 consists of three steps. First the source program (of type BAS) must be created on disk. Second the program is compiled by executing the BASIC-E/65 compiler (COMPILE.COM) with the name of the source program shown as a parameter for the compiler and with any desired compiler options also shown on the command line. Finally the intermediate file (of type INT) created by the compiler is executed by invoking the BASIC-E/65 run-time interpreter (RUN.COM) using the source program name as a parameter. Assuming that a source file named REPORT.BAS already exists the sequence of commands required would be:

```
COMPILE REPORT
RUN REPORT
```

CAUTION

If the compiler output indicated that errors were detected do not attempt to execute the INT file. The results are unpredictable and potentially disastrous.

Creation of the source program will normally be done using the DOS/65 editor EDIT.COM. As mentioned above it must have a type of BAS. The BASIC-E/65 text can be free-form and if a statement is not completed on a single line, a continuation character (\) must be the last character on the line. Spaces or tabs (ctl-i) may precede statements and any number of spaces or tabs may appear anywhere one space is permitted. Line numbers need only be used on statements to which control is passed. Line numbers do not have to be in ascending order. Using identifiers longer than two characters and indenting statements to enhance readability does not affect the size of the INT file created by the compiler.

3.1 COMPILER OPTIONS

There are 6 compiler options available which can be invoked by entering the appropriate letter on the COMPILE command line following a $ as shown below. If multiple options are desired they can be be included in the command line with or without any spaces as shown the following examples:

```
COMPILE TEST $A B
COMPILE REPORT $CD
```

Each option is described below.

3.1.1 OPTION A

Option A is an unlikely option for the user to employ as it generates during the second compiler pass a listing of the productions resulting from each input line. It is mainly intended for use by the developer for debugging of the compiler itself.

3.1.2 OPTION B
Option B causes the normal listing of the source during the second pass of the compiler to be suppressed. Lines containing errors will still be listed.

3.1.3 OPTION C
This option compiles the program but does not generate an INT file. This option is most useful during the early stages of debugging a program when the INT file is probably useless given the high probability of errors.

3.1.4 OPTION D
This option will prevent the compiler from automatically translating all lower case letters outside of strings to upper case. As the compiler normally does this translation, this option is most useful if identifiers are used which are spelled the same as reserved words but are in lower case.

3.1.5 OPTION E
This option causes code to be included in the INT file so that if a run time error occurs the line number will be listed along with the error message. Note that the line number in this case refers to the source line number beginning at 1 and is independent of any line numbers actually contained in the source file.

3.1.6 OPTION F
This option causes the compiler listing to go to the printer rather than the console.

3.2 SPECIAL FEATURES

3.2.1 PRINTER ONLY OUTPUT
Since LPRINTER and CONSOLE statements are not part of the BASIC-E/65 syntax there is no totally clean way to switch all output to the DOS/65 list device instead of the console or to switch back. The ctl-p toggle can be used to cause the output to go to both devices however that may not always be desirable. Because of this two special entry points have been provided at TEA+3 and TEA+6 to allow such switching to take place. These entry points must be entered using the CALL function. The following program illustrates use of these entry points.

```
tea=512  rem this is for tea=$200
printer=tea+3
console=tea+6
dummy=call(printer) rem this sends all following output to
list device
print "This will go to printer!"
dummy=call(console) rem back to console
print "This will go to console!"
end
```

NOTE

Note that after use of the CALL(TEA+3) entry all input prompts will go to the list device but inputs will be echoed on the console. All error messages will go to the console regardless of the output mode selected using these two entry points.

CAUTION

Use of direct output to the list device defeats the normal DOS/65 tab ($09) expansion. Make sure that your driver in SIM will expand any tabs if you plan to include tabs in any output streams through use of the CHR$(9) function.

3.2.2 DIRECT PEM/SIM CALLS

Two entry points have been provided at TEA+9 and TEA+12 to allow direct calls to PEM and SIM. Fixed storage locations have also been allocated at TEA+15, TEA+16 and TEA+17 for the A, Y and X register values needed as input and output parameters for these calls. A CALL(TEA+9) invokes the following assembly language and also returns a value corresponding to the 16 bit number formed by TEA+15 (low byte) and TEA+16 (high byte):

```
LDA TEA+15
LDY TEA+16
LDX TEA+17
JSR PEM
STA TEA+15
STY TEA+16
STX TEA+17
RTS
```

A CALL(TEA+12) is similar except that the assembly language invoked is as follows:

```
LDA TEA+15
LDY TEA+16
JSR SIM+(TEA+17)
STA TEA+15
STY TEA+16
STX TEA+17
RTS
```

Note that JSR SIM+(TEA+17) means that the location called in SIM is determined by adding the byte at TEA+17 to the SIM starting address. Since SIM must begin on a page boundary only the high byte of the SIM address is used in the calculation with TEA+17 supplying the low byte of the address. The resulting address is not checked for validity by BASIC-E/65.

CAUTION

Use of direct PEM and SIM calls may be useful but is potentially dangerous and may interfere with normal BASIC-E/65 file operations or other DOS/65 functions.

Use of either entry point assumes that the user has pre-stored the appropriate data at TEA+15, TEA+16 and TEA+17. The following program illustrates use of these direct calls:

```
tea=512 rem for tea=$200
pem=tea+9
sim=tea+12
a.register=tea+15
y.register=tea+16
x.register=tea+17
poke x.register,6
no.echo.input=call(pem) and 127 rem look at low byte only
print chr$(no.echo.input) rem send to console
end
```

3.3 DISK FILES

Disk files may be read, written or updated using both sequential and random access. Two basic types of files exist. The first type has no declared record size and is referred to as unblocked. The second type has a user-specified record size and is referred to as blocked. Blocked files may be accessed either sequentially or randomly while unblocked files can only be accessed sequentially.

In unblocked files there is no concept of a record as such. Each reference to the file either reads from or writes to the next field. At the end of each PRINT statement a carriage return and linefeed pair are written to the file. This allows data files to be created with EDIT.COM or TYPE(d) using CCM.

Blocked files consist of a series of fixed length records. The user specifies the logical record length with the FILE statement. An error occurs if a linefeed is encountered during a read from a blocked file or if the current record being built exceeds the block size during a write. At the end of a PRINT statement any remaining area in the record is filled with blanks by the system and then a carriage return and linefeed are added.

All data in files is stored as ASCII characters. Either a comma or carriage return denotes the end of a field. Blanks are ignored between fields.

3.4 MEMORY SIZE LIMITATIONS

Because both COMPILE.COM and RUN.COM are very large programs, the normal DOS/65 minimum memory sizes are not adequate for BASIC-E/65. The following table shows the absolute minimum sizes required to load COMPILE.COM and RUN.COM.

TEA	MINIMUM MEMORY SIZE
$200	18K
$400	18K
$800	19K
$1000	21K
$1400	22K
$2000	25K

Note that these minimums do not guarantee that a given source program can be compiled or that a given INT file can be executed. The minimum memory sizes shown allow only a little over 2K (i.e. the length of CCM) for the symbol table and similar areas when the compiler is running or for the INT file and data areas when the run-time interpreter is running. Large source programs will require more memory.

3.5 PAGE ZERO USAGE

BASIC-E/65 complies with the original DOS/65 recommendation that transients not use more than the region from $00 through $8F in page zero.

3.6 COMPILER OUTPUT

During the second pass of the compiler the source program is listed on the console (see section 3.1 for options). A sequential line count is shown at the beginning of each line. If the source line contains a line number that is referenced elsewhere in the program, the compiler line count is followed by a ":". In all other cases the line count is followed by a "*".

APPENDIX A - BIBLIOGRAPHY

1. Draft Proposed American National Standard Programming Language Minimal BASIC. X3J2/76-01 76-01-01. Technical Committee X3J2-BASIC American National Standards Committee X3 - Computers and Information Processing.

2. Worth, Thomas. BASIC for Everyone. Englewood Cliffs: Prentice Hall, Inc., 1976.

3. Albrecht, Fobert L., Fenkel, LeRoy and Brown, Jerry. BASIC. New York: John Wiley & Sons, Inc., 1973.

APPENDIX B - SAMPLE PROGRAMS

PROGRAM #1

```
remark program builds a file of mailing labels from a file containing 100
remark byte records which contain name and address information

      input "Name of source file (in uppercase)";source.file$
      input "Name of label file (in uppercase)";label.file$
      if end # 1 then 100
      file source.file$(100), label.file$ remark label file is unblocked

      for index=1 to 1 step 0 remark loop forever until eof
            read #1; first$, last$, street$, city$, state$, zip

                  remark lines are truncated at 60 characters

            line1$=left$(first$ + " " + last$, 60)
            line2$=left$(street$, 60)

                  remark insure zip not truncated

            line3$=left$(city$ + ", " + state$, 54)
            line3$=line3$ + " " + str$(zip)

            print #2;line1$
            print #2;line2$
            print #2;line3$
      next index

100   print "Job complete"
      stop
      end
```

PROGRAM #2

```
remark program to compute the first "n" fibonacci numbers
remark an input of 0 terminates the program

print "This program computes the first n fibonacci numbers."

for i=1 to 1 step 0 remark do forever
   input "Enter n (0 will terminate execution)";n
   if n=0 then \
      print "Program Terminated" : \
      stop
   if n<0 \
      then \
         print "N must be positive "; : \
         print "Please reenter" \
      else \
         gosub 300 remark calculate and print
next i

300 remark subroutine to calculate fibonacci numbers
    f1=1 : f2=1 remark intial values
    num=f1

remark handle first two numbers as special cases

for j=1 to 2
   gosub 400
   if n=0 then return
next j

remark handle remaining numbers

for j=1 to 1 step 0
   num=f1 + f2
   gosub 400
   f2=f1
   f1=num
   if n=0 then return
next j
return

400 remark print next number and decrement n
   print num, remark 5 to a line
   n=n-1
   return
   end
```

APPENDIX C - PROGRAMMING NOTES

PROGRAMMING NOTE #1 - SPEED OPTIMIZATION

Unlike interpreted BASICs such as Microsoft BASIC, BASIC-E/65 does not run faster if variables rather than constants are used were ever possible in a program. The following example illustrates this situation. The first program uses constants:

```
PRINT "START"
FOR I=1 TO 10000
A=I/10000
NEXT I
PRINT "END"
```

while this version of the same program uses variables in place of constants:

```
O=1
T=10000
PRINT "START"
FOR I=O TO T
A=I/T
NEXT I
PRINT "END"
```

The run time results in seconds at 1 MHz obtained from running these program using both BASIC-E/65 and Microsoft BASIC are shown in the following table:

	BASIC-E/65	MICROSOFT
CONSTANT	67	96
VARIABLE	70	56

These results are quite revealing. Not only is the variable approach not faster than the constant approach for BASIC-E/65, it is actually about 5% slower. It is clear that the old sage about using variables rather than constants is only true for an interpreted BASIC. For BASIC-E/65 you should use constants if possible.

PROGRAMMING NOTE #2 - LOGICAL VARIABLES

There is no such thing as a logical variable per se in BASIC-E/65. There is however one good way to simulate logical variables. If the following variables are assigned:

```
TRUE=-1
FALSE=0
```

then the following program would behave as expected:

```
RANDOM=TRUE
IF RANDOM THEN PRINT "File is random"
RANDOM=FALSE
IF NOT RANDOM THEN PRINT "File is not random"
```

The reason this works so nicely relates to how NOT and IF work. IF statements look at the arithmetic value of the IF expression. If the value is non-zero then the THEN statement list is executed and if the value is zero then the ELSE statement list, if any, is executed. Since NOT does a 1's complement of the 24 bit 2's complement representation of the expression it is clear that:

NOT TRUE --> NOT -1 --> NOT $FFFFFF --> 0

and

NOT FALSE --> NOT 0 --> $FFFFFF or -1

therefore we have exactly what we want in that:

NOT TRUE --> FALSE

and

NOT FALSE --> TRUE.

Things get a bit more complicated when using AND or OR but still work OK as long as all variables in the IF expression are defined as TRUE or FALSE.

PROGRAMMING NOTE #3 - ERROR MESSAGES

While the BASIC-E/65 COMPILE and RUN error messages are clear, they are of necessity brief. In order to assist the user in determining how to fix the program, the following pages list each error message along with some hints as to the possible cause or cure.

DUPLICATE LABELS OR SYNCHRONIZATION ERROR

Every line number must be different. Change the duplicate or eliminate the duplicate if not referenced elsewhere in the program. See BASIC-E/65 Programming Note #7 for other conditions to check for.

IDENTIFIER IN DIM PREVIOUSLY DEFINED

Eliminate the duplicate DIM. If you need to initialize an array more than once in a program then include the DIM in a subroutine or function.

PREDEFINED FUNCTION NAME PREVIOUSLY DEFINED

Make sure all function names are unique.

FOR LOOP INDEX NOT SIMPLE FLOATING POINT VARIABLE

Make sure that you did not use a string or an array.

INCORRECT NUMBER OF PARAMETERS IN FUNCTION REFERENCE

Each function reference must have the same number of parameters as in the DEF for that function.

INVALID PARAMETER TYPE IN FUNCTION REFERENCE

Make sure that the types, string or real, of the parameters in a function reference match the type in the DEF for that function.

UNDEFINED FUNCTION

Make sure the DEF appears in the program prior to the first call to the function. This message could also mean that an array was mistaken as a function reference.

INVALID CHARACTER

Eliminate the incorrect character.

EXPRESSION IN IF STATEMENT NOT FLOATING POINT

Only expressions which result in a numerical result are allowed. Strings can be used in comparisons (e.g. IF A$="Y" THEN 100) but not by themselves (e.g. IF A$ THEN 100).

ILLEGAL FLOATING POINT FORMAT

Make sure number is in correct format.

SUBSCRIPTED VARIABLE NOT PREVIOUSLY DIMENSIONED

BASIC-E/65 does not provide default dimensioning. Insert an explicit DIM in the program prior to the first use of the subscripted variable.

ARRAY NAME USED AS SIMPLE VARIABLE

Change one of the names. The same name can not be used for both a simple and a subscripted variable in BASIC-E/65.

STRING EXPRESSION NOT ALLOWED

Change the expression to a real.

MIXED MODE (STRING - FLOATING) EXPRESSION

Convert strings to reals using VAL or ASC. Alternatively convert reals to strings using STR$ or CHR$.

NEXT VARIABLE DOES NOT MATCH FOR A FOR/NEXT

Loop may not cross the boundary of another FOR/NEXT loop. Check loop boundaries and variable names.

NO PRODUCTION EXISTS

A syntax error was detected. Because BASIC-E/65 can continue statements from line to line, it is sometimes helpful to look at the lines before the point at which the error was flagged.

NEXT STATEMENT WITHOUT MATCHING FOR

Check for missing FOR or eliminate/correct NEXT.

INCORRECT NUMBER OF SUBSCRIPTS

The number of subscripts in the DIM and in a statement using the array do not match. Change the incorrect statement.

COMPILER STACK OVERFLOW

This error should not occur. If it does, please provide a copy of the BASIC-E/65 program to the manufacturer for review.

SYMBOL TABLE OVERFLOW

The program is too long to be compiled with the available memory. Increase available memory or shorten variable names.

UNDEFINED LABEL

A GOTO or GOSUB referenced a line number that could not be found.

VARC TABLE OVERFLOW

This error should not occur. If it does, please provide a copy of the BASIC-E/65 program to the manufacturer for review.

UNTERMINATED STRING

A string ran off the end of a line without the terminating ".

INVALID TYPE IN FILE IDENTIFIER
The file identifier in the FILE statement must be a string.

FOR WITHOUT MATCHING NEXT
The end of the program was reached and one or more FOR/NEXT loops were not terminated with a NEXT statement.

NO SOURCE - ABORTING
The specified source file could not be found. Make sure that the file exists as a .BAS file.

DISK ERROR - ABORTING
An error occurred during the read of the .BAS file or during write to the .INT file. Check for hard disk errors and make sure that there is enough room on the disk for the .INT file and for the .INT directory entry.

NULL STRING PASSED AS PARAMETER TO ASC FUNCTION
A zero length string was used as a parameter by ASC. Test for zero length before calling ASC.

ERROR WHILE CLOSING A FILE
An error code was returned by PEM when the file close function was executed. Check file with EDIT or by TYPEing.

DISK READ ERROR - UNWRITTEN DATA IN RANDOM ACCESS
PEM has indicated that no block is assigned to the DOS/65 record read. Check file with EDIT or by TYPEing.

DISK WRITE ERROR
A PEM error occurred during a disk write. The most likely causes are either a hard disk error or the file may have exceeded the available space on the diskette. Check file with EDIT or by TYPEing.

DIVISION BY ZERO
Check for zero if there is any possibility that divisor could be zero.

EOF FOR DISK FILE AND NO ACTION SPECIFIED
Use IF END# to specify statement to be executed in case EOF of file is reached during a read.

RECORD SIZE EXCEEDED FOR BLOCKED FILE
Check field lengths and remember to count separators, string delimiters and final RETURN and LINEFEED.

INVALID INPUT FROM CONSOLE
At least one parameter must be entered before typing RETURN.

INVALID RECORD IN RANDOM ACCESS
Check expression used to calculate record number. The record number must be one or more and positive.

ACCESSING AN UNOPENED FILE
Correct or add FILE statements to open the file.

ERROR WHILE CREATING A FILE
An error code was returned by PEM during a file creation operation. Most likely error is exhaustion of directory space.

FILE IDENTIFIER TOO LARGE OR ZERO
File identifiers in READ#, PRINT#, CLOSE, or IF END# must be in the range of 1 to 20.

ATTEMPT TO RAISE A NEGATIVE NUMBER TO A POWER
BASIC-E/65 use the LOG of a number to raise it to a power. The LOG of a negative number is not defined. Test for negative number and take absolute value only.

NO INT FILE FOUND IN DIRECTORY
Check directory and make sure that file was compiled.

ATTEMPT TO READ PAST END OF DATA AREA
Add additional DATA statements or insert a RESTORE.

ERROR WHILE OPENING A FILE
An error code was returned by PEM during a file opening operation. The most likely cause of the error is exhaustion of directory space.

INDEX IN ON STATEMENT OUT OF BOUNDS
Zero is not allowed nor is a number greater than the number of labels specified in the ON statement.

ATTEMPT TO READ PAST END OF RECORD ON BLOCKED FILE
Check expression in FILE statement. Recalculate record length remembering to count field separators, string delimiters and final RETURN and LINEFEED.

UNBLOCKED FILE USED WITH RANDOM ACCESS
Use correct format for FILE statement to specify record length.

ARRAY SUBSCRIPT OUT OF BOUNDS
Correct DIM or change subscript. The value used in the DIM specifies the maximum allowable subscript. The minimum is always zero.

STRING LENGTH EXCEEDS 255
Check length before concatenating.

SECOND PARAMETER OF MID IS NEGATIVE
Parameter must be a positive non-zero number.

ATTEMPT TO EVALUATE TANGENT OF PI OVER TWO
Result would be indeterminate. Check number before function call.

OUT OF MEMORY
Add memory to system or shorten the program.

ATTEMPT TO WRITE A QUOTE TO DISK
Quotes (") are used as string delimiters. They can not be written to disk files.

DISK DATA FIELD TOO LONG DURING READ
Field can not be longer than 80 characters in length.

OVERFLOW IN ARITHMETIC OPERATION
Variable was set to maximum. Check for possible errors.

ILLEGAL TAB ARGUMENT
Expression must be between 0 and 132. Negative values are illegal as are all values greater than 132.

ILLEGAL CHARACTER IN FILE NAME
Check for illegal character . ? * = : < > ; and DELETE ($7F).

PROGRAMMING NOTE #4 - BASIC DIFFERENCES

While BASIC-E/65 is very similar to Microsoft BASIC, it is not the same. Most of the differences are obvious (e.g., the file handling provisions of BASIC-E/65) but some are subtle. The following table lists some of the differences the user is most likely to forget.

IDENTIFIER - SIMPLE AND ARRAY

MICROSOFT

The same identifier can be used for both a simple variable and a subscripted variable. For example A and A(I) refer to different variables and can both be used in a single program.

BASIC-E/65

The same identifier can not be used as both a simple variable and a subscripted variable. For example use of A and A(I) in the same program would not be legal.

CHAINED IF STATEMENTS

MICROSOFT

Chained IF statements are legal. For example the following statement is allowed:

```
IF A=1 THEN IF B=2 THEN C=3
```

BASIC-E/65

Another IF statement can not be included in the statement list of an IF statement. The form shown above is not legal. The same result could be achieved with the following statement:

```
IF A=1 AND B=2 THEN C=3
```

SPC FUNCTION

MICROSOFT

The SPC(X) function can be used to insert X spaces into the output.

BASIC-E/65

There is no SPC function. Use the syntax TAB(POS(0)+X) to achieve the same result for X greater than or equal to one.

LEADING BLANKS

MICROSOFT

If a number is positive it is preceded by a blank.

BASIC-E/65

No blank is printed in front of positive numbers.

SPACES IN LINES

MICROSOFT

Spaces are not needed between keywords, variables, or other symbols. For example the following statements are equivalent:

```
IF SIN(A)=.5 THEN A=1
IFSIN(A)=.5THENA=1
```

BASIC-E/65

Spaces or tabs must be used. The first line shown above will compile but the second line would not.

VARIABLE NAME LENGTH

MICROSOFT

Variable names can be any length but only the first two characters are significant. The remainder of the name can not include any reserved words. For example the following would be an illegal variable name:

```
AIF
```

BASIC-E/65

Variable names can be any length but only the first 31 characters are significant. The name may contain reserved words as long as the significant characters do not exactly match a reserved word. The name shown above would be legal but CALL would not be legal.

TABS

MICROSOFT

Tabs (ctl-i) can not be used.

BASIC-E/65

Tabs can be used anywhere a blank is acceptable.

LINE NUMBERS

MICROSOFT

Each line must be numbered. Line numbers must be between 1 and 63999 inclusive.

BASIC-E/65

Only lines referenced by another statement need be numbered. Line numbers can be any valid number up to 31 digits long.

FIELD SEPARATORS

MICROSOFT

Most disk file extensions to Microsoft BASIC operate by redirecting the console input and output streams to the disk. As a result, explicit action must be taken during output to ensure that individual fields can be read into separate variables during subsequent input operations from the disk. In the following example the "," must be used to separate the fields:

```
PRINT #6,I;",";AI$
```

In addition the semicolon must normally be used between items in the print list to prevent wasted space on the disk.

BASIC-E/65

BASIC-E/65 automatically adds the necessary field separator (i.e., a comma) and also encloses all strings in quotes ("). Commas must be used between items in the print list but do not add extra spaces as would be done for console output. The line shown above would look like this when written for BASIC-E/65:

```
PRINT #6;I,AI$
```

LOGICAL OPERATIONS

MICROSOFT

Logical operations are done on 16 bit quantities.

BASIC-E/65

Logical operations are done on 24 bit quantities.

ARRAY DIMENSIONING

MICROSOFT

Default array dimensioning is provided.

BASIC-E/65

There is no default array dimensioning. All arrays must be explicitly dimensioned.

PROGRAMMING NOTE #5 - VERSION 1 TO 2 CHANGES

1) All BASIC-E/65 Version 1 programs will run under Version 2 however they must be re-compiled using the BASIC-E/65 Version 2 compiler (COMPILE.COM).

2) In BASIC-E/65 Version 2 all console output is done immediately. Version 1 placed all output in a buffer until either a carriage return - linefeed was output or the buffer overflowed. Immediate output makes it much easier to do screen handling and cursor positioning but should not affect any existing software.

3) During execution of a FILE statement, all lowercase characters in the file name are converted to uppercase by the run-time interpreter. Checks are also made for illegal characters (. ? * = : < > ; and DELETE ($7F)) and if detected result in program termination.

4) A special mode of operation for TAB has been defined when the argument is zero. In that case a carriage return is sent to the console without a linefeed and the print position pointer is set to one. This feature will help in design of screen handling routines.

5) TAB argument range checking has been implemented.

6) The record number to DOS/65 extent/record calculation in random disk I/O statements has been significantly speeded up. More importantly that calculation has been made compatible with the DOS/65 Version 2 file handling capabilities. Because of those changes the BASIC-E/65 Version 2 run-time interpreter (RUN.COM) can not be used under DOS/65 Version 1.0, 1.1 or 1.2.

PROGRAMMING NOTE #6 - RESERVING SPACE

In some applications it may be desirable to prevent BASIC-E/65 from using part of the TEA. This protected region could then be used to hold machine language routines, as a special I/O buffer, or for similar uses. Code similar to that shown below can be used to accomplish that purpose. This code must be the first code executed in any program. In particular it must appear before any DIM or FILE statements and prior to any array or string manipulations.

The protected region grows from PEM down and hence includes the memory that is normally occupied by CCM. As long as the contents of the region need not be preserved after exit from RUN, no special care need be taken as CCM will be reloaded during the WARM BOOT caused by the exit from RUN. If the data must be preserved after exit from CCM, then the region must be long enough so that the CCM area (i.e., the first 2048 bytes below PEM) is not used.

```
rem: basic-e/65 code to reserve space
rem: change version as needed
version=2
if version=1 \
   then \
      mbase.addr=151 : \
      init.addr=740
if version=2 \
   then \
      mbase.addr=155 : \
      init.addr=751
rem: first indicate length of region to reserve
length=4096
rem: now find current pem start address
pem.low=peek(260)
pem.high=peek(261)
pem=pem.low+256*pem.high
rem: now calculate start of reserved region
newpem=pem-length
newpem.high=int(newpem/256)
newpem.low=newpem-256*newpem.high
rem: make sure not too low
mbase=peek(mbase.addr)+256*peek(mbase.addr+1)
if (mbase+4)>newpem \
   then \
      print" Can not reserve space!" : \
      stop
rem: set pem link
poke 260,newpem.low
poke 261,newpem.high
rem: call routine to re-initialize free storage area and pointers
dummy=call(init.addr)
rem: replace correct pem link for i/o
poke 260,pem.low
poke 261,pem.high
```

PROGRAMMING NOTE #7 - SYNCHRONIZATION ERRORS

The error message:

DUPLICATE LABELS OR SYNCHRONIZATION ERROR

has two possible meanings. The first error condition, i.e., that two lines have the same label, is usually easy to spot and correct. The second condition is more difficult to understand and correct. The condition usually means that the compiler detected a difference in the length of the pseudo-code generated during the first and second pass through the source code. That condition can occur if the compiler detects an error during the first pass in a given line and ceases compilation of the line but does not detect the same error during the second pass.

While there may be more than one condition that could cause this problem, there is a "most likely" error. The following program illustrates the error:

```
1000 a=0
     gosub 2000
     ar$(a)=ar$(a)+"1"
     print a,ar$(a-1)
     stop
2000 dim ar$(20)
     return
     end
```

The problem is that while the array "ar$" will be properly dimensioned during execution under RUN, COMPILE does not know that "ar$" is an array when it is first encountered and hence will call the reference an error and terminate compilation of the line. However as soon as line 2000 is reached in the first pass, the compiler will note in the symbol table that "ar$" is an array. The second pass through the source will then create synchronization errors since the amount of code generated during the second pass will not be the same as that generated during the first pass. The solution is simple --- MAKE SURE THAT ARRAYS ARE DEFINED USING A DIM STATEMENT IN A LINE THAT OCCURS PRIOR TO ANY REFERENCE TO THE ARRAY! The DIM statement must be in a line that COMPILE sees before any reference not just that RUN executes before any other reference (although that also must be the case).