C B A S I C

A commercially oriented,
compiler/interpreter BASIC
language facility for
CP/M systems.

February 17, 1978

Disclaimer:

Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties or merchantability or fitness for any particular purpose. Further, Software Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Software Systems to notify any person of such revision or changes.

TABLE OF CONTENTS

APPENDICES

2.2 Notation

All of the CBASIC statements are described in this manual. Each description is preceded by a synopsis which presents the general form of the statement. The following notation is used for the synopsis:

Keywords and Symbols
: All special characters and capitalized words represent symbols which have special meaning in the language. For instance READ, REM and : are keywords in CBASIC.

Angle Brackets < >
: These terms denote an item which is defined in greater detail in the text.

Brackets [ ]
: Brackets denote an optional feature.

Braces { }
: Braces indicate that the enclosed section may be repeated zero or more times.

2.3 Line Numbers

Line numbers are optional on all statements and are ignored except when they appear in a GOTO, GOSUB, ON, or IF statement. In these cases, the line number must appear as the label of one and only one statement in the program. For example:

```
10  X = 3
20  Y = 2
30  GOTO 10
```

In the preceding program segment the 20 and 30 are not required. They are ignored during compilation. Line numbers may contain any number of digits but only the first 31 are considered significant by the compiler.

An additional feature of CBASIC line numbering is that any number, except 0 or 1, is a valid line number. This allows the use of non-integer line numbers. It is possible to write an entire program or subprogram with line numbers that are all decimal fractions and range between two consecutive integers. Line numbers can even be in exponential (E) format. This is a convenient feature when writing canned procedures that will be included in different mainline programs.

2. GENERAL INFORMATION

2.1 Statements

A program consists of one or more properly formed CBASIC statements. An END statement, if present, terminates the program, and additional statements are ignored. The entire ASCII character set is accepted, but all statements may be written using the common 64 character subset. Lower case letters are converted by the compiler to upper case execpt in strings and remarks. A compiler toggle, described in Chapter 13, will prevent this conversion.

CBASIC statements are free-form with the following requirements:

(1) When a statement is not completed on a single line, a continuation character (\) must be used. The statement can then be continued on the next line.

(2) All characters which follow the continuation character on the same line are ignored by the compiler.

(3) Multiple statements are allowed on one line but they must be separated by a colon (:). DATA, DEF, DIM and END must be the only statement on a line; IF must be the first statement on a line.

Spaces may precede statements; any number of spaces may appear wherever one space is permitted. Using identifiers longer than two characters and indenting statements to enhance readability does not affect the size of the intermediate file created by the compiler.

separated by a character other than a number or letter. In general, spaces will be used to delimit identifiers.

The CBASIC compiler is invoked as follows:

CBASIC <filename> [$<toggle> {<toggle>}]

where filename is the name of the source file. A file type of BAS is assumed by the compiler. Compiler toggles are discussed in Chapter 13.

The compiler produces an intermediate file in the CBASIC machine language with the same name as the source program but of type INT. The source program may be listed on the output device with any error messages following each line of the program. If errors are detected during compilation the source file must be corrected using the text editor. The program is then recompiled. If no errors occur during compilation, the intermediate file may be executed by typing the command:

CRUN <filename>

If errors are found during execution, the program must be corrected and recompiled.

The following are examples of CBASIC line numbers:

```
100
100.2
100E21
```

2.4 REM Statement

[<line number>] REM [<string terminated with CR>]

[<line number>] REMARK [<comment string>]

A REM statement is ignored by the compiler and compilation continues with the statement following the next carriage return. The REM statement may be used to document a program. REM statements do not affect the size of the program that may be compiled or executed. An unlabeled REM statement may follow any statement on the same line. The line number of a remark may occur in a GOTO, GOSUB, IF, or ON statement.

Examples of REM statements follow:

```
REM THIS IS A REMARK
remark This is also a remark
tax = 0.15 * income   rem lowest tax rate
```

The final example shows a REM statement on the same line with another statement. When using the REM statement in this manner a colon is not required between the two statements. In all other cases involving multiple statements on a line the colon must be present. In addition the REM must be the last statement on the line.

2.5 Executing a CBASIC Program

Execution of a CBASIC program consists of three steps. First the source program must be created on disk. Next the program is compiled by executing the CBASIC compiler with the name of the source program provided as a parameter. Finally the intermediate (INT) file created by the compiler may be interpreted by executing the run-time program, again using the source program name as a parameter.

Creation of the source program will normally be accomplished using CP/M's text editor, and must have a file type BAS. Each line of a source program is terminated by a carriage return. The line may be any length, however, the compiler listing will only print the first 80 characters of each line. When typing source programs, identifiers (variable names, reserved words, and user-defined functions) must be spelled exactly and

Examples of valid numbers are:

1, 1.0, -99, 123456.789

1.993, .01, 4E12, 1.77E-9

1.5E+3 is equivalent to 1500

1.5E-3 is equivalent to .0015

3.3 Identifiers

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 numeric. All lower case letters appearing in an identifier are converted to upper case unless compiler toggle D is set. The period is useful for making variable names more meaningful by building them from several words separated by periods.

Examples of valid identifiers are:

A, B, c1, c1234

Payroll.Record, NEW.SUM.AMT

Answer$, file.name$, X$

Each variable has a value associated with it at all times during execution of a program. Initially numbers are zero and strings are null. A string variable does not have a fixed length associated with it. Rather, as different strings are assigned to the variable, the storage is dynamically allocated. The maximum length assigned to a string variable is 255 characters.

3.4 Variables and Subscripted Variables

The general form of a variable is:

<identifier> [(<subscript list>)]

The subscript list indicates that the variable is a subscripted variable and indicates which element of the array is being referenced.

A variable in CBASIC may represent either a number or a string depending on the type of the identifier. A variable which is an element of an array is a subscripted

3. FORMING EXPRESSIONS

3.1 Strings

A string is defined to be zero or more valid alphanumeric characters surrounded by quotation marks ("). Since a continuation character is treated as part of the string, strings defined as constants in the source program must be contained on one line. A carriage return may not be part of a string. Embedded quotation marks may be entered as two adjacent quotes.

The following examples demonstrate valid strings:

```
"123"

"May 24, 1944"

"Enter your name please"

"""Look, look"", said Tom"
```

In the final example the string is:

```
"Look, look", said Tom
```

3.2 Numbers

A number may be in fixed format or exponential notation. In either case it may contain 1 to 14 digits, a sign, and a decimal point. In exponential notation the exponent is of the form Esdd, where 's', if present, is a valid sign (+, -, or blank) and where 'dd' is one or two valid digits. The sign is the sign of the exponent and should not be confused with the optional sign of the mantissa. The numbers range from 1.0E-64 to 9.9999999999999E62.

Although only 14 significant digits are maintained more digits may be included in a number used with CBASIC. If necessary the value is rounded to 14 digits.

3.5 Expressions

Expressions consist of algebraic combinations of function references, variables, constants, and operators. They evaluate to either a numeric or string value. The hierarchy of operators is:

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

String variables may only be operated on by relational operators and the concatenation operator. Mixed string and numeric operations are not permitted. The mnemonic relational operators (LT, LE, etc.) are interchangable with the symbolic operators (<, <=, etc.).

Examples of expressions:

```
amount * tax

cost + overhead * percent

a*b/c(1.2+xyz)

last.name$ + ", " + first.name$
```

Relational operators result in a 0 if false and a -1 if true. NOT, AND, OR, and XOR are performed on a signed 16-bit binary representation of the rounded integer portion of the variable. The result is then converted to a floating point number.

If the integer is greater than 32,768 it cannot be represented by a 16-bit binary number. Logical operations on such a number will give unpredictable results.

Examples of logical operations:

```
12 AND 3 = 0      12 AND 5 = 4

NOT -1   = 0      NOT 3   = -4

12 OR 3 = 15      12 OR 5 = 13

12 XOR 3 = 15     12 XOR 5 = 9
```

variable. Subscripted variables must appear in a DIM statement before being used as a variable.

Examples of variables are:

```
X$
PAYMENT
date.of.deposit$
```

The following examples show subscripted variables:

```
y$(i,j)

INCOME(AMT(CLIENT),CURRENT.MONTH)
```

When subscripts are calculated a check is made to insure that the element resides in the referenced array.

A DIM statement is an executable statement and each execution will allocate a new array. The general form is:

```
[<line number>] DIM <identifier> (<subscript list>)
                {,<identifier> (<subscript list>)}
```

The dimension statement dynamically allocates space for number or string arrays. Elements of string arrays may be any length up to 255 bytes and change in length as they assume different values. Initially numeric arrays are set to zero and string array elements are null strings. An array must be dimensioned explicitly; no default options are provided. Arrays are stored in row-major order.

The subscript list is used to specify the number of dimensions and the extent of each dimension of the array being declared. The subscript list may not contain a reference to the array being dimensioned.

Expressions in subscript lists are evaluated as numbers and rounded to the nearest integer when determining the size of the array. All subscripts have an implied lower bound of zero.

Examples of DIM statements:

```
DIM A(10)

DIM ACCOUNT$(100),ADDRESS$(100),NAME$(100)

DIM B$(2,5,10)

DIM X(A(I),M,N)
```

4. CONTROL STATEMENTS

4.1 GOSUB Statement

[<line number>] GOSUB <line number>

[<line number>] GO SUB <line number>

The address of the next sequential instruction is saved. Control is then transferred to the statement labeled with the line number following the GOSUB.

Examples:

```
GOSUB 700

PRINT "BEFORE TABLE"
GOSUB 200   REM PRINT THE TABLE
PRINT "AFTER TABLE"
STOP
200  REM PRINT THE TABLE
     FOR INDEX = 1 TO 100
          PRINT INDEX
     NEXT INDEX
     RETURN
```

4.2 RETURN Statement

[<line number>] RETURN

The RETURN statement causes the execution of the program to return to the statement that immediately followed the most recently executed GOSUB.

If a return is executed without previously executing a GOSUB or ON...GOSUB an error occurs.

Examples:

```
500 RETURN

IF VALID THEN RETURN
```

3.6 Assignment Statements

<line number> [LET] <variable> = <expression>

The expression is evaluated and assigned to the variable appearing on the left side of the equal sign. The type of the expression, either a number or a string, must match the type of the variable.

Examples:

```
TIME=0

100  LET A = B + C

X(3,A) = 7.32 * Y + X(2,3)

SALARY = (HOURS.WORKED * RATE) - DEDUCTIONS

date$ = month$ + " " + day$ + ", " + year$
```

```
IF VALID THEN \
     PRINT MSG$(CURRENT.MSG) :\
     GOSUB 200 :\       UPDATE RECORD
     GOSUB 210 :\       WRITE RECORD
     NO.OF.RECORDS=NO.OF.RECORDS+1 :\
     RETURN

IF X > 3 THEN X = 0 : Y = 0 : Z = 0

IF YES=TRUE THEN PRINT MSG$(1) \
            ELSE PRINT MSG$(2)

IF TIME>LIMIT THEN \
     PRINT TIME.OUT.MSG$ :\
     BAD.RESPONSES=BAD.RESPONSES+1 :\
     QUESTION=QUESTION+1 \
ELSE \
     PRINT THANKS.MSG$ :\
     GOSUB 1000 :\   ANALYSE RESPONSE
     ON RESPONSE GOSUB \
          2000, 2010, 2020, 2030, 2040 :\
     RETURN
```

4.5 WHILE Statement

[<line number>] WHILE <expression>

Execution of all statements between the WHILE statement and its corresponding WEND is repeated until the value of the expression is zero. If the value is zero initially the statements between the WHILE and WEND will not be executed. Variables in the WHILE expression may change during execution of the loop.

Examples:

```
WHILE -1
     PRINT "X"
WEND

WHILE X > Z
     X = X-1
     PRINT X
WEND
```

4.3 GOTO Statement

[<line number>] GOTO <line number>

[<line number>] GO TO <line number>

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

Examples:

```
80   GO TO 35

GOTO 100.5
```

4.4 IF Statement

[<line number>] IF <expression> THEN <line number>

[<line number>] IF <expression> THEN <statement list>
[ELSE <statement list>]

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, when the expression is not equal to zero, an unconditional branch to the line number occurs.

A statement list is composed of one or more statements in which each pair of statements is separated by a colon (:) or a colon/backslash combination (:\). The colon is not required after the THEN nor is it required before or after the ELSE. It is only used to separate statements. An IF statement must be the first statement on a line; it may not follow a colon.

Examples:

```
IF ANSWER$="YES" THEN GOSUB 500

IF DIMENSIONS.WANTED THEN PRINT LENGTH, HEIGHT
```

4.7 FOR Statement

[<line number>] FOR <index> = <expression> TO
<expression> [STEP <expression>]

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 met when the index exceeds the value of the TO expression. If the step is negative, the index must be less than the value of the TO expression for the exit criteria to be satisfied.

The index must be an unsubscripted variable. It is initially set to the value of the first expression. Both the TO and STEP expressions are evaluated on each loop; all variables associated with these expressions may change within the loop. Additionally the index may be changed during execution of the loop. If the STEP clause is omitted, a default value of one is assumed. A FOR loop is always executed at least once.

Examples:

```
FOR X=1 TO 10
     SUM=SUM+VECTOR(X)
NEXT X

FOR POSITION=MARGIN+TABS TO PAPER.WIDTH STEP TABS
     PRINT TAB(POSITION);SET.TAB$;
NEXT POSITION
```

If a step of one is desired, the STEP clause should be omitted. The execution will be substantially faster since fewer run-time checks must be made.

```
TIME = 0
TIME.EXPIRED = FALSE
WHILE TIME < LIMIT
     TIME = TIME + 1
     IF INP(CONSOLE.STATUS.PORT) \
          AND STATUS.MASK THEN RETURN
WEND
TIME.EXPIRED = TRUE
RETURN

WHILE ACCOUNT.IS.ACTIVE
     GOSUB 100   REM ACCUMULATE INTEREST
WEND

WHILE FILE.EXISTS
     WHILE TRUE
          IF ARG$ = ACCT$ THEN \
               ACTIVITY = TRUE :\
               RETURN
          IF ARG$ < ACCT$ THEN \
               ACTIVITY = FALSE :\
               RETURN
          GOSUB 3000    REM READ ACCT REC
     WEND
WEND
ACTIVITY = FALSE
RETURN

WHILE TRUE
     INPUT LINE STRING$
     IF STRING$ = CONTINUE$ THEN RETURN
WEND
```

4.6 WEND Statement

[<line number>] WEND

A WEND statement denotes the end of the closest unmatched WHILE statement.

Examples:

```
WHILE TRUE
     GOSUB 100      REM ENTER PROCESS DESIRED
     GOSUB 110      REM TRANSLATE PROCESS TO NUMBER
     IF PROCESS.DESIRED=0 THEN RETURN
     IF PROCESS.DESIRED>6 THEN \
          GOTO 400       REM ERROR
     ON PROCESS.DESIRED GOSUB \
          1000, \      ADD A RECORD
          1010, \      ALTER NAME
          1020, \      UPDATE QUANTITY
          1030, \      DELETE A RECORD
          1040, \      CHANGE COMPANY CODE
          1050     REM GET PRINTOUT
WEND
```

4.10 STOP Statement

[<line number>] STOP

When a STOP statement is encountered program execution terminates. All open files are closed, the print buffer is emptied and control returns to the host system. Any number of STOP statements may appear in a program.

A STOP statement is appended to all programs by the compiler.

Examples:

```
400  STOP

     IF STOP.REQUESTED THEN STOP
```

4.11 RANDOMIZE Statement

[<line number>] RANDOMIZE

A RANDOMIZE statement initializes the random number generator. It must be issued after an INPUT statement to have any effect on the sequence of random numbers.

Examples:

```
450 RANDOMIZE
```

4.8 NEXT Statement

```
[<line number>] NEXT [<identifier> {,<identifier>}]
```

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.

The following example of nested FOR loops shows the use of a list of identifiers:

```
FOR I = 1 TO 10
     FOR J = 1 TO 20
          X(I,J) = I + J
NEXT J, I
```

4.9 ON Statement

```
[<line number>] ON <expression> GOTO
          <line number> {, <line number>}

[<line number>] ON <expression> GOSUB
          <line number> {, <line number>}
```

The expression, rounded to the nearest integer value, 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 statement 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.

The keywords GOTO and GOSUB may alternately be coded as GO TO and GO SUB.

Examples:

```
PRINT

PRINT AMOUNT.PAID

PRINT TAB(MARGIN+OFFSET);QUANTITY,PRICE,QUANTITY*PRICE

PRINT "TODAY'S DATE IS: ";MONTH$;" ";DAY;", ";YEAR
```

5.3 LPRINTER Statement

[<line number>] LPRINTER [WIDTH <expression>]

After execution of the LPRINTER statement all PRINT statement output normally directed to the console will be output on the list device. The expression following the optional WIDTH must be numeric and will be used to set the line width of the list device.

If the width option is not present the previously set width is used. Initially the width is set to 132. An error occurs if the width is less than 1 or greater than 133.

Examples:

```
500 LPRINTER

IF HARDCOPY.WANTED THEN LPRINTER WIDTH 120

LPRINTER WIDTH REQUESTED.WIDTH
```

5.4 CONSOLE Statement

[<line number>] CONSOLE

Execution of the CONSOLE statement restores printed output to the console.

Examples:

```
490 CONSOLE
```

5. INPUT/OUTPUT STATEMENTS AND FUNCTIONS

5.1 General Information

This chapter discusses input and output statements and functions. File accessing statements are discussed in chapter 10; formatted printing is explained in chapter 9.

CBASIC buffers output and only prints an output line when it has been completely formed. The output from print statements is collected in the "print buffer". The print buffer has a pointer associated with it which locates the next available position in the buffer.

Input from the console is read a line at a time instead of a character at a time. This allows the user to take advantage of the CP/M line editing functions.

5.2 PRINT Statement

[<line number>] PRINT <expression> <delim>

The PRINT statement outputs the value of each expression on the console unless an LPRINTER statement (described below) is in effect. In the latter case output is directed to the line printer. If the length of a numeric item would cause the right margin of the print buffer to be exceeded the buffer is printed and the current number begins a new line. Strings are output until the buffer is full and then the remainder of the string, if any, is output on the next line.

The <delim> between expressions may be either a comma or a semicolon. The comma causes automatic spacing to the next column that is a multiple of 20. If this spacing results in a print position greater than the currently specified width, the print buffer is output and the print position is set to one. A semicolon causes one blank to be output after a number and no spacing to occur after a string.

Partial lines resulting from a PRINT whose list ends in a comma or semicolon are not output until another PRINT whose list does not end in a <delim> is executed, or the line width is exceeded, an LPRINTER or CONSOLE is executed, or the program terminates. A PRINT with no expression will dump the print buffer; a blank line is printed if the buffer is empty.

5.7 READ Statement

[<line number>] READ <variable> {, <variable>}

A READ statement assigns values from DATA statements to the variables following the READ. DATA statements are processed sequentially as they appear in the program. An attempt to read past the end of the last DATA statement produces an error.

Examples:

```
READ NAME$,AGE,EMPLOYER$,SSN

FOR PROD.NO = 1 TO NO.OF.PRODUCTS
     READ PRODUCT.NAME$(PROD.NO)
NEXT PROD.NO
```

5.8 DATA Statement

[<line number>] DATA <constant> {, <constant>}

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. A DATA statement must be the only statement on a line and it must not be continued with a continuation character.

Examples:

```
400 DATA 332.33, 43.0089E5, "ALGORITHM"

DATA ONE, TWO, THREE, 4, 5, 6
```

5.9 RESTORE Statement

[<line number>] RESTORE

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 a RESTORE statement is to allow rereading the constants contained in DATA statements.

```
IF END.OF.PAGE THEN \
     CONSOLE :\
     PRINT USING "##,### WORDS ON THAT PAGE";WORDS :\
     INPUT "INSERT NEW PAGE, THEN RETURN";LINE TRASH$ :\
     LPRINTER
```

The width of the console device may be changed with the POKE statement (Chapter 6). The console width is one byte at location 272 (110H). The new console width will become effective at the next execution of the CONSOLE statement. The console line width is normally 80.

5.5 POS Predefined Function

POS

POS returns the current position of the output line buffer pointer. This value will range from 1 to the line width currently in effect.

Examples:

```
PRINT  "THE PRINT HEAD IS AT COLUMN: ";POS

IF WIDTH.LINE-POS < 15 THEN PRINT
```

5.6 TAB Predefined Function

TAB (<expression>)

TAB positions the output buffer pointer to the position specified by the value of the expression rounded to the nearest integer. If the value of the expression is less than or equal to the current print position the print buffer is dumped and the buffer pointer is set as described above. The TAB function may only be used in PRINT statements.

If the expression, rounded to an integer, is greater than the current line width an error occurs.

Examples:

```
PRINT TAB(15);"X"

PRINT "THIS IS COL. 1";TAB(50);"THIS IS COL. 50"

PRINT TAB(X+Y/Z);"!";TAB(POS+OFFSET);

PRINT TAB(LEN(STR$(NUMBER)));NUMBER
```

Examples:

```
INPUT "ENTER ADDRESS";LINE ADDR$

INPUT "TYPE RETURN TO CONTINUE";LINE X$
```

Prompt strings are directed to the console even when an LPRINTER statement is in effect.

If a null prompt string is used the question mark can be suppressed but one blank will still be printed.

5.11 OUT Statement

[<line number>] OUT <expression> , <expression>

The low-order eight bits of the integer portion of the second expression is sent to the 8080 machine output port selected by the integer portion of the first expression.

Both arguments must be numeric and in the range 0 to 255 for the results to be meaningful.

Examples:

```
OUT 1,58

OUT FRONT.PANEL,RESULT

IF X>5 THEN OUT 9,((X*X)-1)/2

OUT TAPE.DRIVE.CONTROL.PORT,REWIND

OUT PORT(SELECTED),ASC("$")
```

5.12 INP Predefined Function

INP (<expression>)

INP returns the value input from the 8080 I/O port specified by the expression. This function is useful for accessing peripheral devices directly from the CBASIC program.

The argument is rounded to the nearest integer and for the results to be meaningful it must be in the range of 0 to 255.

Examples:

```
500 RESTORE

IF END.OF.DATA THEN RESTORE
```

5.10 INPUT Statement

```
[<line number>] INPUT [<prompt string> ;]
          <variable> {, <variable>}
```

If the prompt string is present it is printed on the console otherwise a question mark is output. In both cases a blank is then printed and 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 entered at the console must be separated by commas and are terminated by a carriage return. Strings may be enclosed in quotation marks in which case commas and leading blanks may be included in the string.

If too many or too few data items are entered, a warning is printed on the console and the entire line must be reentered.

Examples:

```
INPUT RESPONSE$

INPUT "WHAT FILE, PLEASE?";FILE.NAME$

INPUT "ENTER YOUR NAME AND NUMBER";NAME$,NUMBER

INPUT "";ZIP.CODE
```

A special type of INPUT statement is the LINE INPUT. The general form of this statement is:

```
[<line number>] INPUT [<prompt string> ;]
          LINE <variable>
```

This statement functions as described above with the following exceptions. Only one variable is permitted and it must be of type string. Any data entered from the console is accepted and assigned to the variable. The data is terminated only by a carriage return. A null string may be entered by responding to the prompt with a carriage return.

6. MACHINE LANGUAGE LINKAGE STATEMENTS AND FUNCTIONS

6.1 PEEK Predefined Function

PEEK (<expression>)

The PEEK function returns the contents of the memory location given by the expression. The value returned ranges from 0 to 255. The memory location must be within the address space of the computer being used for the results to be meaningful.

The expression must be numeric and is rounded to the nearest integer.

Examples:

```
100 MEMORY=PEEK(1)

FOR INDEX = 1 TO PEEK(BUFFER)
     IN.BUFFER$(INDEX)=CHR$(PEEK(BUFFER+INDEX))
NEXT INDEX
```

6.2 POKE Statement

[<line number>] POKE <expression> , <expression>

The low-order eight bits of the the second expression is stored at the memory address selected by the first expression. The first expression must evaluate to a valid address for the computer being used.

Both expressions must be numeric; they are rounded to the nearest integer.

Examples:

```
750 POKE 1700,ASC("$")

FOR POSIT=1 TO LEN(OUT.MSG$)
     POKE MSG.LOC+POSIT,ASC(MID$(OUT.MSG$,POSIT,1))
NEXT POSIT
```

Examples:

```
PRINT INP(ADDR)

IF INP(255) > 0 THEN PRINT CHR$(7)

ON INP(INPUT.DEVICE.PORT) GOSUB \
     100, 200, 300, 400, 400, 400, 500
```

7. PREDEFINED FUNCTIONS

7.1 Numeric Functions

The following functions return numeric values. Arguments, when required, must be expressions that evaluate to valid floating point numbers.

FRE

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

```
X=FRE

IF FRE < 1500 THEN GOSUB 10
```

ABS(arg)

ABS returns a value that is the absolute value of the argument, represented by arg. If arg is greater than or equal to zero the returned value is arg, otherwise the returned value is -arg.

```
DISTANCE=ABS(START-FINISH)

IF ABS(DELTA.X) <= LIM THEN STOP
```

INT(arg)

INT returns the integer part of the variable. The fractional part is truncated.

```
TIME=INT(MINUTES)+INT(SECONDS)

IF (X/2)-INT(X/2)=0 THEN PRINT \
     "EVEN" ELSE PRINT "ODD"
```

RND

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

When using RND a seed is necessary to avoid identical sequences of numbers each time the program is executed. The RANDOMIZE statement is used to seed the random number generator in CBASIC. The time taken by an operator to respond to an INPUT statement is used to set the seed. This time will

6.3 CALL Statement

[<line number>] CALL <expression>

The CALL statement is used to link to a machine language subroutine. The expression is the address of the subroutine being referenced. This value must be within the address space of the computer being used. Control is returned to the CBASIC program by executing a 8080 RET instruction. The hardware registers may be altered by the subroutine and need not be restored prior to returning.

The expression must be numeric and is rounded to the nearest integer.

Examples:

```
CALL 5

2000 CALL ANALOG.INPUT

WHILE PEEK(PARAMETER) <> 1
     CALL GET.RESPONSE
WEND
RETURN
```

Arguments may be passed to machine language subroutines with the POKE and PEEK instructions.

User-written assembly language programs may be used with CBASIC by generating a CP/M system smaller than the available memory and loading the machine code in the area which is now available above CP/M. Load the programs using CP/M's DDT program prior to executing CRUN. All the memory below (lower address than) CP/M is used by CBASIC and is not available to store subroutines.

LOG(arg)

The natural logarithm of the argument is returned by LOG.

```
BASE.TEN.LOG=LOG(X)/LOG(10)

PRINT "LOG OF X IS ";LOG(X)
```

SIN(arg)

SIN returns the sine of the arg. The argument is expressed in radians.

```
FACTOR(Z)=SIN(A - B/C)

IF SIN(ANGLE)=0 THEN PRINT "HORIZONTAL"
```

SQR(arg)

SQR returns the square root of the arg. If arg is negative, a warning message is printed and the square root of the absolute value of arg is returned.

```
HYPOT=SQR((SIDE1^2)+(SIDE2^2))

PRINT USING \
     "THE SQUARE ROOT OF X IS: #####.###"; \
     SQR(X)
```

TAN(arg)

TAN returns the tangent of arg. The argument is expressed in radians.

```
TANGENT=TAN(X)

Q=TAN(X - 3*COS(Y))
```

7.2 String Functions

ASC(arg$)

ASC returns the ASCII numeric value (in decimal) of the first character of the string argument. If the length of the string is zero (null string) an error will occur.

```
IF ASC(DIGIT$)>47 AND ASC(DIGIT$)<58 THEN \
     PRINT "VALID DIGIT"

OUT PORT,ASC("*")
```

vary with each execution of a program. Therefore, for RANDOMIZE to work correctly, it must be preceded by an INPUT statement.

```
DIE=INT(RND*6)+1

IF RND > .5 THEN PRINT \
     "HEADS" ELSE PRINT "TAILS"
```

SGN(arg)

SGN returns a numeric value that represents the algebraic sign of the argument. It will return -1 if arg is negative, 0 if arg is zero, and +1 if arg is greater than zero.

```
IF SGN(X) THEN PRINT "X=0"

IF SGN(BALANCE)=-1 THEN PRINT "OVERDRAWN"
```

ATN(arg)

ATN returns the arctangent of arg. Other inverse trigonometric functions may be computed from the arctangent. The argument is expressed in radians.

```
X=ATN(RADIANS)

TEMPERATURE=K+N(L)/ATN(X)
```

COS(arg)

COS returns the cosine of arg. The argument is expressed in radians.

```
IF COS(ANGLE)=0 THEN PRINT "VERTICAL"

PRINT CONSTANT*COS(X)
```

EXP(arg)

EXP returns the value of the constant "e" raised to the power given by arg.

```
Y=A*EXP(BX)

E=EXP(1)
```

4) The above characters will match only themselves if immediately preceded by a backslash (\).

Examples:

```
match("is","Now is the",1) returns 5

match(" ##","August 9, 1974",1) returns 10

match("a?","character",4) returns 5

match("\#","123#45",1) returns 4
```

The following program may be used to experiment with the match function.

```
 1: TRUE=-1
 2: FALSE=0
 3: edit$=" The number of occurrences is ###"
 4: while TRUE
 5:         input "enter object string";line object$
 6:         input "enter argument string";line arg$
 7:         gosub 620
 8:         print using edit$;occurrence
 9: wend
10:
11: 620    rem-----count occurrences----------
12:        location=1
13:        occurrence=0
14:        while TRUE
15:            location= match (arg$,object$,location)
16:            if location=0 then RETURN
17:            occurrence=occurrence+1
18:            location=location+1
19:        wend
20: END
```

MID$(object$,start,len)

MID$ returns a string consisting of the n characters of object$ starting at the mth character. The value of m is equal to start rounded to the nearest integer while n is len rounded to the nearest integer. Object$ must evaluate to a string. If start is greater than the length of object$ a null string is returned. If len is greater than the length of object$ all the characters from start to the end of object$ are returned. An error occurs if start or len is negative.

```
DIGIT$=MID$(OBJECT$,X,1)

PRINT MID$("MONTUEWEDTHUFRISATSUN",DAY,3)
```

CHR$(arg)

CHR$ returns a one character string consisting of the character whose ASCII equivalent is the integer part of arg. CHR$ can be used to send control characters to an output device. For instance the statement "PRINT CHR$(10)" will output a line feed to the console.

```
IF CHR$(INP(IN.PORT)) = "A" THEN GOSUB 100

PRINT CHR$(BELL)+CHR$(FORM.FEED)
```

LEFT$(object$,len)

LEFT$ returns a string consisting of the first len characters of object$. If len is greater than the length of object$, the entire string will be returned. If len is zero a null string will be returned, if len is negative an error will occur.

```
PRINT LEFT$(INPUT.DATA$,25)

IF LEFT$(IN$,1)="Y" THEN GOSUB 400
```

LEN(arg$)

LEN returns the length of arg$. Zero is returned if arg$ is a null string.

```
IF LEN(TEMPORARY$)>25 THEN PRINT "TOO LONG"

FOR X=1 TO LEN(OBJECT$)
     PRINT X
NEXT X
```

MATCH(pattern$,object$,start)

MATCH returns the position of the first occurrence of pattern$ in object$ starting with the character position given by start. A zero will be returned if no match is found. The following pattern matching features are available:

1) A pound sign (#) will match any digit (0-9).

2) An exclamation mark (!) will match any upper or lower case letter.

3) A question mark (?) will match any character.

exist in the CP/M file directory. The RENAME facility will allow a CBASIC program to use the following backup convention:

1. The output file is opened with a filetype of '$$$' indicating that it is temporary.

2. Any file with the same name as the output file but with a type 'BAK' is deleted.

3. Data is written to the temporary file as the program does its processing.

4. At the end of processing, the program renames any file with the same filename and filetype as the output file to the same filename but with the filetype 'BAK'.

5. The program renames the temporary output file to the proper name and type.

SIZE(filename$)

SIZE returns the size in blocks of the file specified by filename$. If the file is empty or does not exist zero is returned. Filename$ may be any CP/M ambiguous filename.

Examples:

```
size("NAMES.BAK")

SIZE(COMPANY$+DEPT$+".NEW")

size("B:ST?RTR?K.*")

SIZE("*.*")

size("*.BAS")
```

The SIZE function returns the number of blocks of diskette space consumed by the file or files referred to by the argument. When the operating system allocates diskette space to a file it does so in one block increments. A file of 1 character (1 byte) will consume a full block of space. This means the SIZE function returns the amount of space that has been reserved by the file rather than the size of the data that is in the file.

This function is useful in a program that must duplicate or construct a file on disk. If the program knows that it will create a file of a given size, possibly dependent on the size of its input file, it can first determine whether or not there is

RIGHT$(object$,len)

RIGHT$ returns the n rightmost characters of object$. The value of n is equal to len rounded to the nearest integer. If len is negative an error occurs; if n is greater than the length of object$ the entire string is returned. Object$ must evaluate to a string.

```
IF RIGHT$(ACCOUNT.NO$,1)="0" THEN PRINT \
     "TITLE ACCT"

NAME$=RIGHT$(NAME$,LEN(NAME$)-LEN(FIRST.NAME$))
```

STR$(arg)

STR$ returns the character string which represents the value of the number arg.

```
PRINT STR$(NUMBER)

IF LEN(STR$(VALUE))>5 THEN ED$="#######"
```

VAL(arg$)

VAL converts arg$ into a floating point number. Conversion continues until a character is encountered that is not part of a valid number or until the end of the string is encountered.

If arg$ is a null string or the first character of arg$ is not a +, -, or digit zero is returned.

```
PRINT ARRAY$(VAL(IN.STRING$))

ON VAL(ACCOUNT$) GOSUB 10, 20, 30, 40, 50
```

7.3 Disk Functions

RENAME(newname$,oldname$)

RENAME changes the name of the file selected by oldname$ to the name given by newname$. Renaming a file to a name that already exists produces an error.

The RENAME facility of CBASIC is implemented as a function call although at the present time the returned value is always a 255. A file must be closed before it is renamed otherwise, when CBASIC automatically closes files at the end of processing, it will attempt to close the renamed file under the name with which it was opened. This will cause an error because the original file name will no longer

8. USER DEFINED FUNCTIONS

8.1 Function Names

The name of a user defined function must begin with 'FN' followed by any combination of numbers, letters or periods. A function name may be up to 31 characters long. No spaces are allowed between the FN and the remainder of the name however a period is allowed. If the function returns a string value the name must end with a dollar sign. The name is used in defining and referencing the function.

Examples:

```
FN.THIS.IS.A.VALID.FUNCTION

FN3.1416

FNFUNCTION$

FN.TIMES

FN.TRUNCATE$
```

8.2 Function Definition

A function must be defined before it is referenced. Functions are defined with the DEF statement whose general form is:

```
{<line number>] DEF <function name>
     [(<dummy arg list>)] = <expression>
```

The type of the expression must match the type of the function name. There may be none or any number of dummy arguments specified and they may be used freely within the expression. A dummy argument is either a string or numeric variable. When there is more than one argument they are separated by commas. The type of the dummy arguments is independent of the function type. The dummy variables are local to the function definition. Variables of the same name in other portions of the program remain unaffected by the use of the function. Variables, constants and other functions may also be referenced in the expression. Recursive calls are not permitted.

sufficient free space on the disk before building the new file. For example, consider a program which reads a file named 'INPUT' from drive A, processes the data, and then writes a file named 'OUTPUT' to drive B. The size of 'OUTPUT' will be 125% of 'INPUT'. The following program segment will insure that space is available on disk B prior to processing.

```
70   rem------test for enough room------
     size.of.input=size("A:INPUT")
     size.of.output = int((size.of.input*1.25)+.5)
     free.space = 240 - size("B:*.*")
     if free.space < size.of.output then \
          enough.room = FALSE \
          else enough.room = TRUE
     return
```

CP/M supports 240 user accessible blocks on single density systems. The number of blocks in use subtracted from 240 gives the remaining space on the disk.

9. FORMATTED PRINTING

9.1 General

This chapter describes the PRINT USING statement. PRINT USING allows specification of printed output using a format string. A format string is composed of data fields and literal data. Data fields may be numeric or string; any character in the format string that is not part of a data field is a literal character. The general form of a PRINT USING statement is:

```
[<line number>] PRINT USING <format string> ;
     [<file reference>] <expression list>
```

The format string is any string expression. This allows the format to be determined dynamically during program execution. The expression list consists of expressions separated by commas or semicolons. The comma does not cause spacing as it does with the unformatted print. Each expression in the list is matched with a data field in the format string. If there are more expressions than fields in the format string, the format string is reused.

while searching the format string for a data field, the type of the next expression in the list, either string or numeric, determines what data field is used. For instance, when outputting a string and a numeric data field is encountered, the characters that make up the numeric data field will be treated as literal data. If there is no data field within the format string of the type required an error will occur.

A PRINT USING statement without the file reference causes an output line to be written to either the console or the line printer. The console is selected unless an LPRINTER statement is in effect. If the file reference is present the line is composed as it would be if the output was being printed on a list device. The entire line is written as a record in the selected file. Chapter 10 discusses the use of PRINT USING with disk files in more detail.

Examples:

```
DEF FN25 = RND*25

DEF FNLEFT.JUSTIFY(A$,LENGTH)=LEFT$(A$+"          ",LENGTH)

DEF FN.HYPOT(SIDE1,SIDE2)= \
     SQR((SIDE1*SIDE1) + (SIDE2*SIDE2))

DEF FN.FUEL.USE(MILES)=SPEED*FN.CONSTANT*MILES+OVERHEAD

DEF FN.EOJ=FLAG1 OR FLAG2 OR FLAG3 OR FLAG4

DEF FNINPUT(PORT)=INP(PORT) AND STATUS.MASK(PORT)
```

8.3 Function Reference

The user defined function call may be used in any expression. The same number of parameters must be specified in the call as are defined in the DEF statement. Parameters may be any valid expression but they must match the type of those specified in the definition. During evaluation, the current value of each expression is substituted for the dummy variable in the definition.

Examples:

```
PRINT FN.A(FN.A(X))

IF LEN(FNS("INPUT DATA",X$,Q)) < LIMIT THEN GOSUB 100

WHILE FN.ALTITUDE > MINIMUM.SAFE
     CURRENT.ALTITUDE=INP(ALTIMETER)
WEND
```

```
COMPANY$ = "SMITH INC."
PRINT USING "& &"; "THIS REPORT IS FOR",COMPANY$
```

will output:

```
THIS REPORT IS FOR SMITH INC.
```

A string may be right justified within a fixed field using the variable string field. The following program segment shows how this would be done:

```
FIELD.SIZE = 20
BLANK$ = "                    "
PHONE$ = "707-642-4824"
PRINT USING "#&"; RIGHT$(BLANK$ + PHONE$, FIELD.SIZE)
```

which would output:

```
#        707-642-4824
```

In the above example the pound sign is used as a literal character since the print list contains only a string expression. A pound sign may also indicate a numeric data field as will be shown in the next section.

9.5 Numeric Data Fields

A numeric data field is specified by a pound sign (#) to indicate each digit required in the resulting number. One decimal point may also be included in the field. Values are rounded to fit the data field. Leading zeros are replaced with blanks. If the number is negative a minus sign is printed to the left of the most significant digit. A single zero is printed on the left of the decimal point if the number is less than 1 and a position is provided in the data field. The following example illustrates the use of numeric data fields.

```
X = 123.7546
Y = -21.0
FOR$ = "####.####   ###.#   ###"
PRINT USING FOR$; X, X, X
PRINT USING FOR$; Y, Y, Y
```

Execution of the above program produces the following printout:

```
123.7546   123.8   124
-21.0000   -21.0   -21
```

9.2 String Character Field

A one character string data field is specified with an exclamation point. The first character of the next expression in the print statement list is output. For example:

```
F.NAME$ = "JOHN" : M.NAME$ = "SMITH"
PRINT USING "!. "; F.NAME$,M.NAME$
```

would output:

```
J. S.
```

In the example the period is treated as literal data. Since there are two expressions in the list the format string is reused to output the second expression.

9.3 Fixed Length String Fields

A fixed length string data field of more than one position is specified by a pair of slashes (/) separated by zero or more characters. The width of the field is equal to the number of characters between the slashes plus two. Any character may be placed between the slashes; these fill characters are ignored.

A string expression from the print list is left justified in the fixed field and if necessary padded on the right with blanks. A string which is longer than the data field is truncated on the right. For example:

```
FOR1$ = "THE PART REQUIRED IS /...5....0....5/"
PART.DESCRP$ = "GLOBE VALVE, ANGLE"
PRINT USING FOR1$; PART.DESCRP$
```

will output:

```
THE PART REQUIRED IS GLOBE VALVE, ANG
```

The use of the periods and numbers between the slashes makes it easy to verify that the data field is 16 characters long.

9.4 Variable Length String Fields

A variable length string field is specified with an ampersand (&). This results in a string being output exactly as it is defined. For example:

Asterisk fill of a numeric data field is accomplished by appending two asterisks to the beginning of the data field. A floating dollar sign may be obtained by appending two dollar signs to the field in a similar manner. Exponential format may not be used with either asterisk fill or the floating dollar sign. The pair of asterisks or dollar signs are included in the count of digit positions available for the field and they appear in the output only if there is sufficient space for the number and the asterisk or dollar sign. The dollar sign is suppressed if the value printed is negative. For example:

```
COST = 8742937.56
PRINT USING "**###,######.##    "; COST, -COST
PRINT USING "$$##,######.##    "; COST, -COST
```

prints:

```
**8,742,937.56   *-8,742,937.56
 $8,742,937.56    -8,742,937.56
```

A number may be output with a trailing sign instead of the leading sign if the last character in the data field is a minus sign. If the number is positive a blank replaces the minus sign in the printed result. For example:

```
PRINT USING "###- ###^^^^- "; 10, 10, -10, -10
```

will output:

```
10  100E-01   10- 100E-01-
```

If a minus sign is the first character in a numeric data field the sign position is fixed as the next output position. If the number being printed is positive a blank is output otherwise a minus sign is printed. The following example demonstrates this feature.

```
PRINT USING "-#####   "; 10, -10
```

which outputs:

```
 10  -  10
```

Any time a number will not fit within a numeric data field without truncating digits before the decimal point a percent sign is printed followed by the number in the standard format.

Numbers may be printed in exponential format by appending one or more uparrows (^) to the end of the numeric data field. For example the following program segment:

```
X = 12.345
PRINT USING "#.###^^   "; X, -X
```

would output:

```
1.235E 01   -.123E 02
```

The exponent is adjusted so that all positions represented by the pound signs are used. For instance:

```
PRINT USING "###.##^^^^"; 17.987
```

results in:

```
179.87E-01
```

Four positions are reserved for the exponent regardless of the number of uparrows used in the field.

If one or more commas appear embedded within a numeric data field the number is printed with commas between groups of three digits before the decimal point. For example:

```
PRINT USING "##,###   "; 100, 1000, 10000
```

prints:

```
100     1,000    10,000
```

Each comma which appears in the data field is included in the width of the field. Thus, even though only one comma is required to obtain embedded commas in the output, it is clearer to place commas in the data field in the positions they will appear on the output. For instance the following data fields will produce the same results except that the width of the first allows only 9 digits to be output. With the second 10 digits may be output.

```
#,#########

#,###,###,###
```

If the exponent option is used commas are not printed.

10. FILES

10.1 CP/M Files

CBASIC uses the CP/M file accessing routines to store and retrieve data from soft sectored IBM compatible diskette files. This section will provide a brief introduction to the file organization employed by CP/M. More detailed information is available in the CP/M manuals.

CP/M maintains a directory of File Control Blocks (FCB's) on each diskette. The FCB contains the file name, number of records in the file, and references to physical locations occupied by the data on the diskette. CP/M interfaces with the disk hardware through primitives that are used by transient programs, including CBASIC, to access files on disk. The primitives allow a file to be created, opened, closed, read or written. All data is processed in 128 byte segments. However, CBASIC maintains all necessary pointers and buffers data so the user is not restricted to 128 byte records.

Each CBASIC statement used to access files will now be discussed. Chapter 11 provides additional information on programming with files.

10.2 OPEN Statement

The OPEN statement activates an existing file for reading or updating. The general form of an OPEN statement is:

```
[<line number>] OPEN <expression> [RECL <expression>]
                AS <expression>
                {, <expression> [RECL <expression>]
                AS <expression>}
```

The first expression represents the name of a file on diskette. The name may contain an optional drive reference; if the drive reference is not present the currently logged drive is used. The file name must conform to the CP/M format for unambiguous file names. Lower case letters used in file names are converted to upper case. The following examples show valid file names:

9.6 Escape Characters

At times it may be desired to include a character as literal data which, following the above rules, would be part of a data field. This can be accomplished by "escaping" the character. A backslash (\) preceding any character causes the next character after the backslash to be treated as a literal character. This allows, for instance, a pound sign to precede a number as shown in the following example.

```
ITEM.NUMBER = 31
PRINT USING "THE ITEM NUMBER IS \# ##"; ITEM.NUMBER
```

which outputs:

```
THE ITEM NUMBER IS # 31
```

An escape character following an escape character causes a backslash to be output as a literal character.

All active files are automatically closed when a STOP statement is executed or a control-Z is entered in response to an INPUT statement. Files are not closed if a control-C is entered from the console or if an error occurs.

Each expression must be numeric and, after rounding to the nearest integer, be in the range 1 to 20.

Examples:

```
800 CLOSE FILE.NO

CLOSE NEW.MASTER.FILE,OLD.MASTER.FILE,UPDATE.812

FOR X = 1 TO NO.OF.WORK.FILES
     CLOSE X
NEXT X
```

10.4 CREATE Statement

The CREATE statement is identical to an OPEN statement except that a new file is created on the selected drive. The general form of a CREATE statement is:

```
[<line number>] CREATE <expression> [RECL <expression>]
                AS <expression>
                {, <expression> [RECL <expression>]
                AS <expression>}
```

If a file with the same name is present the existing file will be erased and a new file created.

Examples:

```
1200 CREATE "NEW.FIL" AS 19

CREATE ACC.MASTER$ RECL MASTER.REC.LEN AS ACC.FILE.NO

CREATE "B:"+NAME$+LEFT$(STR$(CURRENT.WORK)+"    ",3) \
     AS CURRENT.WORK
```

```
ACCOUNT.MST
CBASIC.COM
B:INVENTOR.BAK
```

The third example shows a reference to drive B.

The directory on the selected drive is searched and the named file is opened. If the file is not found in the directory it is treated as if an end of file had been encountered during a read. See the IF END statement for information on end of file processing.

The AS expression assigns an identification number to the file being opened. This value is used in future references to the file. Each active file must have a unique number assigned to it. If the expression rounded to the nearest integer in not between 1 and 20 an error occurs.

If the optional RECL expression is present the file consists of fixed length records. The record length is rounded to the nearest integer and it must be greater than zero. A file may be accessed randomly or sequentially when a record length is specified, otherwise only sequential is allowed.

Twenty files may be active at one time. Buffer space for files is allocated dynamically. Therefore storage space may be conserved by opening files as they are required.

Examples:

```
555 OPEN "TRANS.FIL" AS 9

OPEN FILE.NAME$ AS FILE.NO

OPEN WORK.FILE.NAME$(CURRENT.FILE) \
     RECL WORK.LENGTH AS CURRENT.FILE
```

10.3 CLOSE Statement

The CLOSE statement deactivates an OPEN file; the file is no longer available for input or output operations. The general form of a CLOSE statement is:

[<line number>] CLOSE <expression> {, <expression>}

Each expression references an active file. The file is closed, the file number is released, and all buffer space used by the file is deallocated. Before the file may be referenced again it must be reopened. An error will occur if the specified file has not previously been opened with a CREATE, OPEN or FILE statement.

that does not exist will cause execution to continue as if an end of file had been encountered.

In the following example if the file MASTER.DAT does not exist on drive B control will be transferred to statement 500.5. After a successful OPEN an end of file during a read will cause execution to continue with statement 500.

```
IF END #MASTER.FILE.NO THEN 500.5
OPEN "B:MASTER.DAT" AS MASTER.FILE.NO
IF END #MASTER.FILE.NO THEN 500
```

10.7 FILE Statement

```
[<line number>] FILE <variable> [(<expression>)]
                {, <variable> [(<expression>)]}
```

A FILE statement opens a file if it is present on the referenced disk otherwise a file with the specified name is created. The variable contains the name of the file to be accessed. As each file is activated it is assigned the next unused file number starting with 1. If all 20 numbers are already assigned then an error occurs. If the expression enclosed in parentheses is present the value of the expression is rounded to the nearest integer and becomes the record length.

The variable must not be subscripted and it must be of type string. It may not be a literal or an expression using any operators.

Examples:

```
FILE NAME$

FILE FILE.NAME$(REC.LEN)
```

10.8 READ Statement

There are four forms of the READ statement which access data from disk files. Each of the four statements will be discussed in turn and then some general comments about reading from disk files will be made. The first two types of the READ statement access files in a manner analogous to using the READ to access data from DATA statements. The last two forms are similar to the INPUT LINE statement.

The general form of the sequential read is:

```
[<line number>] READ # <expression> ; <variable>
                {, <variable>}
```

10.5 DELETE Statement

The DELETE statement erases the active file or files referenced by each expression. The general form of a DELETE statement is:

[<line number>] DELETE <expression> {, <expression>}

Each expression, rounded to the nearest integer, must be in the range of 1 to 20. If the number is not currently assigned to an active file an error occurs.

Examples:

```
DELETE 1

DELETE FILE.NO, OUTPUT.FILE.NO

I=0
WHILE I < NO.OF.WORKFILES
     I = I + 1
     DELETE I
WEND
```

10.6 IF END Statement

The IF END statement allows the programmer to process an end of file condition on an active file. The general form of the IF END statement is:

[<line number>] IF END # <expression> THEN <line number>

When an end of file is detected on a file, one of two actions will take place. If an IF END statement has been executed for the file, control is transferred to the statement labeled with the line number following the THEN. If no IF END statement has been executed an error occurs.

Any number of IF END statements may appear in a program for a given file. The most recently executed IF END is the one that will be in effect.

The expression must be numeric and after rounding to the nearest integer be in the range 1 to 20.

Examples:

```
IF END # 7 THEN 500

IF END # FILE.NO THEN 100.1
```

An IF END statement may be executed prior to assigning the file number to a file. A subsequent OPEN on a file

The READ LINE statement permits CBASIC to access records containing ASCII data in any format on a line-by-line basis. For instance any file created with the CP/M text editor could be read a line at a time. In the following example:

```
READ # 12; LINE in.string$
```

all characters in the next record will be read until a carriage return followed by a line feed is encountered.

Additional examples follow:

```
READ # 12 ; LINE LINE.OF.TEXT$

READ # INPUT.FILE, RECORD; LINE NEXT.ONE$
```

In all READ statements to files the first expression rounded to the nearest integer must be in the range of 1 to 20 and represent an active file. The second expression, if present, is rounded to the nearest integer; it must be greater than zero.

10.9 PRINT Statement

There are four variations of the PRINT statement which output data onto disk files. Each of these will be discussed in this section. Both sequential and random files may be written using the following forms of the PRINT statement:

```
[<line number>] PRINT # <expression> ;
                <expression> {, <expression>}

[<line number>] PRINT # <expression> ,
                <expression> ;<expression> {, <expression>}
```

The first form of the PRINT statement outputs the next sequential record to the file specified by the first expression. Each of the expressions in the expression list will be written as a separate field separated by commas. String fields will be surrounded by quotation marks and the last field will be followed by a carriage return and a line feed.

The second form of the PRINT statement outputs a random record specified by the second expression to the disk file specified by the first expression. The same format as described above is used. The file must have been opened with a fixed record length. An error occurs if there is insufficient space in the record for all the data.

The above READ statement reads sequentially from the file specified by the first expression. The file will be read field by field into the variables until every variable has been assigned a value. Fields may be floating point or string values and are delimited by a comma.

Examples:

```
READ # 7; STRING$, NUMBER

READ # FILE.MASTER; NAME$, ADDRESS$,CITY$,STATE$
```

The general form of the next variation of the READ statement is:

```
{<line number>] READ # <expression> , <expression> ;
               [<variable>] {, <variable>}]
```

A random record specified by the second expression is read from the disk file specified by the first expression. The fields in the record are assigned to the variables in the variable list. An error occurs if there are more variables than fields in the record. The file must have been opened with the RECL option.

Examples:

```
READ # RANDOM.FILE,RELATIVE.REC.NO; NAME$, PAY, HOURS,\
     TERM.OF.EMPLOY,SSN$
```

The following two forms of the READ statement treat files as lines of text. The general form of the sequential variant is:

```
[<line number>] READ # <expression> ; LINE <variable>
```

This statement reads sequentially all data from the specified file until a carriage return followed by a line feed is encountered. All the data read up to but not including the carriage return and line feed is assigned to the single string variable specified in the READ LINE statement.

The random variant of the READ LINE has the following general form:

```
[<line number>] READ # <expression> , <expression> ;
          LINE <variable>
```

The final variation of the READ statement reads the record specified by the second expression off of the file specified by the first expression. The data is assigned to the string variable as described for the previous form of the READ.

If this procedure is executed, the result on file will be:

```
The "X-RAY MACHINE" is worth  $91,327.44crlf
```

The use of two adjacent double-quotes results in a single double-quote being printed.

10.10 Appending to a File

A file may be appended to by reading sequentially until the end-of-file is detected with IF END, then printing additional records.

An example of appending to a file is shown below:

```
     true = -1
     if end # 3 then 200
     open "master" as 3
     if end # 3 then 100
     while true
          read # 3; dummy
     wend
100  print # 3; "this added to end"
     stop
200  print "file not found"
```

Except for this case, sequential reading and printing should not be intermixed.

10.11 Re-Initializing the Disk System

If it becomes necessary to change diskettes during execution of a CBASIC program, CP/M must be given an opportunity to reallocate its internal diskette usage maps for the new diskette. If this is not done valid data may be overwritten. It is implemented in CBASIC as a machine language subroutine call to the decimal address 264. The following statement will re-initialize all disks:

```
call 264
```

The user program must close ALL files before this statement is executed.

Examples:

```
PRINT # 3; "JONES, BILL"

PRINT #FILE.NO; NAME$, ADDR$, SALARY

PRINT #PAY,EMPL.NO; EMPL.NAME(EMPL.NO),HOURS(EMPL.NO)

PRINT # 10, 55; DATE
```

Both forms of the PRINT statement discussed above produce files which may be read using the READ statement discussed in section 10.8. All values output to the file are delimited with commas or a carriage return line feed pair. In addition all strings are enclosed in quotation marks. If the data must be output in a specific format, such as when a report is being produced for later printing, the PRINT USING statement may be used with disk files. This type of the PRINT statement takes on the following general forms:

```
[<line number>] PRINT USING <expression> ;
          # <expression> ; <expression> { , <expression> }

[<line number>] PRINT USING <expression> ;
          # <expression> , <expression> ;
          <expression> { , <expression> }
```

These statements write data to files using the formatted printing options specified in the expression following the USING. Formatting options are described in Chapter 9 and are the same as those for console output. The first form is for sequential access and the second is used with random access. Records are delimited with a carriage return followed by a line feed.

The PRINT USING statement with disk files gives the programmer the same extensive facilities for formatting data that the USING clause permits when printing to the console or system list device; numbers can be formatted with commas and decimal points, asterisks and dollar-signs can be floated. Records can also be written to disk that contain embedded quotes and commas.

For example:

```
cents.wanted=TRUE
edit1$="$$##,###.##"
edit2$="$$##,###"
if cents.wanted then \
     edit$=edit1$ \
     else edit$=edit2$
print using "The ""&"" is worth "+edit$; \
     #file.no;product$,price
```

Because CBASIC reads each record on a field by field basis it is recommended that each record on a given file contain the same number of fields. If there is no information to fill any field of one record, either a zero or null string should be written. This will allow, for example, the fifth field of a sales transaction file to represent the amount of the sale, even if some or all of the first four fields are not used in a particular transaction.

Sometimes it is necessary to insure that a given field starts at the same relative position within a record. Usually, there will be some fields of fixed length and some fields of variable length. Numeric fields will always fall into the latter category unless the range of numbers is restricted. String fields, however, can always be made to be of fixed length by padding them with blanks.

For example:

```
string$ = left$(string$ + "                    ",20)
```

This will always produce a field that is 20 characters in length. By use of the STR$ function numbers can also be converted to strings and then padded. This will allow unrestricted numeric data to be of fixed length.

11.5 File Accessing Methods

An access method describes the order in which data is read from or written to a file. CBASIC supports two access methods, sequential and random. Either access method may be used on files that are organized as fixed. Only sequential may be used on a stream organized file.

11.6 Sequential Access

In sequentially accessed files there is one field of concern, the "next" field. The program cannot backtrack or skip ahead, it must proceed one field at a time.

A procedure to sequentially access a file and write it to the console is shown below. The file contains the following records:

```
"first field","second field","third"crlf
"","5","xxx123yyy"crlf
```

11. PROGRAMMING WITH FILES

11.1 File Facilities

The facilities available to the CBASIC user for accessing diskette files are extremely versatile, providing different file organizations and accessing methods. The emphasis of this chapter will be on the practical organization of files and the way in which they are accessed.

11.2 File Organization

The organization of a file describes the way it is represented on the diskette. All data written to files by CBASIC is in character format using the ASCII code. The contents of both string and numeric variables are written as their representative ASCII characters, not as binary data. This permits the use of both resident and transient CP/M commands with CBASIC data files.

Characters within CBASIC data files are organized into distinct groupings. The lowest level of grouping is called a field. A field can contain either string or numeric data. A string field is surrounded by- quotation marks ("). A numeric field is never surrounded by quotes and may contain any valid number as described in Chapter 3. Fields are separated from one another by either commas or a carriage return line feed pair.

CBASIC offers two file organizations, stream and fixed. These techniques are compatible to provide more flexibility for the programmer.

11.3 Stream Organization

When it is desired to store data sequentially, item by item, stream organization is used. Accessing is done on a strict field by field basis. There is no restriction on the values or lengths of data that may be written; each item of data takes only as much room as needed for data and delimiters. In other words there is no padding.

```
ED1$="&"
ED2$=" $$,###.##"
PRINT USING ED1$+ED2$+ED1$+ED2$;#17,TRANSACTION.NO; \
     "PRINCIPAL:",PRIN,"INTEREST:",INTEREST

PRINT USING "&";#PRINTER.FILE;" "      REM BLANK LINE

PRINT USING "/234567/";#WORK.FILE,REL.REC.NO;SORT.KEY$

IN$="X"
WHILE IN$<>""
     INPUT "ENTER DATA";LINE IN$
     PRINT USING "/...5....0....5....0../";#TEMP.FILE;IN$
WEND
CLOSE #TEMP.FILE
```

The READ LINE statement allows a file to be accessed as though there was one field per record. Any commas or quotes will be read as part of the data. Only a carriage return and line feed will be treated as the delimiter. In effect there is no field structure in a file accessed with the READ LINE. For example, if the following file exists:

```
"field one","two","3","","four"crlf
"five","six"crlf
```

and the following procedure is executed:

```
read #file.no;line string$
print string$
```

the data printed on the console would be:

```
"field one","two","3","","four"
```

This should be compared with the following statements:

```
read # file.no; string$
print string$
```

which would output:

```
field one
```

All quotes and commas are considered part of the data; the data does not include either the carriage return or the line feed. In effect the READ LINE redefines the field delimiter of a file from a comma to a carriage return and line feed pair.

For example:

```
create file.name$ recl 25 as file.no
a$="one"
b$="record two"
c$="3"
d$=""
e$="five"
f$="abc123def"
print #file.no;a$,b$
print #file.no;c$,d$,e$
print #file.no;f$
```

produces the following file:

```
"one","record two"     crlf
"3","","five"          crlf
"abc123def"            crlf
```

The record delimiter (crlf) always occupies the last two bytes of the record and must be included in the specified record length. The space between the record delimiter and the last valid field is padded with blanks.

A fixed file READ statement will always access a new record each time it is used.

```
while TRUE
     read #file.no;field$
     print field$
wend
```

will print on the console (using the data from the previous example):

```
one
3
abc123def
```

The fixed organization of files implies a well defined structure to the accessed data. The processing program can then decide the meaning of a given field by its relative position in a record rather than by the value of the data itself. This provides savings in processing time and programming effort.

Files that are organized as fixed provide fast and easy access to the individual fields within each record because all fields can be read in at one time. Fixed files may be reorganized by sorting on a key within each record. In addition, fixed files permit random access as described below.

easier to debug large programs if they are composed of numerous, small, individually tested routines.

The include directive allows the programmer to build a library of common routines which reduces programming time. System standards, such as I/O port assignments, can be put in included routines. If the programs are moved from one system to another the include routine must be changed and the programs recompiled.

Commonly used procedures, such as searches, validation routines, or input routines are candidates for include files. If many programs in a system access the same file, all file access commands, such as READ, PRINT or OPEN can be set up as separate include files. If the file definition needs a change it can be made in one common file instead of several application programs.

12.4 END statement

[<line number>] END

An END statement indicates the end of the source program. It is optional and, if present, it terminates reading of the source program. Any statements following the END statement are ignored.

Examples:

```
500 END

END
```

```
while TRUE
     read #file.no;field$
     print field$
wend
```

The output on the console would be:

```
first field
second field
third

5
xxx123yyy
```

While reading data from a file sequentially, the READ statement will consider a field completed when it encounters either a comma or a carriage return. Within the double quotes of a string field it is permissable to have embedded commas.

When accessing a stream file, every field on the file will be read once and none will be skipped. It is possible to read in more than one field with a single read statement.

For example:

```
while TRUE
     read #file.no;fielda$,fieldb$
     print fielda$,fieldb$
wend
```

would print the following on the console (using the file from the previous example):

```
first field          second field
third
5                    xxx123yyy
```

The same field organization is used when writing a stream file. Each variable specified in the PRINT statement produces a single field in the file. When more than one variable is output in a single PRINT statement, the corresponding fields will be delimited by commas. The last (or only) field written by each PRINT statement will be delimited by a carriage return and line feed.

Toggle C suppresses the generation of an INT file. Since the first compilation of a large program is likely to have errors, this toggle will provide an initial syntax check without the overhead of writing the intermediate file.

Toggle C is initially off.

Toggle D suppresses translation of lower case letters to upper case. For example, if toggle D is on, 'AMOUNT' will not refer to the same variable as 'amount'.

Initially toggle D is off.

Toggle E is useful when debugging programs. If this toggle is set, it will cause the run-time program to accompany any error messages with the CBASIC line number in which the error occurred. This will determine the exact line where the program has "blown-up". Toggle E will increase the size of the resultant INT file and therefore should not be used with debugged programs. Toggle E must be set in order for the TRACE option (section 13.4) to work.

Initially toggle E is off.

TOGGLE F will cause the compiled output listing to be printed on the system list device in addition to the system console. This provides a hardcopy of the compiled program. Even if the B toggle is set a complete listing is provided if toggle F is set.

Initially toggle F is off.

Toggle G will cause the compiled output listing to be written to diskette. The file containing the compiled listing has the same name as the BAS file with a type of LST. If toggles G and B are specified only errors will be output at the console but a disk file of the complete program will be made.

Initially toggle G is off.

fields containing their name, social security number, and rate of pay. The twenty records would be placed on the diskette file in employee number order using the sequential access method with a fixed organization. Then, when an application program needed the data on employee number 12, a random read would be issued for relative record number 12 and the proper data would be retrieved. The following program would access the file described above:

```
true = -1
open "employee.mst" recl 50 as 3
if end # 3 then 500.1
while true   rem loop until eof
     input "enter employee number"; employ.no
     read # 3, employ.no; name$,ssn$,pay
     print using "&'s pay rate is ###.##"; name$,pay
wend
500.1 stop
```

To recap, the READ statement used on a stream organized file will always access the next available field on the file regardless of the field length or which delimiter is used. In a fixed organization file each READ statement will access the next record, delimited by a carriage return and a line feed. PRINT statements function in a similar manner.

11.8 Special Features

The PRINT USING statement can be used to write data to files as well as to the console or printer. Its use and the format of its output is the same when writing to diskette as it is when writing to the console or printer. If the file is fixed, the single unquoted field written by each execution of the PRINT USING statement will be padded to the specified record length as with normal fixed files. The PRINT USING is well suited to text processing applications.

The following examples demonstrate the PRINT USING statement with files:

```
PRINT USING "&";#TEXT.FILE.NO;LINE.OF.TEXT$

PRINT USING "VELOCITY=#####.### KPH";#OUTPUT.FILE,TIME; \
     VELOCITY(TIME)
```

```
AMOUNT = 12.13
TIME = 45
PRINT TIME * AMOUNT
```

If the above program was compiled using the following command:

```
CBASIC TEST $E
```

and then executed with the trace option:

```
CRUN TEST TRACE 1,3
```

the following output would be produced:

```
AT LINE 0001
AT LINE 0002
AT LINE 0003
 545.85
```

The TRACE option functions only if the toggle E has been set on during compilation of the program.

The first number is used to specify the statement number where the trace is to begin. The second number specifies where the trace is to stop. If no statement numbers are given the entire program is traced; if only the first statement number is present, tracing starts at this point and continues to the end of the program.

12. COMPILER DIRECTIVES

12.1 Directive Format

Directives are used to control the action of the compiler. Except for the END statement, all directives begin with a percent sign (%). The percent sign must be in column one. Characters on the same line following the directive are ignored by the compiler.

12.2 %LIST and %NOLIST

%LIST

%NOLIST

These directives allow listing only selected portions of a program while it is being compiled. The listing control directives may be placed anywhere in a source program and may be used as many times as desired.

%LIST sets toggle B (chapter 13) on while %NOLIST resets toggle B.

The %LIST / %NOLIST options do not affect the diskette or printer listings.

12.3 %INCLUDE

%INCLUDE <filename>

The %INCLUDE directive causes the compiler to insert the file specified in the include statement in the source listing immediately following the %INCLUDE. The file name may contain a drive reference and must be of type BAS. Included statements will be indicated in listings with a equal sign (=) following the CBASIC assigned statement number. Includes may be nested four deep but they may not include themselves. For example:

```
%include b:readin
```

will include the file READIN.BAS from drive B.

Since the files incorporated with include directives are of type BAS they may be compiled separately. It is

EF

A number in exponential format was input with no digits following the E.

FD

A function name that has been previously defined is being redefined in a DEF statement.

FI

An expression which is not an unsubscripted numeric variable is being used as a FOR loop index.

FN

A function reference contains an incorrect number of parameters.

FP

A function reference parameter type does not match the parameter type used in the function's DEF statement.

FU

A function has been referenced before it has been defined.

IE

An expression used immediately following an IF evaluates to type string. Only type numeric is permitted.

IF

A variable used in a FILE statement is of type numeric where type string is required.

IP

An input prompt string was not surrounded by quotes.

IS

A subscripted variable was referenced before it was dimensioned.

IU

A variable defined as an array is used with no subscripts.

MF

An expression evaluates to type string when type numeric is required.

MM

Variables of type string and type numeric are combined in the same expression.

NI

A variable referenced by a NEXT statement does not match the variable referenced by the associated FOR statement.

13. OPERATIONAL CONSIDERATIONS

13.1 System Requirements

CBASIC may be executed on any floppy disk based CP/M operating system having at least 20K bytes of memory. In order to make the best use of the power and flexibility of CBASIC, a dual floppy disk system and at least 32K of memory is recommended. If CBASIC is executed in a system smaller than 20K a CP/M LOAD ERROR may occur.

13.2 CBASIC Compile-Time Toggles

Compiler toggles are a series of switches that can be set when the compiler is executed. The toggles are set by typing a dollar-sign ($) followed by the letter designations of the desired toggles starting one space or more after the program name on the command line. Toggles may only be set for the compiler.

Examples of the use of compiler toggles are:

```
CBASIC INVENTRY $BGF

B:CBASIC A:COMPARE $GEC

CBASIC PAYROLL $B

CBASIC B:VALIDATE $E
```

Toggle B suppresses the listing of the program on the console during compilation. When using a slow CRT or teletype, this toggle will reduce compilation time.

If an error is detected the source line with the error and the error message will be printed even if toggle B is set. Toggle B does not affect listing to the printer (toggle F) or disk file (toggle G).

Initially toggle B is off.

APPENDIX B

Run-Time Errors

NO INTERMEDIATE FILE

A file name was not specified with the CRUN command, or no file of type INT and the specified file name was found on the disk.

IMPROPER INPUT - REENTER

This message occurs when the fields entered from the console do not match the fields specified in the INPUT statement. This can occur when field types do not match or the number of fields entered is different from the number of fields specified. Following this message all fields specified by the input statement must be reentered.

Other errors detected during run time cause a 2 letter code to be printed. If the code is preceded by the word WARNING, execution continues. If the code is preceded by the word ERROR, execution terminates. The possible codes are:

Warning Codes

DZ

A number was divided by zero. The result is set to the largest valid CBASIC number.

FL

A field length greater than 255 bytes was encountered during a READ LINE. The first 255 characters of the record are retained; the other characters are ignored.

LN

The argument given in the LOG function was zero or negative. The value of the argument is returned.

NE

A negative number was specified following the raise to a power operator (^). The absolute value is used in the calculation.

13.3 Compiler Output

CBASIC does not require that each statement of a program be assigned a line number. The only statements that must be given a line number are those that have control passed to them by the GOTO, GOSUB, ON or IF statements. During compilation CBASIC assigns a unique number to each source statement regardless of the line number which may be used by the programmer. The CBASIC assigned statement number is the one referred to in error messages (If toggle E is specified) and when using the TRACE option. The statement number takes one of three forms:

```
n:   or   n*   or   n=
```

where n is the number assigned. In most cases the colon (:) will follow the number. The equal sign (=) is printed when the statement has been read in from a disk file with an %include directive. The asterisk (*) is used when the statement contains a user assigned line number that is not referenced anywhere in the program. For example:

```
1:      print "start"
2:      name$="FRED"
3* 10   gosub 40            rem print name
4:      stop
5:
6:%include printrtn         rem rtn to print
7= 40   rem-----rtn to print-----------
8=      print name$
9=      return
```

In the example, statement 3 has an asterisk because the '10' is not referenced at any place in the program. This can be useful in debugging or understanding large programs written in other dialects of BASIC. When all unreferenced line numbers are removed, it is easier to see the logic of the program.

13.4 TRACE

CRUN <filename> [TRACE [<number> [,<number>]]]

The TRACE option is used for run-time debugging. It will print on the console the statement number of each statement as it is executed. The statement number printed is the number assigned to each statement by the compiler. Consider the following program:

LW

A line width less than 1 or greater than 133 was specified in an LPRINTER WIDTH statement.

ME

An error occurred while creating or extending a file because the disk directory was full.

MP

The third parameter in a MATCH function was zero or negative.

NF

The file number specified was less than 1 or greater than 20.

NM

There was insufficient memory to load the program.

NN

An attempt was made to print a number with a PRINT USING statement but there was not a numeric data field in the USING string.

NS

An attempt was made to print a string with a PRINT USING statement but there was not a string field in the USING string.

OD

A READ statement was executed with no DATA statement, or all data statements having already been read.

OE

An attempt was made to OPEN a file that didn't exist and for which no IF END statement had been previously executed.

OI

The expression specified in an ON...GOSUB or an ON...GOTO statement evaluated to a number less that 1 or greater than the number of line numbers contained in the statement.

OM

The program ran out of memory during execution.

QE

An attempt was made to PRINT to a file a string containing a quotation mark.

RE

An attempt was made to read past the end of a record in a fixed file.

APPENDIX A

Compiler Errors

NO SOURCE FILE: <filename>.BAS

The compiler could not locate a source file used in either a CBASIC command or an INCLUDE directive.

PROGRAM CONTAINS n UNMATCHED FOR STATEMENT(S)

There are n FOR statements for which a NEXT could not be found.

PROGRAM CONTAINS n UNMATCHED WHILE STATEMENT(S)

There are n WHILE statements for which a WEND could not be found.

WARNING INVALID CHARACTER IGNORED

The previous line contained an invalid ASCII character. The character is ignored by the compiler. A question mark is printed in its place.

Other errors detected during compilation cause a 2 letter error code to be printed, and a ^ (up-arrow or circumflex) under the position of the error in the line in the listing. The possible error codes are:

CE

The intermediate (INT) file could not be closed.

DE

A disk error occurred while trying to read the BAS file.

DF

There was no space on the disk or the disk directory was full. The intermediate file was not created.

DL

The same line number was used on two different lines. Other compiler errors may cause a DL error message to be printed even if duplicate line numbers do not exist.

DP

A variable in a DIM statement was previously defined.

APPENDIX C

RESERVED WORDS

ABS	EQ	LEN	POS	STOP
AND	EXP	LET	PRINT	STR$
AS	FEND	LINE	RANDOMIZE	SUB
ASC	FILE	LOG	READ	TAB
ATN	FOR	LPRINTER	RECL	TAN
CALL	FRE	LT	REM	THEN
CHR$	GE	MATCH	REMARK	TO
CLOSE	GO	MID$	RENAME	USING
CONSOLE	GOSUB	NE	RESTORE	VAL
COS	GOTO	NEXT	RETURN	WEND
CREATE	GT	NOT	RIGHT$	WHILE
DATA	IF	ON	RND	WIDTH
DEF	INP	OPEN	SGN	XOR
DELETE	INPUT	OR	SIN	
DIM	INT	OUT	SIZE	
ELSE	LE	PEEK	SQR	
END	LEFT$	POKE	STEP	

NU

A NEXT statement occurs without an associated FOR statement.

OO

More than 25 ON statements were used in the program.

SE

The source line contained a syntax error.

SN

A subscripted variable contains an incorrect number of subscripts.

SO

The expression is too complex and should be simplified and placed on more than one line.

TO

The program is too large for the system. The program must be simplified or the system size increased.

UL

A line number that does not exist has been referenced.

US

A string has been terminated by a carriage return rather than by quotes.

VO

Variable names are too long for one statement. This should not occur. If it does please send a copy of this statement to Software Systems.

WE

The expression immediately following a WHILE statement is not numeric.

WU

A WEND statement occurs without an associated WHILE statement.

This program demonstrates a method by which the MATCH function can be used to verify input data.

```
CBASIC COMPILER VER 1.00
   1:       true=-1
   2:       print "Enter your address, please."
   3:       print
   4:       print "Use as many lines as you like."
   5:       print "When you are done, type an extra carriage return."
   6:       in.addr$=" "
   7:       gosub 100
   8:       print using "your address is: &";in.addr$
   9:       stop
  10:
  11:100    rem-----enter addr-----------------
  12:       while true
  13:              addr$="not return"
  14:              while addr$<>""
  15:                     input "";line addr$
  16:                     in.addr$=in.addr$+" "+addr$+" "
  17:              wend
  18:              if match(" ##### ",in.addr$,1) then \
  19:                     return
  20:              print "I believe you forgot to give me your zip ";
  21:              print "code.  Please enter it now."
  22:       wend
  23: END
NO ERRORS DETECTED
```

OF

A calculation produced a number too large. The result is set to the largest valid CBASIC number.

SQ

A negative number was specified in the SQR function. The absolute value is used.

Error Codes

AC

The string used as the argument in an ASC function evaluated to a null string.

CE

An error occurred upon closing a file.

CU

A close statement specified a file number that was not active.

DF

An OPEN or CREATE was specified with a file number that was already active.

DU

A DELETE statement specified a file number that was not active.

DW

An error occurred while writing to a file. This occurs when either the directory or the disk is full.

EF

A read past the end of file occurred on a file for which no IF END statement had been executed.

ER

An attempt was made to write a record of length greater than the maximum record size specified in the associated OPEN, CREATE or FILE statement.

FR

An attempt was made to rename a file to an existing file name.

FU

An attempt was made to read or write to a file that was not active.

IR

A record number less than one was specified.

```
 48: 100            rem sort
 49:        flag=true
 50:        while flag=true
 51:               flag=false
 52:               x=1
 53:               while x<no.recs
 54:                      if element$(x)>element$(x+1) then \
 55:                             temp$=element$(x) :\
 56:                             element$(x)=element$(x+1) :\
 57:                             element$(x+1)=temp$ :\
 58:                             flag=true
 59:                      x=x+1
 60:               wend
 61:        wend
 62:        return
 63:
 64: 150    rem print the sorted output
 65:        if disk.wanted then \
 66:               gosub 160 :\   rem print to disk
 67:               return
 68:        if printer.wanted then \
 69:               lprinter
 70:        x=1
 71:        while x<no.recs
 72:               print using edit$;element$(x)
 73:               x=x+1
 74:        wend
 75:        if printer.wanted then \
 76:               console
 77:        return
 78:
 79: 160    rem print to disk
 80:        x=1
 81:        while x<no.recs
 82:               print using edit$;#2;element$(x)
 83:               x=x+1
 84:        wend
 85:        return
 86:
 87: 200    rem input file not there
 88:        print "the input file is not there"
 89:        print "program stopped"
 90:        stop
 91: END
NO ERRORS DETECTED
```

RG

A RETURN occurred for which there was no GOSUB.

RU

A random read or print was attempted to other than a fixed file.

SB

An array subscript was used which exceeded the boundaries for which the array was defined.

SL

A concatenation operation resulted in a string of more than 255 bytes.

SS

The second parameter of a MID$ function was zero or negative.

TF

An attempt was made to have more than 20 active files simultaneously.

TL

A TAB statement contained a parameter less than 1 or greater than the current line width.

UN

A PRINT USING statement was executed with a null edit string.

WR

An attempt was made to write to a file after it had been read, but before it had been read to the end of the file.

The following example of the include is a routine that requests the month from the console and verifies it. It assumes the following files are on disk drive A:

```
     350.BAS         inputs the month number
     360.BAS         checks the month number
     370.BAS         check for numeric data
```

```
 1:        rem-----------------------------------
 2:        rem    mainline code to request and verify month
 3:        rem-----------------------------------
 4:
 5:        FALSE=0
 6:        TRUE=-1
 7:
 8:        gosub 350             rem enter and verify month
 9:        print using "the month number is ##";month
10:        stop
11:
12: %include 350          rem enter and verify month
13= 350    rem-----get the month number---------
14=        valid=FALSE
15=        while not valid
16=               input "enter number of month ";line check$
17=               gosub 360             rem check month number
18=        wend
19=        return
20=
21= %include 360          rem check month number
22= 360    rem-----check month number-------
23=        gosub 370             rem is it all numeric?
24=        valid=TRUE
25=        if not numeric then valid=FALSE
26=        month=val(check$)
27=        if month<1 or month>12 then valid=FALSE
28=        return
29=
30= %include 370          rem is it all numeric?
31= 370    rem-----is it all numeric?--------
32=        numeric=FALSE
33=        while len(check$)>0
34=               for x=1 to len(check$)
35=                      if mid$(check$,x,1)<"1" or \
36=                         mid$(check$,x,1)>"9" then \
37=                             return
38=               next x
39=               numeric=TRUE
40=               return
41=        wend
42=        return
43: END
NO ERRORS DETECTED
```

APPENDIX D

SAMPLE PROGRAMS

This program allows an editing string to be specified for a print using statement. Sample data may be repeatedly input and the edited data is printed.

```
CBASIC COMPILER VER 1.00
   1: true=-1
   2: while true
   3:         input "do you want alpha or numeric (A/N)?";line ans$
   4:         if ans$="stop" or ans$="STOP" then stop
   5:         gosub 5     rem enter edit$
   6: wend
   7:
   8: 5       rem enter edit$
   9:         while true
  10:                input "enter edit$";line edit$
  11:                if edit$="stop" or edit$="STOP" then return
  12:                if ans$="a" or ans$="A" then \
  13:                       gosub 20 \      rem enter alpha
  14:                else \
  15:                       gosub 10        rem enter numbers
  16:         wend
  17:
  18: 10     rem enter numbers
  19:         while true
  20:                input "enter number";line num$
  21:                if num$="stop" or num$="STOP" then return
  22:                print using edit$;val(num$)
  23:         wend
  24:
  25: 20     rem enter alpha
  26:         while true
  27:                input "enter alpha";line alpha$
  28:                if alpha$="stop" or alpha$="STOP" then return
  29:                print using edit$;alpha$
  30:         wend
  31: END
NO ERRORS DETECTED
```

Software Systems invites your comments regarding the CBASIC language and the contents of this manual. Your views, favorable or unfavorable, aid us in improving our products. Remove this page from the manual and use the backside for your comments. Fold it along the dotted lines, stamp, seal and mail it to Software Systems. We appreciate your consideration. All comments and suggestions become the property of Software Systems.

fold

stamp

To:

SOFTWARE SYSTEMS
Post Office Box 1705
Vallejo, California
94590

fold

From:

This program does a bubble-sort on data read from a disk file. Output can be to another file or to the console or a printer.

```
CBASIC COMPILER VER 1.00
   1: true=-1
   2: input "what file to sort";name$
   3: input "what is record length";length
   4: max.recs=50
   5: dim element$(max.recs)
   6: gosub 10               rem ask where to put output file
   7: if end #1 then 200      rem error
   8: open name$ recl length as 1
   9: if end #1 then 50.1
  10: edit$="&"
  11: print "reading"
  12: gosub 50               rem read in file
  13: print "sorting"
  14: gosub 100              rem sort word array
  15: print "writing"
  16: gosub 150              rem write sorted output
  17: print
  18: print using "Number of records sorted was ###";no.recs
  19: print
  20: stop
  21:
  22: 10     rem ask where to put sorted file
  23:        ans$="XXX"
  24:        while ans$<>"P" and ans$<>"p" and ans$<>"c" and ans$<>"C" \
  25:               and ans$<>"D" and ans$<>"d"
  26:               input "printer, console or disk (p,c,d)?";line ans$
  27:        wend
  28:        if ans$="D" or ans$="d" then \
  29:               disk.wanted=true :\
  30:               input "what is name of output file?";out.name$ :\
  31:               create out.name$ recl length as 2 :\
  32:               return
  33:        if ans$="P" or ans$="p" then \
  34:               printer.wanted=true :\
  35:               return
  36:        return
  37:
  38: 50     rem read in the file
  39:        no.recs=0
  40:        while true
  41:               read #1;in$
  42:               no.recs=no.recs+1
  43:               element$(no.recs)=in$
  44:        wend
  45: 50.1   rem here at eof
  46:        return
  47:
```

This program demonstrates a binary search routine.

```
CBASIC COMPILER VER 1.00
   1: true=-1
   2: array.size=100
   3: dim array(array.size)
   4: gosub 110      rem init array
   5: while true
   6:         arg=0
   7:         while arg<1 or arg>array.size
   8:                input "enter argument ";arg
   9:         wend
  10:         gosub 100
  11:         if found=0 then \
  12:                print "not there" :\
  13:         else \
  14:                print array(found)
  15: wend
  16:
  17: 110    rem init array
  18:         for x=1 to array.size
  19:                array(x)=x
  20:         next x
  21:         return
  22:
  23: %include inclbin       rem binary search rtn
  24= 100    rem binary search
  25=         lower=0
  26=         upper=array.size+1
  27=         while true
  28=                if upper-lower=1 then \
  29=                       found=0 :\
  30=                       return
  31=                found=int(((upper-lower)/2)+.5)+lower
  32=                if array(found)=arg then \
  33=                       return
  34=                if arg>array(found) then \
  35=                       lower=found :\
  36=                else \
  37=                       upper=found
  38=         wend
  39=         return
  40: END
NO ERRORS DETECTED
```

The mainline procedure has one include statement for paragraph 350. Procedure 350 in turn has an include for procedure 360; 360 then includes 370. Each of these routines with the exception of the mainline, are modular, use a well-defined method of communication, and may be called by other portions of code. 'Canned' procedures can result in saving coding and debugging time.

OM Current program exceeded available memory. Close unneeded opened files, nulify unused strings, and read data from a disk file.

QE A PRINT string contained a quotation mark and could not be written to the specified file.

RB Attempted random access to a file activated with BUFF where more than one buffer was specified.

RE Attempted read past the end of a record in a fixed file.

RG A RETURN was issued for which there was no associated GOSUB statement.

RU A random read or print was attempted to a file that was not fixed.

SB An ARRAY subscript exceeded the defined boundries.

SL A string longer than 255 bytes resulted from a concatenation operation.

SO The file specified in SAVEMEM was not on the indicated disk.

SS The second parameter in the MID$ function, or the last parameter in the LEFT$, or RIGHT$, was negative or zero.

TL The TAB statement parameter was less than 1 or greater than the current line width.

UN A PRINT USING statement contained a null edit string, or an escape character (\) was the last in an edit string.

WR An attempt was made to write to a file after it had been read, but before it had been read to the end of the file.

CS The CHAINed program reserved a different amount of memory with a SAVEMEM statement than the calling program.

CU An inactive file number was specified in the CLOSE statement.

DF An already active file number was specified in an OPEN or CREATE statement.

DU An inactive file number was specified by a DELETE statement.

DW Indicates a write to a file for which no IF END statement has been executed; may occur if the director of disk is full.

EF Indicates a read past an end of file for which no IF END statement has been executed.

ER A write to a record whose length exceeds the maximum record length specified by an OPEN, CREATE, or FILE statement was attempted.

FR The renamed filename already exists.

FU A read or write operation was attempted to an inactive file.

IF An invalid file name was specified.

IR A record number of zero was specified.

IV Execution of and INT file created by a version 1 compiler was attempted. Recompile using version 2 compiler.

IX A FEND statement was encountered before execution of a RETURN statement.

ME A full directory resulted in an error while creating or extending a file.

MP A third MATCH function parameter was zero or negative.

NF A file number less than 1 or greater than 20 was specified, or a file statement was executed when 20 files were already active.

NM Not enough memory was available to load the program.

NN A PRINT USING statement could not print a number because no numeric data field could be found in the USING string.

OD A READ statement was executed with no corresponding data.

OE Invalid execution of an OPEN statement for a nonexistent file when no prior IF END statement has been executed.

OI An ON GO SUB expression, or an ON GOTO statement evaluated to a number less than 1, or greater than the number of line numbers in the statement.

IMPROPER INPUT-REENTER The fields entered at the keyboard do not match those specified in the INPUT statement.

WARNING CODES

Two letter codes preceded by the word "WARNING", indicate errors that do not prevent execution of the program, but should be attended to. These codes are:

DZ A number divided by zero resulted in the largest CBASIC number.

FL A field length greater than 255 bytes was encountered during a READ LINE; the remainder is ignored.

LN A LOG function argument was zero or negative; the value of the argument is returned.

NE A negative number before the raise to a power operator (^) was encountered, resulting in the absolute value of the parameter being calculated.

OF A real variable calculation produced an overflow. The result is set to the largest valid CBASIC real number. Overflow is not detected with integer arithmetic.

SQ A negative number was specified in the SQR function. The absolute value is used.

RUN-TIME ERROR CODES

The following two letter codes are preceded by the word "ERROR" and cause execution to terminate:

AC An ASC function string argument evaluated to a null string.

BN The BUFF value in either the OPEN or CREATE statement is less than 1 or greater than 52.

CC The CHAINed program code area is greater than the calling program's code area. Use %CHAIN for adjustment.

CD The CHAINed program data area is greater than the calling program's data area. Use %CHAIN for adjustment.

CE The file being closed could not be found in the directory.

CF The CHAINed program constant area is greater than the calling program's constant area. Use %CHAIN for adjustment.

CP The CHAINed program variable storage area is greater than the calling program's variable storage area. Use %CHAIN.

ND A DEF statement could not be found for a corresponding FEND statement.

NI A NEXT variable reference did not match that referenced by the associated FOR statement.

NU A NEXT statement occurred without an associated FOR statement.

OF An illegal branch from within a line function was attempted.

OO The ON statement limit of 40 was exceeded.

PM A DEF statement was encountered within a multiple line function. Functions cannot be nested.

SE A syntax error occurred in the source line, usually as the result of an improperly formed statement or misspelled keyword.

SF A numeric instead of a string expression was used in a SAVEMEM statement. Check for quotes around string constants.

SN An incorrect number of subscripts were found in a subscripted variable, or a DIM variable was previously used with a different number of dimensions.

SO A too complex statement should be simplified in order to be compiled.

TO Indicates a symbol table overflow; meaning the program is too large for the current OSBORNE 1 memory configuration.

UL Reference to a non-existent line number has been made.

US A string was terminated with a carriage return, rather than quotes.

VO Variable names are too long for one statement.

WE The expression following the WHILE statement is not numeric.

WN The nesting level of WHILE statements (12) has been exceeded.

WU A WEND without an associated WHILE statement was encountered.

RUNTIME ERRORS

The following run-time error messages are displayed below the most recent screen line, to indicate conditions which usually terminate program execution.

NO INTERMEDIATE FILE A filename of type .INT could not be located on the specified drive.

OM Current program exceeded available memory. Close unneeded opened files, nulify unused strings, and read data from a disk file.

QE A PRINT string contained a quotation mark and could not be written to the specified file.

RB Attempted random access to a file activated with BUFF where more than one buffer was specified.

RE Attempted read past the end of a record in a fixed file.

RG A RETURN was issued for which there was no associated GOSUB statement.

RU A random read or print was attempted to a file that was not fixed.

SB An ARRAY subscript exceeded the defined boundries.

SL A string longer than 255 bytes resulted from a concatenation operation.

SO The file specified in SAVEMEM was not on the indicated disk.

SS The second parameter in the MID$ function, or the last parameter in the LEFT$, or RIGHT$, was negative or zero.

TL The TAB statement parameter was less than 1 or greater than the current line width.

UN A PRINT USING statement contained a null edit string, or an escape character (\) was the last in an edit string.

WR An attempt was made to write to a file after it had been read, but before it had been read to the end of the file.

CV A subscripted variable in a COMMON statement was not properly defined.

DL Duplication of the same line number, an undefined function, or a DIM statement which does not precede all referenced arrays, was detected.

DP A DIM variable was previously defined in another DIM statement or was used as a simple variable.

FA A function name not used in the function was encountered to the left of the equal sign in an assignment statement.

FD A function name is the same in two DEF statements.

FE An incorrect mixed mode expression exists in a FOR statement, usually the expression following TO is involved.

FI The FOR loop index is not an unsubscripted numeric variable expression.

FN An incorrect number of parameters are used in the function reference.

FP The function reference parameter type does not match that in the DEF statement.

FU An undefined function has been referenced.

IE The IF statement expression erroneously evaluates to type string.

IF The FILE statement variable is type numeric instead of type string.

IP An input prompt string was not enclosed in quotes.

IS A subscripted variable was not dimensioned before it was referenced.

IT Indicates that an invalid compiler directive was issued.

IU A DEF statement defined array was not subscripted.

MC A variable was defined more than once in a COMMON statement.

MF The expression evaluates to type string instead of type numeric.

MM An invalid mixed mode was encountered, usually caused by a mixture of string and numeric types in an expression.

MS A numeric instead of a string expression was used.

CS The CHAINed program reserved a different amount of memory with a SAVEMEM statement than the calling program.

CU An inactive file number was specified in the CLOSE statement.

DF An already active file number was specified in an OPEN or CREATE statement.

DU An inactive file number was specified by a DELETE statement.

DW Indicates a write to a file for which no IF END statement has been executed; may occur if the director of disk is full.

EF Indicates a read past an end of file for which no IF END statement has been executed.

ER A write to a record whose length exceeds the maximum record length specified by an OPEN, CREATE, or FILE statement was attempted.

FR The renamed filename already exists.

FU A read or write operation was attempted to an inactive file.

IF An invalid file name was specified.

IR A record number of zero was specified.

IV Execution of and INT file created by a version 1 compiler was attempted. Recompile using version 2 compiler.

IX A FEND statement was encountered before execution of a RETURN statement.

ME A full directory resulted in an error while creating or extending a file.

MP A third MATCH function parameter was zero or negative.

NF A file number less than 1 or greater than 20 was specified, or a file statement was executed when 20 files were already active.

NM Not enough memory was available to load the program.

NN A PRINT USING statement could not print a number because no numeric data field could be found in the USING string.

OD A READ statement was executed with no corresponding data.

OE Invalid execution of an OPEN statement for a nonexistent file when no prior IF END statement has been executed.

OI An ON GO SUB expression, or an ON GOTO statement evaluated to a number less than 1, or greater than the number of line numbers in the statement.

REV: 2/28/82

CBASIC ERROR MESSAGES

CBASIC COMPILER ERRORS:

The following compiler error messages appear during compilation of a source file:

NO SOURCE FILE: <filename>.BAS- The source file could not be found on the indicated drive.

OUT OF DISK SPACE Insufficient disk space was encountered by the compiler while writing the .INT or .LST file.

OUT OF DIRECTORY SPACE The compiler ran out of directory entries while attempting to create or extend an .INT or .LST file.

BDOS ERROR ON (A,B) CP/M error message indicates that an error occurred while reading from, or writing to, a disk file.

PROGRAM CONTAINS n UNMATCHES FOR STATEMENT(S) n FOR statements have no associated NEXT statements,

PROGRAM CONTAINS n UNMATCHES WHILE STATEMENT(S) n WHILE statements have no associated WEND statements.

PROGRAM CONTAINS 1 UNMATCHED DEF STATEMENT A multiple line function was not terminated with a FEND statement, possibly causing further errors.

WARNING INVALID CHARACTER IGNORED An invalid character was detected in the last line, which was replaced by a question mark and ignored.

INCLUDE NESTING TOO DEEP NEAR LINE n An INCLUDE statement exceeded the maximum nesting level near line n.

COMPILER ERROR CODES

The following two letter error codes display with the line number and position of the error:

BF Invalid branch into a multiple line from outside of the function.

BN Invalid numeric constant was encountered.

CI Improper filename used in an %INCLUDE directive.

CS The COMMON statement was not the first program statement preceded only by a directive, remark, or blank line.

IMPROPER INPUT-REENTER The fields entered at the keyboard do not match those specified in the INPUT statement.

WARNING CODES

Two letter codes preceded by the word "WARNING", indicate errors that do not prevent execution of the program, but should be attended to. These codes are:

DZ A number divided by zero resulted in the largest CBASIC number.

FL A field length greater than 255 bytes was encountered during a READ LINE; the remainder is ignored.

LN A LOG function argument was zero or negative; the value of the argument is returned.

NE A negative number before the raise to a power operator (^) was encountered, resulting in the absolute value of the parameter being calculated.

OF A real variable calculation produced an overflow. The result is set to the largest valid CBASIC real number. Overflow is not detected with integer arithmetic.

SQ A negative number was specified in the SQR function. The absolute value is used.

RUN-TIME ERROR CODES

The following two letter codes are preceded by the word "ERROR" and cause execution to terminate:

AC An ASC function string argument evaluated to a null string.

BN The BUFF value in either the OPEN or CREATE statement is less than 1 or greater than 52.

CC The CHAINed program code area is greater than the calling program's code area. Use %CHAIN for adjustment.

CD The CHAINed program data area is greater than the calling program's data area. Use %CHAIN for adjustment.

CE The file being closed could not be found in the directory.

CF The CHAINed program constant area is greater than the calling program's constant area. Use %CHAIN for adjustment.

CP The CHAINed program variable storage area is greater than the calling program's variable storage area. Use %CHAIN.

SAMPLE PROGRAMS:

```
REM TRANSFERS CONTROL TO A PROGRAM OF THE USER'S
REM CHOICE DIRECT COMPILER TO RESERVE EXTRA SPACE
REM FOR PROGRAM'S CONSTANT, CODE, DATA, AND VARIABLE
REM AREAS (32, 1000, 32, AND 32 BYTES RESPECTIVELY)
REM (TO PREVENT OVERWRITING BY THE CHAINED PROGRAM)

% CHAIN 32, 1000, 32, 32

 INPUT "WOULD YOU LIKE TO RUN A PROGRAM (Y/N)?";RUN$

   IF RUN$ = "Y" THEN \
    INPUT "WHICH ONE?"; PROG.NAME$:\
CHAIN PROG.NAME$

  STOP


REM PROGRAM CHECKCHAIN -- WHEN MAIN PROGRAM
REM CHAINS HERE A MESSAGE IS PRINTED TO SHOW THE
REM CHAIN WAS SUCCESSFUL AND THEN CONTROL IS
REM TRANSFERRED BACK TO THE MAIN PROGRAM

 PRINT "YOU HAVE SUCCESSFULLY CHAINED TO PROGRAM CHECKCHAIN"
 CHAIN "MAIN"
 STOP
```

CLOSE- Close specified files

OVERVIEW:

The CLOSE statement closes open files.

FORMAT: **CLOSE integer expression {,integer expression}**

Each integer expression denotes an open file that is to be closed. Reference to a file that has not been opened will cause an error. When a file is closed, the file number is released and the associated buffer space is returned to the system.

NOTE: CLOSE will terminate any IF END statement that references the file being closed.

EXAMPLES:

```
CLOSE 1
CLOSE input.file.id%, temp.file.l%
```

SAMPLE PROGRAM

```
REM CREATE TWO NEW FILES, WRITE DATA TO THEM,
REM AND CLOSE THE FILES
```

REV: 2/28/82

ND A DEF statement could not be found for a corresponding FEND statement.

NI A NEXT variable reference did not match that referenced by the associated FOR statement.

NU A NEXT statement occurred without an associated FOR statement.

OF An illegal branch from within a line function was attempted.

OO The ON statement limit of 40 was exceeded.

PM A DEF statement was encountered within a multiple line function. Functions cannot be nested.

SE A syntax error occurred in the source line, usually as the result of an improperly formed statement or misspelled keyword.

SF A numeric instead of a string expression was used in a SAVEMEM statement. Check for quotes around string constants.

SN An incorrect number of subscripts were found in a subscripted variable, or a DIM variable was previously used with a different number of dimensions.

SO A too complex statement should be simplified in order to be compiled.

TO Indicates a symbol table overflow; meaning the program is too large for the current OSBORNE 1 memory configuration.

UL Reference to a non-existent line number has been made.

US A string was terminated with a carriage return, rather than quotes.

VO Variable names are too long for one statement.

WE The expression following the WHILE statement is not numeric.

WN The nesting level of WHILE statements (12) has been exceeded.

WU A WEND without an associated WHILE statement was encountered.

RUNTIME ERRORS

The following run-time error messages are displayed below the most recent screen line, to indicate conditions which usually terminate program execution.

NO INTERMEDIATE FILE A filename of type .INT could not be located on the specified drive.