CBASIC 2

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

Version 2
November 1981

Digital Research, Inc.
P. 0. Box 579
Pacific Grove, CA 93950

408-649-3896

DISCLAIMER

Digital Research 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, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereo without obligation of Digital Research to notity any person of such revision or changes.

TRADEMARKS

CBASIC and MP/M are trademarks of Digital Research
CP/M is a registered trademark of Digital Research

1. CBASIC

1.1 INTRODUCTION

This manual describes version two of CBASIC, a comprehensive, commercially oriented compiler/interpreter designed for use with the CP/M (tm) and MP/M-80 operating systems. CP/M and MP/M-80 are trademarks of Digital Research. CP/M is available on a multitude of 8080, 8085, and Z80 microcomputer systems.

In this manual, unless it is stated otherwise, CP/M will be used to indicate version I or 2 of CP/M or MP/M. There are many derivations of CP/M. CBASIC should also operate with these systems. CPU will refer to the microprocessor chip installed in the system.

CBASIC has a variety of extended features including the IF . . . THEN . . . ELSE and WHILE constructs and access to disk files. CBASIC also allows the use of 31 character variable names, and the free use of comments, spaces, and tabs. These aid in creating programs that are self-documenting and maintainable.

Version two of CBASIC adds integer variables, multiple line functions, chaining with common variables, and additional predefined functions as well as other improvements. A cross reference lister is also provided.

The CBASIC system consists of three programs. The first program, the compiler, converts the user's source language program into a series of coded operations that are placed on an intermediate disk file. The second program, the runtime monitor, directly executes the operations included in the intermediate file. The final program, XREF.COM, will produce a cross reference listing of all variables used in a CBASIC source program.

To use CBASIC a microcomputer system using the CP/M operating system must be available. This manual assumes a working knowledge of the following CP/M documentation:

(a) AN INTRODUCTION TO CP/M FEATURES AND FACILITIES

(b) ED: A CONTEXT EDITOR FOR THE CP/M DISK SYSTEM

(c) CP/M INTERFACE GUIDE

These manuals are available from Digital Research, PO Box 579, Pacific Grove, California.

A newcomer to the field of computers would do well to read an introductory text on the Basic Language.

The reference section, chapters 2 through 10, describes the facilities of the language. Chapter 11 expands on the use of files. Chapters 12 and 13 describe operation of the compiler. Three appendices follow which list compiler and runtime error messages and list key words.

1.2 For CBASIC I Programmers

Programmers familiar with version 1 of CBASIC should review thi manual paying particular attention to the use of integer variables. Chapter 3 provides details on using integers in expressions. The sections concerning new statements and functions should be read in detail. Chapters 12 and 13 also contain much new information.

A program that compiled and executed with version I should operate properly with version 2. However, an INT file created by the version I compiler will not execute with the version 2 runtime monitor. The source program must be recompiled.

If statements appear not to operate properly, Digital Research Inc. would appreciate a note which includes the statement or statements which are causing the problem along with a description of the problem

1.3 Program Identification Numbers

All Digital Research language programs sign-on with the program name followed by an identification number. These numbers are in the following form:

V.RC

"V" is the version number. This manual describes version 2 programs. The "R" is the release number of the program. As errors are corrected in a particular version, new releases are made available. The "C" is the configuration. A zero means that the program is configured to operate with standard CP/M version 2 and MP/M. Other configurations may be made available in the future.

2. General Informatio

2.1 STATEMENTS

A program consists of zero or more properly formed CBASIC statements contained in a diskette file. CBASIC source statements are also called source code or source statements. An END statement, if present, terminates the program, and any statements following the END statement are ignored. An end of-file on the source file also terminates the program. In this case the END statement is supplied by CBASIC.

In this manual the term line, in the context of a line of source code, means a string of characters terminated with a carriage return and line feed. A statement may span more than one line or multiple statements may appear on the same line.

The entire ASCII character set is accepted, but most statements may be written using the common 64 character subset. Lower case letters are converted by the compiler to upper case except when they appear in strings or remarks. A compiler toggle, described in Chapter 13, will inhibit al conversion to upper case.

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. (To produce the \ character on Franklin ACE computers, press CTRL / or CTRL SHIFT / simultaneously). The statement can then be continued on the next line. CBASIC keywords, variable names and string constants may not be broken in the middle and continued on the next line. A continuation character may not be used in a Data Statement since it is treated as a character within a string constant. Likewise backslash

characters within string constants enclosed in quotation marks (see section 3. 1) are not treated as continuation characters.

(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; an IF statement must be the first statement on a line. See the REM statement (section 2.4) for an exception to this rule.

Spaces may precede statements; any number of spaces may appear wherever one space is permitted. Extra spaces, such as for indenting statements to enhance readability, do not increase the size of the intermediate file created by the compiler.

2.2 NOTATION

All of the CBASIC statements are described in this manual. Each description includes 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 PRINT are keywords in CBASIC. Appendix C contains a list of all keywords used by CBASIC.

Angle Brackets < >

Angle brackets enclose 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 STATEMENT NUMBERS

Statement numbers are optional. They are ignored except when they appear in a GOTO, GOSUB, ON, or IF statement. In these cases, the statement number must appear as the label of one and only one statement in the program Statement numbers do not have to be in sequential order. For example

```
40 INPUT ITEM
PRINT ITEM
30 GOTO 40
```

In this program the line number 30 is not required; it is ignored during compilation. However, the 40 appears in a GOTO statement and thus must be used as a statement number once and only once in the program. Statement numbers may contain any number of digits but only the first 31 are considered significant by the compiler.

An additional feature of CBASIC statement numbering is that any valid number may be used as a statement number. This allows the use of non-integer statement numbers. It is possible to write an entire program or subprogram with statement numbers that are all decimal fractions and range between two consecutive integers.

Statement numbers can even be in exponential (E) format. This is a convenient feature when writing procedures that will be included in other programs because it helps to insure that statement numbers will be unique.

The following are examples of valid CBASIC statement numbers:

```
1
0
100
100.0
100.213
100E21
```

The statement numbers 100 and 100.0 are treated as different statement numbers by the compiler. In other words it is the string of characters which determines the statement number and not the numeric value.

2.4 REM STATEMENT

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

[<stmt number>] REMARK [<string terminated with CR>]

A REM statement is ignored by the compiler, and compilation continues with the statement following the next carriage return. A continuation characte causes the next line to be part of the remark. 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 statement number of a remark may be used in a GOTO, GOSUB, IF, or ON statement.

Examples of REM statements follow:

REM THIS IS A REMARK. This is also a remark tax = 0. 15 * income rem lowest tax rate

REM this section contains the tax tables for California

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 optional between the two statements. In all other cases involving multiple statements on the same line, the colon must separate the statements. In addition, if the REM statement is used on the same line with other statements, it must be the last statement on the line.

2.5 EXECUTING 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 file name. Finally the intermediate (INT) file created by the compiler is executed by invoking the runtime program, again using the source program name as a file name.

The source program will normally be created using a text editor. The source program must have a file type of BAS. Each line of a source program is terminated by a carriage return and line feed. The line may be any length, however, the compiler will only print the first 132 characters of each line.

When typing source programs, identifiers (variable names, reserved words, and user-defined function names) may not be abbreviated and must be separated by a character other than a number or letter. In general, spaces wil be used to delimit identifiers. All letters in identifiers are converted to uppercase unless the conversion is inhibited by compiler toggle D (see Chapter 13).

CBASIC differs from many other basics in its requirement that keywords and identifiers may not be run together. For instance:

```
READA
```

is not accepted by the CBASIC compiler. The statement must be written

```
READ A
```

FORI=JTOIO is a valid CBASIC statement, but it assigns the variable JTO10 to the variable FORI.

The CBASIC compiler is invoked as follows:

```
CBAS2 <filename> [<disk ref>] [$<toggle> {<toggle>}]
```

where filename is the name of the source file. A file type of HAB Is assumed by the compiler. Compiler toggles, preceded by a dollar sign, may follow the file name. They are discussed in Chapter 13.

The compiler produces an intermediate file in the CBASIC machine language. The intermediate file uses the same name as the source program but of type INT. The INT file is normally placed on the same disk as the source file. The disk reference is used to specify the drive on which the programmer desires to have the INT file placed. The disk reference is optional; if present it is of the form A:, B:, etc.

The following command will compile the program INVENTORY.BAS taking the source from the currently selected drive, and place the INT file on drive B:

CBAS2 INVENTORY B

If a listing is selected (section 13.2), the name of the program as it appears following CBAS2 and any other characters up to the dollar sign or end of the command will appear in the heading of each page of the listing. For instance:

CBAS2 COST ON 7 NOVEMBER 1990 $EBF will result in the following heading

CBASIC COMPILATION OF COST ON 7 NOVEMBER 1980

The source program is normally listed on the console device. Any error messages will be listed after the statement in which the error was detected (see section 13.3). If errors are detected during compilation, the source file must be corrected using the text editor. The compiler error messages are listed in appendix A. The program is then recompiled. If no errors occur during compilation, the intermediate file may be executed by typing the command:

CRUNC2 <filename> [TRACE [<Inl>[,<ln2>]]] [<cmd>]

The trace option is described in chapter 13. The command field (<cmd>) i used with the COMMAND$ pre-defined function discussed in chapter 7.

If errors are found during execution, the source program must be corrected and then recompiled. Runtime error messages are described in appendix B.

3. Forming Expressions

This chapter discusses the formation of expressions. First the components of expressions, constants and variables, are described. These elements are then combined to form expressions. Expressions are a fundamental building block used in many CBASIC statements.

3.1 STRINGS

A string constant is defined as zero or more valid alphanumeric characters enclosed 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 a single line. A carriage return may not be part of a string. Embedded quotation marks are entered as two adjacent quotes.

The following examples demonstrate valid string constants

- 123 -

-July 4, 1979"

"Enter your name please

- - "Look, look," - said Tom"

In the final example the string is:

"Look, look," said Tom

Internally, strings are stored with the length of the string as the first byte. The characters of the string follow. The length is stored as a binary number from 0 to 255.

3.2 NUMBERS

Two types of numeric quantities are supported by CBASIC, Integer and Real. A real constant may be written in either fixed format or exponentia notation. In both cases it may contain from 1 to 14 digits, a sign, and a decimal point. In exponential notation the exponent is of the form "ssdd", 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 I.0E-64 to 9.9999999999999E62. Although only 14 significant digits are maintained internally by CBASIC, more digits may be included in a real constant. Real constants are rounded to 14 significant digits.

Real numbers are stored in eight bytes of memory. The first byte is the sign and exponent. The exponent is maintained in excess 64 code. The seven remaining bytes contain a normalized mantissa stored as packed decimal digits. The high order four bits of the rightmost byte is the most significant digit of the mantissa.

If a constant does not contain an embedded decimal point, Is not in exponential notation, and ranges from - 32768 to + 32767, the constant is treated as an integer. Integer values are stored as sixteen bit two's complement binary numbers.

Integer constants may also be expressed as hexadecimal and binary constants. If the constant is terminated by the letter H it is hexadecimal. The letter B terminates a binary constant. The first digit of a hexadecimal constant must be numeric. For instance 255 in hexadecimal would be OFFH, not FFH. FFH would be a valid identifier (see section 3.3).

Binary and hexadecimal constants may not contain a decimal point. The value retained is the sixteen least significant bits of the number specified.

In this manual the term real number and floating point number will be used interchangeably. The term numeric will apply to either a real or integer quantity.

Examples of valid numbers are:

1, 1.0, - 99, 123456.789

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

1. 5 E + 3 is equivalent to 1500. 0

1.5E-3 is equivalent to .0015

lab0H, 10 11111 0B OFFFFH

3.3 IDENTIFIERS

An identifier begins with an alphabetic character followed by any number of alphanumeric characters or periods. Identifiers identify or name variables used within a program. Only the first 31 characters are considered unique, however the identifier may be of any length. If the last character in the identifier is a dollar sign, the identifier is of type string. If the identifier ends in a percent sign, it represents an integer. Those identifiers not ending with a dollar sign or percent sign are of type real.

All lower case letters appearing in an identifier are converted to upper case unless compiler toggle D is set (Chapter 13). Using periods in identifiers make programs more readable. For instance BAD.DEBT% is clearer than BADDEBT%.

Using identifiers which are longer than two characters improves program readability without increasing the size of the intermediate file created by the compiler.

Examples of valid identifiers are:

A, B$, cl, c1234%

Payroll.Record, NEW.SUM.AM

INDEX% FLAG.3%, counter%

ANSWER$, file.name$, CUSTOMER. ADDRESS$

3.4 VARIABLES AND SUBSCRIPTE VARIABLES

The general form of a variable is:

<identifier> [(<subscript list>)]

The general form of a subscript list is: <expression> I , <expression>

The expressions in a subscript list must be numeric. Access to array elements is more efficient if integer expressions are used in subscript lists. I the expression is real, the value is rounded to the nearest integer prior to using the value. If an expression in a subscript list is of type string, an error occurs. The subscript list indicates that the variable is a subscripted variable and indicates which element of the array is being referenced.

Each variable has a value associated with it at all times during execution o a program. Initially numbers are zero and strings are null strings. 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 which may be assigned to a string variable is 255 characters.

The identifier used to represent a variable may not begin with FN. Such identifiers are used to specify user defined functions (See chapter 8).

A variable in CBASIC may represent an integer, real number, or a string depending on the type of the identifier.

Examples of variables are:

```
X$
PAYMEN
day.of.deposit%
```

The following examples show subscripted variables:

```
Y$(i%,j%)

COST(3,5)

POS%(XAXIS%,YAXIS%)

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

When subscripts are calculated, a check is made to ensure that the element selected resides in the referenced array. A runtime error occurs if it does not. The runtime check insures that the location calculated is included within the physical storage area of the array. It is not necessarily a valid entry.

Before a subscripted variable may be referenced in a program, it must be dimensioned using the DIM statement. The DIM statement specifies the upper bound of each subscript and allocates storage for the array.
A DIM statement is an executable statement; each execution will allocate a new array. If the array contains numeric data the previous array is deleted prior to allocating space for a new array. If the array is of type string each element must be set to a null string prior to re-executing the DIM statement to regain the maximum amount of storage.

The general form of a DIM statement is:

[<stmt number>] DIM <identifier> (<subscript list>)

[,<identifier> (<subscript list>)]

The dimension statement dynamically allocates space for numeric 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 all elements of string arrays 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.

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), SALES. PERSON%(STAFF. SIZE%)

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

The same identifier may be used as both a variable and as a subscripte variable within the same program.

3.5 EXPRESSIONS

Expressions consist of algebraic combinations of function references, variables, constants, and operators. They evaluate to an integer, real, or string value. Function references are discussed in chapter 8.

The hierarchy of operators is:

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

Arithmetic and relational operations may be performed on either integer or real numbers. If an integer and real number are to be combined using one o these operators, the integer value is first converted to a real number. The operation is then performed on the two real values resulting in a real value. This is referred to as mixed mode arithmetic.

Mixed mode operations take additional time to execute and the compiler generates more code. A mixed mode expression will always evaluate to a rea value.

If real values are used, the power operator calculates the logarithm of the number being raised to the power. Since the logarithm of a negative number is undefined, a warning results when the number to the left of the operator is negative. The absolute value of the negative quantity is used to calculate the result. The exponent may be either positive or negative.

If both values used with the power operator are either integer constants or integer variables, the result is calculated by successive multiplication. This allows a negative integer number to be raised to an integer power. In the case of integers, if the exponent is negative, the result is zero. In all cases, 0 ^ 0 is I and 0 ^ X (when X is not equal to 0) is 0.

If the exponent is an integer but the base is real, the integer is converted to a real value prior to calculating the result. Likewise, if the exponent is real but the base is an integer quantity, the result is calculated using real values.

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 interchangeable with the corresponding algebraic operators (<, <, etc.).

Example of expressions

```
amount * tax
cost + overhead percent
a*b/c(I.2 + xyz)
last.name$ + "  "+ first.name$
index% + I
```

Relational operators result in integer values. A 0 is false and a - 1 is true. Logical operators NOT, AND, OR, and XOR operate on integer values and result in an integer number. If a real value is used with logical operators it is first converted to an integer.

If a numeric quantity is greater than 32,767 or less than - 32,768, it cannot be represented by a 16 bit two's complement binary number. Logical operations on such a number will give unpredictable results.

Results of logical operations

12 AND 3 =0	1100B AND 0101B = 4
NOT -1 =0	NOT 3H = -4
12 OR 3 = 15	OCH OR 5H = 13
12 XOR 3 = 15	12 XOR 5 = 9
12.4 XOR 3.2 = 15	12.4 XOR 3.7 = 8

By using integer expressions for relational tests and logical operations a substantial increase in efficiency results. Programs written in version 1 of CBASIC should be converted to use integer variables where ever possible.

The following point should be understood about numeric constants. If the string of digits contains no decimal point or ends in a decimal point, CBASIC atempts to store it as an integer. If the resulting number is in the range o CBASIC integers, it is treated as an integer. If the constant is then required in an expression as a real number, a conversion to a real number occurs at runtime. For instance:

```
X = X + 1.
```

would cause the integer constant 1. to be converted to a real value prior to adding it to X. This extra conversion can be eliminated by embedding the decimal within the number as shown below

```
X = X + 1.0
```

In actual practice there is very little difference in execution speed. A similar situation exists in the following statements

```
Y% = X% + 1.0
```

In this case the X% is converted to a real number prior to the addition ot the real constant. The result is then converted back to an integer prior to assignment to Y%.

In general, the programmer should avoid mixed mode expressions when possible, and should not use real constants with integer variables. Most whole numbers used in a program will be stored as integers. This normally provides the most efficient execution.

If an overflow occurs during an operation between real values, a warning is printed and execution continues with the result of the operation set to the largest real number.

In the case of integers no checking for overflow is performed since this would reduce the efficiency of integer operations. It should be understood that if the results of an integer operation fall outside the range of integer values, the calculated value will be incorrect.

3.6 ASSIGNMENT STATEMENTS

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

The expression is evaluated and assigned to the variable appearing on the left side of the equal sign. The variable and expression must either both be o type string or both be a numeric type. If the variable and expression are both numeric but one is integer and the other is real, an automatic conversion to the type of the variable on the left of the equal sign is performed.

Examples:

```
100 LET A = B + C
X(3,POINTER%) = 7.32 * Y + X(2,3)
SALARY = (HOURS.WORKED * RATE) - DEDUCTIONS
date$ = month$ + " " + day$ + " " + year$
INDEX% = INDEX% + I
REC. NUMBER = OFFSET% + NEXTREC%
```

4. Control Statements

4.1 GOSUB STATEMENT

[<stmt number>] GOSUB <stmt number>

[<stmt number>] GO SUB <stmt number>

The location of the next sequential instruction is saved on the return stack. Control is then transferred to the statement labeled with the statement numbe following the GOSUB.

Subroutine calls may not be nested greater than 20 deep.

Examples:

```
GOSUB 700
```

```
PRIN            "BEFORE TABLE"
GOSUB 200       REM PRINT THE TABLE
PRIN            "AFTER TABLE"
STOP

200 REM PRINT THE TABLE

FOR INDEX% = I TO TABLE.SIZE%
    PRINT TABLE(INDEX%)
NEXT INDEX%
RETURN
```

4.2 RETURN STATEMENT

[<stmt number>] RETURN

The RETURN statement causes the execution of the program to return to the statement that immediately follows the most recently executed subroutine call. That is, execution continues at the location at the top of the return stack. The call may be a GOSUB statement, ON. . GOSUB statement, or multiple line function call. See Chapter 8 for a discussion of multiple line functions. Refer also to section 4.12 for information on the effect of CHAINING on subroutine linkage.

If a return is executed without previously executing a GOSUB, ON OSUB, or multiple line function call, a runtime error occurs.

Examples:

```
500 RETURN
IF ANSWER.VALID% THEN RETURN
```

4.3 OTO STATEMENT

[<stmt number>] GOTO <stmt number>
[<stmt number>] GO TO <stmt numb r>

Execution continues at the statement labeled with the statement number following the GOTO or GO TO. If the statement number branched to is not a executable statement, execution continues with the next executable statement after the statement number.

If the statement number to which control is being transferred does not exist, an error will result.

Examples:

```
80 GO TO 35
GOTO 100. 5
```

4.4 IF STATEMENT

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

If the value of the expression is not zero, the statements which make up the first statement list are executed. Otherwise, the statement list following the ELSE is executed, if present, or the next sequential statement following the IF statement is executed.

In the second form of the IF statement, when the expression is not equal to zero, an unconditional branch to the statement number occurs. Note that this form of the IF statement may not have an else clause. This variation is included in CBASIC for compatibility with previous versions of Basic.

The expression in an IF statement will normally be a logical expression. That is, it evaluates to either true (- 1) or false (0). However, CBASIC will accept any numeric expression treating a value other than zero as true. The expression should be of type integer. This will reduce execution time and also reduce the size of the intermediate file generated by the compiler. If the expression is real, the value is rounded and converted to an integer. A string expression will result in an error.

A statement list is composed of one or more statements in which each pair of statements is separated by a colon (:). 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. In other words IF statements may not be nested.

Examples:

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

IF DIMENSIONSWANTED% THEN PRINT LENGTH, HEIGHT

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

IFX> 3THENX = 0: Y = 0: Z = 0

IF YES% = TRUE% THEN PRINT MSG$(I)
ELSE PRINT MSG$(2)

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

In the examples above, note that the colon is used to separate statements within a statement list and the backslash (\) is used to continue a statement onto another line.

Since the compiler ignores anything following and on the same line with the backslash, comments may be inserted without using the keyword REM.

4.5 WHILE STATEMENT

[<stmt 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. I the value is zero initially the statements between the WHILE and WEND wil not be executed. Variables used in the WHILE expression may change during execution of the loop.

The expression should be of type integer. This will reduce execution tim and also reduce the size of the intermediate file generated by the compiler. I the expression is real, the value is rounded and then converted to an integer. A string expression will result in an error.

4.6 WEND STATEMENT

[<stmt number>] WEND

A WEND statement denotes the end of the closest unmatched WHILE statement. A WEND statement must be present for each WHILE statement in a program.

Branching to a WEND statement is the same as branching to its corresponding WHILE statement.

Examples:

```
WHILE - I
    PRINT "X"
WEND

WHILE X > Z
        PRINT X
    X = X - 1.0
```

```
WEND

TIME = 0.0
TIME.EXPIRED% = FALSE%
WHILE TIME < LIMIT
        TIME = TIME + 1.0
        IF CONSTAT% THEN
RETURN REM ANSWERED IN TIME
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
```

4.7 FOR STATEMENT

[<stmt 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 expression is positive, the loop exit criteria is met when the index exceeds the value of the TO expression. If the step expression 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. The type of the index and all expressions should be the same. They may be either real or integer. If any of the expressions are of type string, an error occurs. Particular care should be taken to insure proper matching of the expression types.

For instances:

```
FOR 1% = I to DONE
```

will generate unnecessary code because DONE is real but 1% and I are integers. A more subtle example is:

```
FOR I = 1. to DONE
```

In this case I and DONE are real but 1. is an integer.

There is one situation when a FOR statement that appears to be valid wil generate a compiler error "FE". This occurs if the type of the expression following the TO is not the same as the type of the loop index variable.

For example:

```
FOR I = I TO 13 STEP 3
```

results in an error "FE" because the index variable I is real but the value following the TO is an integer. Changing the index to 1% will eliminate the error.

If the STEP clause is omitted, a default value of one is assumed. The type of the STEP expression in this case will be the same as the type of the index.

The statements within a FOR loop are always executed at least once.

Examples:

```
FOR INDEX% = I TO 10
    SUM ~ SUM + VECTOR(INDEX%)
NEXT INDEX%

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 much faster since fewer runtime checks will be made. In addition, less intermediate code is produced.

The speed of execution will also be substantially improved if all the expressions are of type integer.

4.8 NEXT STATEMENT

[<stmt number>] NEXT [<identifier> (,<idenfifier>)]

A NEXT statement denotes the end of the closest unmatched FOR statement. If the optional identifier is present, it must match the index variabl of the FOR statement being terminated.

The list of identifiers allows terminating multiple FOR statements. The statement 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 o identifiers:

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

The final example shows the use of a NEXT statement without an identifier.

```
FOR LOOPS% = I TO ARRAY.SIZE%
        GOSUB 200
        GOSUB 300
NEXT
```

4.9 ON STATEMENT

[<stmt number>] ON <expression> GOTO <stmt number> 1, <stmt number>j

[<stmt number>] ON <expression> GOSUB <stmt number> 1, <stmt number>j

The expression is used to select the statement number at which execution will continue. If the expression evaluates to 1, the first statement number is selected, and so forth. In the case of an ON . . GOSUB statement the address of the statement following the ON statement is saved on the return stack. A runtime error occurs if the expression is less than one or greater than the number of statement numbers in the list.

The expression must be numeric. A string expression will generate an error. Integer expressions will improve execution speed. If a real value is used, it is rounded to the nearest integer prior to selecting the statement number to branch to.

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

Examples:

```
ON 1% GOTO 10, 20, 30

ON J% - I GO SUB 12.10, 12.20, 12.30, 12.40

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
        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
    ELSE GOSUB 400 REM ERROR - RETRY
WEND
```

4.10 STOP STATEMENT

[<stmt 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

[<stmt number>] RANDOMIZE

The RANDOMIZE statement initializes or seeds the random number generator. The time taken by the operator to respond to an INPUT statement (chapter 5) is used to set the seed. This time will vary with each execution of a program. Therefore, for RANDOMIZE to work correctly, it must be preceded by an INPUT statement.

The configuration 3 runtime package uses the real-time clock to seed the random number generator when operating under MP/M.

Examples:

```
450 RANDOMIZE

RANDOMIZE
```

4.12 CHAIN STATEMENT

[<stmt number>] CHAIN <expression>

The CHAIN Statement transfers control from the program currently being executed to the program selected, by the expression. The expression must be of type string or an error will occur. The expression must also evaluate to any unambiguous file name. A file with that name and of type INT must reside on the specified drive. If no drive is specified, the currently logged-in drive is used. In the discussion on chaining the first program executed is the main program.

The following statement:

```
CHAIN "B:PAYROLL"
```

will cause execution to continue with the first statement in the progra PAYROLL. PAYROLL.INT must reside on drive B. Regardless of the file type specified, a type of INT is forced.

The CBASIC runtime monitor maintains four partitions in memory. They are designated the constant, code, data statement, and variable areas. The size of these areas is determined by the compiler. If in a chained program one or more of these areas is larger than that corresponding area in the original or main program, a runtime error occurs. In other words the main program constant, code, data statement, and variables areas must be as large or large than any corresponding area in a program that is subsequently chained. If this is not the case, the programmer must use the %CHAIN compiler directive to adjust the size of the main programs partitions. The %CHAIN directive is discussed in Chapter 12.

In order to determine the size of each partition in a program the compiler produces a table of these values after each compilation. The values include the effect of the %CHAIN directive if present. The %CHAIN directive need only be used in the main program. The relationship of partition size between programs chained is not significant.

A CHAIN statement may appear in any program. A program may chain back to the program which invoked it, to a new program, or to itself. If STOP statement is executed in any program, execution stops and control is returned to CP/M.

Upon execution of a CHAIN statement the return stack is reset. All ope files are closed and a restore is performed. Data may be passed from one program to another using the COMMON statement discussed below.

4.13 COMMON STATEMENT

[<stmt number>] COMMON <variable> <variable>

If present, COMMON statements must be the first statements in a program except that blank lines and REM statements may precede COMMON statements. A COMMON statement is a non-executable statement and specifies that the variables listed will be common to the main program and al programs executed through a CHAIN statement.

If the main program contains COMMON statements, each chained program must have COMMON statements that match the COMMON statements in the main program. Matching means that there are the same number of variables in each COMMON statement and, that the type of each variable in the main program's COMMON statement matches the type of each variable in the chained program's COMMON statement. Also, dimensioned variables must have the same number of subscripts in each program.

Subscripted variables are specified by placing the number of subscripts in parenthesis following the array name. For instance:

COMMON X, Y, A(3), B$(2)

specifies that X and Y are nonsubscripted real variables and will be common to all chained programs. A and 13$ are arrays which may be accessed by al programs. A has three subscripts while B$ has two. The COMMON Statement does not indicate the size of any subscript.

The specification of an array in a COMMON statement is not, in general, the same as the specification in a DIM statement. This point must be clearly understood.

For example:

COMMON A(3)
might be used with

DIM A (20,30,20)
but if it was used with

DIM A(3)
an error would occur.

Prior to accessing an element in an array in COMMON, the array must be created using the DIM statement. Failure to do this will lead to catastrophic results! The first program requiring access to the array should insure that a DIM statement is executed specifying the desired range for each subscript. Subsequent programs may access this array with the data remaining unchanged through the chaining process. If a subsequent program executes a DIM statement for this array, the data in the array will be lost. In other words the array will be re-initialized. However, in the case of string arrays, elements in the array will not be freed from memory. The programmer should set elements of string arrays to null strings prior to executing a second DIM statement for the array.

5. Input/Output Statements and Functions

5.1 General Information

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

CBASIC prints each c haracter as it is generated. If the length of the line being printed exceeds the width of a print line, printing continues on the next line. That is, a carriage return and a linefeed are output. The width of the print line may be controlled by the user.

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. A control-C entered from the keyboard may return the user to CP/M without closing open files.

In this manual console refers to the physical device assigned to the CP/M logical device CON:. Likewise the list device refers to the physical unit assigned to the CP/M logical device LST:. For more information on logical and physical devices refer to the Digital Research publication "An Introduction To CP/M Features and Facilities. -

5.2 PRINT Statement

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

The PRINT statement outputs the value of each expression to the console unless an LPRINTER statement (described below) is in effect. In the latter case output is directed to the list device (see section 5.3). If the length of a numeric item would result in the line width being exceeded, the number to be printed begins on the next line. Strings are output until the line width is reached and then the remainder of the string, if any, is output on the next line.

The delimiter 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. I this spacing results in a print position greater than the currently specified width, printing continues on the next line. A semicolon causes one blank to be output after a number and no spacing to occur after a string.

A arriage return and a linefeed are automatically printed when the end of a print statement is encountered unless the last expression is followed by a comma or a semicolon. These partial lines are not terminated until one of the following conditions occur:

(1) anotherPRINT whose list does not end in either a comma or semicolon is executed,
(2) the line width is exceeded,
(3) an I-PRINTER or CONSOLE statement is executed, or
(4) the program executes a stop statement. A PRINT statement with no expression list will cause a carriage return and a linefeed to be printed.

If execution of a program is terminated due to an error, a carriage return and a linefeed are output.

Examples:

```
PRIN
PRINT AMOUNT.PAID
PRINT QUANTITY, PRICE, QUANTITY * PRICE
PRINT "TODAY'S DATE IS: ";MONTH$;" ";DAY%;-, ";YEAR%
```

5.3 LPRINTER STATEMENT

[stmt number>] I-PRINTER [WIDTH <expression>]

After execution of the I-PRINTER statement all PRINT statement output, which would normally be directed to the console, will be output on the list device. The list device is the physical unit currently assigned to LST: by CP/M. The WIDTH clause is optional. If present the expression will be used to set the line width of the list device.

If the console's cursor position is not 1, a carriage return and linefeed is output to the console. In this context the cursor position is the value that would be returned by the POS function (see section 5.5) just prior to executing the I-PRINTER statement.

The expression should be of type integer. If it is real, the value is rounded to an integer. An error occurs if the expression is of type string.

If the width option is not present, the most recently assigned width is used. Initially the width is set to 132. A width of 0 will result in an infinite line width. With a zero width in effect carriage returns and linefeeds are never automatically output to the printer as a result of exceeding the line width.

Examples:

```
500 I-PRINTER

IF HARDCOPY. WANTED% THEN I-PRINTER WIDTH 120

LPRINTER WIDTH REQUESTED. WIDTH%
```

5.4 CONSOLE STATEMENT

[stmt number>] CONSOLE

Execution of the CONSOLE statement restores printed output to the console. the console is the physcial unit currently assigned to CON: by CP/M.

If the list device print position is not 1, a carriage return and linefeed are output to the list device.

Examples:

```
490 CONSOLE

IF END.OF.PAGE% THEN
    CONSOLE:
    PRINT USING "##,### WORDS THIS PAGE";WORDS%:\
    INPUT "INSERT NEW PAGE, THEN CR-;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 base 10 or I I OH. The new console width will become effective at the next execution of a CONSOLE statement. The console line width is initially set to 90 (5014).

A width of zero (0) results in an infinite width. With a zero width in effect, carriage returns and linefeeds are never automatically output to the console as a result of exceeding the line width.

5.5 POS PRE-DEFINED FUNCTION

POS

POS returns the next position to be printed on either the console or the line printer. This value will range from I to the line width currently in effect.

If a LPRINTER statement is in effect, POS will return the next position to be printed on the printer. Note that POS returns the actual number of characters sent to the output device. If cursor control characters are transmitted, they are also counted even though the cursor is not advanced.

Examples:

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

IF (WIDTH. LINE - POS) < 15 THEN PRIN
```

5.6 TAB PRE-DEFINED FUNCTION

TAB (<expression>)

TAB causes the cursor or print head to be positioned to a position specified by the value of the expression. If the value of the expression is less than or equal to the current print position, a carriage return and linefeed are output and then the tab is executed.

The TAB function is implemented by outputting blank characters until the desired position is reached. If cursor or printer control characters have been output, the cursor or print head could be positioned incorrectly.

The TAB function may only be used in PRINT statements.

The expression must be numeric. If a string expression is specified, an error occurs. If the expression is real, it is first rounded to an integer. If the expression is greater than the current line width, an error occurs.

Examples:

```
PRINT TAB(15);"X

PRINT "THIS IS COL. I -;TAB (50); "THIS IS COL. 50"

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

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

5.7 READ STATEMENT

[stmt number>] READ <variable> 1, <variable>

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

Examples:

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

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

5.8 DATA STATEMENT

[stmt number>] DATA <constant> 1, <constant>l

DATA statements are nonexecutable statements which define string, floating point, and integer constants which are assigned to variables using a READ statement. Any number of DATA statements may occur in a program. They may be placed anywhere in the program.

The constants are stored consecutively in a data area as t hey 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 may not be continued with a continuation character. However, all DATA statements in a program are treated collectively as a concatenated list of constants separated by commas.

Examples:

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

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

In the second example ONE, TWO and THREE are strings.

If a real constant is assigned to an integer variable with a READ statement, the constant is rounded off to the integer portion of the real number. If the value of number assigned to an integer is outside the range of CBASIC integers, incorrect values will be assigned. If a real number exceeds the range of real numbers, an overflow warning occurs and the largest CBASIC number is used in its place.

5.9 RESTORE STATEMENT

[stmt number>] RESTOR

A RESTORE statement repositions the pointer into the data area so that the next value read wth a READ statement will be the first item in the first DATA statement.

The purpose of a RESTORE statement is to allow re-reading the constants contained in DATA statements.

Examples:

500 RESTORE

IF END.OF.DATA% THEN RESTORE

When a CHAIN statement is executed a RESTORE is performed.

5.10 INPUT STATEMENT

[stmt number>] INPUT [<prompt string> ;1 <variable> f, <variable>1

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 variables may be either simple or subscripted string or numeric variables.

At most 255 characters may be entered in response to an INPUT statement. If 255 or more characters are entered, inputing is automatically terminated and the first 255 characters are retained. Additional characters will be lost. The 255 characters include all characters entered in response to an input statement no matter how many varibles appear in the variable list.

All CP/M line editing functions such as control-U and rubout are in effect. A control-C may terminate the program without closing open files. If a control-Z is the first character entered in response to an INPUT statement the program i terminated in the same manner as if a STOP statement had been executed.

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.

The prompt string must be a string constant. If it is an expression or a numeric constant, an error occurs.

If the value entered for assignment to an integer is real, the number entered is truncated to the integer portion of the real number. If the value of a number assigned to an integer variable is outside the range of integers, an incorrect value will be assigned. If a real number exceeds the range of CBASIC real numbers, the largest real number is assigned to the variable, and a warning is printed on the console.

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

Examples:

INPUT AMOUNTI, AMOUNT2, AMOUNT3

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

INPUT "YOUR PHONE NUMBER PLEASE:-, PHONE.N$

INPUT " ";ZIP.CODE%

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

[stmt number>] INPUT [<prompt string> j LINE <variable>

This statement functions as described above with the following exception. Only one variable is permitted following the keyword LINE. It must be of type string. Any data entered from the console is accepted and assigned to this variable. The data is terminated by a carriage return.

A null string may be accepted by responding to an INPUT LINE Statement with a carriage return.

An error occurs if the variable specified to receive the inpu t is not of type string.

Examples:

INPUT "ENTER ADDRESS";LINE ADDR$

INPUT "TYPE RETURN TO CONTINUE-;LINE DUMMY$

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

5.11 OUT STATEMENT

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

The low-order eight bits of the second expression are sent to the CPU output port selected by the low-order eight bits of the first expression.

Both arguments must be numeric; they should be in the range of 0 to 255 for the results to be meaningful. An error occurs if either expression is of type string. Real values are converted to integers prior to performing an OUT instruction.

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 PRE-DEFINED FUNCTION

INP (<expression>

INP returns the value input from the CPU 1/0 port specified by the expression. This function is useful for accessing peripheral devices directly from the CBASIC program.

The argument must be numeric. An error occurs if it is a string. A real value will be rounded to the nearest integer. For the results to be meaningful, the argument must be in the range of 0 to 255.

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
```

5.13 CONSTAT% PRE-DEFINED FUNCTION

CONSTAT%

CONSTAT% returns the console status as an integer value. If the console device is ready, a logical true is returned otherwise a logical false is returned.

Examples:

```
IF CONSTAT% THEN
    GOSUB 100 REM PROCESS OPERATOR INTERRUPT

WHILE NOT CONSTAT%
WEND
```

5.14 CONCHAR% PRE-DEFINED FUNCTION

CONCHAR%

CONCHAR% reads one character from the console device. The value returned is an integer. The lower eight bits of the returned value are the binary representation of the ASCII character input. The high order eight bits are zero.

Examples:

```
1% = CONCHAR%
CHAR% = 0
IF CONSTAT% THEN
    CHAR% = CONCHAR%
IF CHAR% = STOPCHAR% THEN \
    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 is an integer ranging 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. An error occurs if a string expression i specified. Real values are rounded to the nearest integer.

Examples:

```
100 MEMORY% = PEEK(l)

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

6.2 POKE Statement

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

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

Both expressions must be numeric. An err or occurs if a string expression is specified. Real values are rounded to the nearest integer.

Examples:

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

```
FOR LOC% = I TO LEN(OUT.MSG$)
    POKE MSG.LOC% + LOC%, ASQM1D$(OUT.MSG$,LOC%, 1))
NEXT LOC%
```

6.3 CALL Statement

[<stmt 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, with the exception of the stack-pointer, they need not be restored prior to returning.

The expression must be numeric. An error occurs if a string expression is used. Real values are rounded to the nearest integer.

Examples:

```
CALL 5H

2000 CALL ANALOG.INPUT%

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

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

6.4 SAVEMEM Statement

[<stmt number>] SAVEMEM <constant> , <expression>

The SAVEMEM statement reserves space for a machine language subroutine, and loads the specified file during execution. Only one SAVEMEM statement may appear in a program.

The constant must be an unsigned integer which specifies the number of bytes of space to reserve for machine language subroutines. The space is reserved in the topmost (highest) address space of the CP/M transient program area. The beginning address of the reserved area is calculated by taking the constant specified in the SAVEMEM statement and subtracting it from the 16 bit address stored by CP/M at absolute address 6 and 7.

The expression must be of type string and may specify any valid unambiguous file name. The selected file is loaded into memory starting with the address calculated above. Records are read from the file until either an end of file is encountered or the next record to be read would over-write a location above the transient program area.

If the constant specifies less than 128 bytes to be saved, nothing will be read in, but the space will still be reserved. If the expression is a null string, space is saved but no file is loaded.

If a main program has a SAVEMEM statement, any chained program that has a SAVEMEM statement must reserve the same amount of space. Each chained program may load a new machine language file, or it may use the file loaded by a previous program. The space reserved by the main program may not be reclaimed by a subsequent program.

It is the programmers responsibility to insure that the machine language routines are assembled to execute at the proper address. In addition, it should be noted that the location at which a program is loaded is dependent upon the size of the CP/M system being used.

Examples:

```
SAVEMEM 256, "SEARCH.COM"

SAVEMEM 512, DR$+ "CHECK." + ASSY$ (FN.CPM.SIZE%)
```

6.5 Use of Integers

Although all the machine language linkage statements will accept either real or integer values where an expression is required, it is much more efficient to use integer quantities. The size of the INT file will be reduced, and the program will execute faster.

Since the largest positive CBASIC integer is 32767, the use of integer variables to address the upper 32K of memory requires that the desired address be converted to an appropriate negative number. Remember that in 2's complement representation of binary numbers a - 1 is 16 1's. This is most easily overcome by expressing addresses as either hexidecimal or binary constants. For instance, if a programmer desires to call an assembly language program at 48000 decimal, the following instruction will accomplish this:

```
CALL 0C0000H
```

7. Predefined Functions

This chapter describes predefined functions provided by CBASIC. Predefined functions are used to build expressions as explained in section 3.5. When a predefined function has arguments, the arguments may be any valid expression which evaluates to the correct type, either numeric or string.

In general, when a numeric expression is required, real and integer arguments may be used interchangeably. However, efficiency is improved by using expressions as arguments that do not require conversion. In the definitions below, string arguments are represented by A$, B$, etc, integers by 1%, J% etc, and real values by X, Y, etc.

Some predefined functions are discussed in chapters 5 and 6.

7.1 Numeric Functions

The following functions return numeric values. Arguments, when required, may be any expression that evaluates to either a floating point or integer number.

FRE

FRE returns the number of bytes of unused space in the free storage area. The value returned is a floating point number.

```
X = FRE

IF FRE < 500.0 THEN GOSUB 10 REM PRINT WARNING
```

ABS(X)

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

The value returned by ABS is a floating point number. If X is a string an error occurs. If X is an integer, it is first converted to a floating point number.

```
DISTANCE = ABS(START-FINISH)

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

INT(X)

INT returns the integer part of the argument X. The fractional part is truncated.

The value returned is a floating point number. If X is a string expression, an error occurs. If X is an integer, it is first converted to a real value.

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

INT%(X)

INT% converts the argument X to an integer value. If X is a string, an error will occur. If X is an integer, it is first converted to a real value, and then it is converted back to an integer.

```
J% =INT%(REC.NO)

WIDTH% = DIMEN.1% + INT%(DIMEN.2)
```

FLOAT(I%)

FLOAT converts the argument 1% to a real value. If 1% is a string, an error occurs. If 1% is real, it is first converted to an integer, and then it is converted back to a real number.

```
AMOUNT      FLOAT(COST%)
POSITION    SIN(FLOAT(ANG%) * OFFSET
```

RND

RND generates a uniformly distributed random number between 0 and 1. The value returned is a real number.

To avoid identical sequences of random numbers each time a program is executed, the RANDOMIZE statement must be used to seed the random number generator.

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

IF RND > .5 THEN
   HEADS%      TRUE%
ELSE \
   TAILS% TRUE%
```

SGN(X

SGN returns an integer value that represents the algebraic sign of the argument. It will return - I if X is negative, 0 if X is zero, and + I if X is greater than zero.

X may be either integer or real. Integer values of X are converted to real numbers. If X is a string, an error occurs. SGN always returns an integer.

```
IF SGN(BALANCE <> 0 THEN
    OUTSTANDINGBAL% = TRUE%

IF SGN(BALANCE = - I THEN
    OVERDRAWN% = TRUE%
```

ATN(X)

ATN returns the arctangent of X. Using simple identities, other inverse trigonometric functions may be computed from the arctangent. The argument i expressed in radians.

The value returned is real. If X is an integer, it is first converted to a real number.

```
X = ATN(RADIANS)

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

ASIN      ATN(X/(SQR(l. - X*X)))

ACOS         P1/2. - ATN(X/SQR(I - X*X))
```

COS(X

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

The value returned is real. If X is an integer, it is first converted to a real value.

```
IF COS(ANGLE) = 0.0 THEN VERTICAL% = TRUE%
PRINT CONSTANT * COS(ROTATION)
```

EXP(X)

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

The value returned is real. If X is an integer, it is first converted to a real number.

```
Y = A * EXP(BX%)

E = EXP(l) REM CONSTANT E = 2.7182 .....
LOG(X)
```

The natural or Naperian logarithm of the argument X is returned by LOG.

The value returned is real. If X is an integer, it is first converted to a real number.

```
BASE.TEN.LOG = LOG(X)/LOG(IO)
PRINT "LOG OF X IS "; LOG(X)
```

SIN(X)

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

The value returned is real. If X is an integer, it is first converted to a real number.

```
FACTOR(Z) = SIN(A - B/C)
IF SIN(ANGLE/(2.0 * P1)) = 0.0 THEN\
    PRINT "HORIZONTAL"
```

SQR(X

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

The value returned is real. If X is an integer, it is first converted to a real number.

```
HYPOT = SQR((SIDEI-2.0)+(SIDE2^2.0))
PRINT USING \
    "THE SQR ROOT OF X IS: ####.##"; SQR(X)
```

TAN(X)

TAN returns the tangent of the argument X. X is expressed in radians.

The value returned is real. If X is an integer, it is first converted to a real number.

```
    POWER.FACTOR TAN(PHASE. ANGLE)
QUIRK = TAN(X 3.0 * COS(Y))
```

7.2 STRING FUNCTIONS

ASC(A$)

ASC returns the ASCII numeric value (in decimal) of the first character of the string argument. If the length of A$ is zero (null string), a runtime error wil occur.

The value returned is an integer. If the argument is numeric, an error wil occur.

```
IF ASC (DIGIT$)>47 AND ASC(DIGIT$)<58 THEN
    PRINT "VALID DIGIT"
    OUT TAPE.PORT%, ASQ-*-)
```

CHR$(I%

CHR$ returns a one character string consisting of the character whose ASCII equivalent is 1%. 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.

The value retumed is a string. If 1% is real, it is first converted to an integer value.

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

PRINT CHR$(BELL%) REM ring the bell!
```

LEFT$(A$,I%)

LEFT$ returns a string consisting of the first 1% ch aracters of A$. If 1% is greater than the length of A$, the entire string will be returned. If 1% is zero, a null string will be returned; if 1% is negative, a runtime error will occur.

A$ must be a string; otherwise an error will occur. 1% should be nu eric. If 1% is real, it will first be converted to an integer. If 1% is a string, an error wil occur.

```
PRINT LEFT$(INPUT.DATA$,25)
IF LEFT$(IN$, 1) = ';Y" THEN GOSUB 400
```

LEN(A$)

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

The value returned by LEN is an integer. An error occurs if the argument is numeric.

```
IF LEN(TEMPORARY$) > 25 THEN
    TOO.LONG% = TRUE%

FOR INDEX% = I TO LEN (OBJECT$)
    NUM%(INDEX%) = ASC (MID$(OBJECT$,INDEX%,I))
NEXT INDEX%
```

UCASE$(A$)

UCASE$ returns a string in which the lower case characters in A$ have been translated to uppercase. Other characters are not altered. A$ remains unchanged unless A$ is set equal to UCASE$(A$).

The value returned by UCASE$ is a string. An error occurs if A$ is numeri .

```
    IF UCASE$(ANS$) = "YES" THEN
        RETURN\
ELSE STOP

NAME$ = UCASE$(NAME$)
```

MATCH (A$,B$,I%)

MATCH returns the position of the first occurrence of A$ in 13$ starting with the character position given by 1%. 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.

4) A backslash (\) character serves as an escape character to indicate the character that follows does not have special meaning. For instance a question mark signifies that any character is a match unless preceeded by a backslash.

A$ and B$ must be strings. An error will occur if either of these arguments are numeric. If 1% is real, it will first be converted to an integer; if 1% is a string, an error will occur. If 1% is negative or zero, a runtime error will occur. When 1% is greater than the length of B$, zero is returned. If B$ is a null string a 0 is always returned. If 13$ is not null but A$ is null a I will be returned.

Examples:

MATCH(- is"," Now is the",I) returns 5

MATCH(- # # ", - October 8, 1976 ", 1) returns 12

MATCH(- a?"," character",4) returns 5

MATCH("4"," 123#45",1) returns 4

MATCH(- ABCD ", "ABC ",I) returns 0

Note that the third example returns a 5 instead of a 3 because the starting position for the match is position 4. In example four the backslash causes the pound sign to match only another pound sign. Without the backslash a I would be returned.

The next example is a more complicated statement using the backslash

MATCH (-\# I\\\?", -I# 1\?2#", 1) returns 2

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

```
    TRUE% -I
    FALSE% 0
    edit$ =    - The number of occurrences is ###"
    WHILE TRUE%
        INPUT-enter object string" ; LINE object$
        INPUT "enter argument string" ; LINE arg$
        GOSUB 620
        PRINT USING edit$; occurrence%
WEND

620 rem ----- count occurrences ---------
location% = I
occurrence% = 0
WHILE TRUE%
    location% = MATCH(arg$,object$, location%)
    IF location% = 0 THEN RETURN
        occurrence% = occurrence% +
```

```
        location% = location% + I
    WEND
    END
```

MID$(A$,I%,J%)

MID$ returns a string consisting of the J% characters of A$ starting at the 1% character. If 1% is greater than the length of A$, a null string is returned. If J% i greater than the length of A$, all the characters from 1% to the end of A$ are returned. An error occurs if either 1% or J% is negative. A runtime error also occurs if 1% is zero. A zero value of J% will return a null string.

A$ must be a string expression; otherwise an error will occur. 1% and J% must be numeric. If 1% or J% are real, they will first be converted to integers; if either I% or J% are strings, an error will occur.

```
DIGIT$=MID$(0I3JECT$,P0S%,I)

DAY$=MID$("MONTUEWEDTHUFRISATSUN", DAY%*3-2,3)
```

RIGHT$(A$,I%)

RIGHT$ returns a string consisting of the 1% rightmost characters of A$. If I% is negative, a runtime error occurs; if 1% is greater than the length of A$, the entire string is returned. If 1% is zero, a null string is returned.

A$ must evaluate to a string; otherwise an error will occur. 1% must be numeric. If 1% is real, it will first be converted to an integer; if I% is a string, an error will occur.

```
IF RIGHT$(ACCOUNT.NO$, 1) = -0- THEN\
    TITLE.ACCT% = TRUE%

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

STR$(X)

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

If X is a string, an error will occur. If X is an integer, it will be converted to a real value.

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

VAL(A$)

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

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

A$ must be a string; otherwise an error will occur.

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

ON VAL(PROG.SEL$) GOSUB 10, 20, 30, 40, 50
```

COMMAND$

COMMAND$ returns a string which contains the CP/M command line modified as described below. Refer to Digital Research publication "CP/M Interface Guide" for a discussion of the Command Line.

The name of the program being executed is not included in the string returned by COMMAND$. In addition, if the TRACE option is used with CRUN, the word TRACE and associated line numbers, if present, will not be included. If any of the following commands are used to execute a CBASIC intermediate file:

```
CRUN2 PAYROLL NOCHECKS TOTALS

CRUN2 PAYROLL TRACE NOCHECKS TOTALS
```

the COMMAND$ function will return the following string

NOCHECKS TOTALS

Leading blanks are removed. A maximum of 50 characters will be etained by the COMMAND$ function. All alphabetic characters are converted to upper case.

THE COMMAND$ function may be used at anytime in a program, as many times as desired, and by any program which is subsequently loaded by a CHAIN Statement.

SADD(A$)

SADD returns the address of the string assigned to the argument A$. The first byte is the length of the string followed by the characters in the string. The length is stored as an unsigned binary integer.

Therefore, if the string is "TOTAL", the SADD function would return the address of a byte containing a binary 5. The next byte would be an ASCII "T" etc.

The value returned by SADD is an integer. If A$ is not a string, an error occurs. When the parameter evaluates to a null string, a zero may be returned.

The SADD function, in conjuncti on with PEEK and POKE, may be used to pass a string to an assembly language routine for processing.

The following statements will put the address of STRING$ into the address stored in PARM.LOC%:

```
POKE PARM.LOC%,SADD(STRING$) AND OFFH
POKE PARM.LOC% + 1,SADD(STRING$)/256
```

VARPTR (<variable>)

VARPTR returns the permanent storage location assigned to the <variable> by the run-time monitor.

In the case of an unsubscripted numeric quantity, this is the actual location o the variable in question. For string variables, the value returned is the address of a sixteen-bit pointer to the referenced string. Because strings are dynamically allocated the actual location of the string may vary, but the value returned by VARPTR remains unchanged during execution of a program. If the variable is in common, then the location returned by VARPTR will remain unchanged after chaining.

If the <variable> is subscripted, the value returned by VARPTR is the address of a pointer to the array dope vector in the free storage area. The array foilows the dope vector. The first byte of the dope vector is the number of dimensions. Following this single byte are n - I (n is the number of dimensions) 16 bit offsets into the arrray. The final 16 bit quantity in the dope vector is the number o entries in the array. The array follows in row major order.

SIZE(A$)

SIZE returns the size in blocks of the file specified by A$. If the file is empty or does not exist, zero is returned. A$ may be any CP/M ambiguous file name. Digital Research publication "An Introduction to CP/M Features and Facilities" explains ambiguous file names.

The argument must be a string expression. A numeric value will result in an error. The SIZE function returns an integer.

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 I character will occupy a full block of space. This means the SIZE function returns the amounts of space that has been reserved by the file rather than the size of the data that is in the file.

7.3 DISK FUNCTIONS

RENAME(A$,B$)

RENAME is a function that changes the name of the file specified by B$ to the name given by A$. Renaming a file to a name that already exists produces a runtime error.

The RENAME predefined function returns an integer value. A true (- 1) is returned if the rename is successful and a false (0) is returned in cases where the rename fails. For instance false is returned if B$ does not exist.

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 a runtime error because the original file name will no longer exist in the CP/M file directory.

Both arguments must be of type string. If either A$ or B$ is numeric an error will occur.

The RENAME function will allow a CBASIC programmer 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.

Examples:

```
DUMMY% = RENAME ("PAYROLL. MST", "PAYROLL.$$$")

IF RENAME (NEWFILE$,OLDFILE$) THEN RETURN
```

8. User Defined Functions

Functions or subprograms are defined by a programmer when the same computation is to be performed in a number of locations within a program. The required routine is coded as a function and then referenced or called from any location within the program. The function may be passed values or parameters to be used in each invocation.

All CBASIC functions return a value. Thus, the function is, in effect, a reference to a routine which results in a value, either string or numeric. CBASIC provides two types of functions, single statement and multiple statement.

A function must be defined prior to any reference to the function. That is, the compiler must encounter the function definition prior to any reference to the function.

8.1 FUNCTION NAMES

The name of a user defined function must begin with 'FN' followed by any combination of numbers, letters and periods. A function name may be any length. Only the first 31 characters are considered when determining the uniqueness of a function name. No spaces are allowed between the FN and the remainder of the name.

The type of the function name determines the type of the value returned by the function. If the function name ends with a dollar sign, a string is returned; if the name ends with a percent sign, an integer is returned. Otherwise, a real value i returned. A function name is used to both define a function and to reference a function.

Examples of function names:

FN.THIS.IS.A.VALID.FUNCTION

FN3.1416%

FN.FUNCTION$

FNJIMES

FN.TRUNCATE$

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 i 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. Assume the size o "OUTPUT" will be 125% of "INPUT". The following routine will insure that space is available on disk B prior to processing.

```
70 rem ------ test for enough room -----
size.of.output% = 1.25 * size ("A:INPUT")
free.blocks% = 241 - size("B:*.*")
if free.space% < size.of.output% then
    enough.room% = FALSE% \
    else enough.room% = TRUE%
    return
```

CP/M supports 241 user accessible blocks on single density systems. The number of blocks in use, subtracted from 241, gives the remaining space on the disk.

Note that some systems, such as those with double density disk dr ives, may not provide results consistent with standard disks.

CBASIC determines the number of blocks in a file by counting the non-zero bytes in the file control block allocation map.

The body of a multiple statement function consists of any number of CBASIC statements except that DEF and COMMON statements may not appear in the body of a function. A multiple statement function may reference itself within the body of the function but, all local variables retain their most recent definition when returning from the function.

If a DIM statement appears in a multiple statement function, a new array is allocated on each execution of the DIM statement. The previous data stored in the array is lost. Note that the array is global to the entire program.

A value is returned from a multiple statement function by having the name of the function appear on the left hand side of an assignment statement. Any number of such assignments may appear in the body of a function. The most recent assignment is the value returned by the function.

The function returns when a return statement is executed. Any number of return statements may be present in the body of a multiple statement function. If no assignment is made to the function name, the value returned is the last value assigned to the function name. If no value has been assigned, zero is returned.

The body of a multiple statement function is terminated by a FEND statement. The general form of an FEND statement is:

[<stmt number>] FEND

Execution of a FEND statement implies that a multiple statement unction was exited without executing a RETURN statement. In this ase a runtime error occurs.

Examples:

```
DEF FN.READ.INPUT(INPUT.NO%)
    READ # INPUT.NO%; CUSTNO%, AMOUNT
    RETURN
FEND
```

8.2 FUNCTION DEFINITIONS

Single statement functions are defined with the DEF statement whose general form is:

[<stmt number>] DEF <function name> [(<dummy arg list>)] = <expression>

The type of the expression must match the type of the function name. here may be none, or any number of dummy arguments, and they may be used freely within the expression. A dummy argument is either 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 arguments 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.

Examples:

```
DEF FN25 = RND*25.0

DEF FN. LEFF. JUSTIFY$(A$,LEN%) = LEFr$(A$ + BLNKS$,LEN%)

DEF FN.HYPOT(SIDEI,SIDE2) = \
        I SQR((SIDE I *SIDE 1) + (SIDE2*SIDE2))

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

DEFFN.EOJ%=FLAGI% OR FLAG2% OR FLAG3% OR FLAG4%

DEF FN. INPUT%(PORT%) = INP(PORT%) AND MASK%(PORT%)
```

A multiple statement function consists of a multiple statement function definition, a function body and a FEND statement. Multiple statement function definitions use the following forrn of the DEF statement:

<stmt number>] DEF <function name> [(<dummy arg list>)]

The dummy argument list is identical to that described for single tatement functions. The parameters are local to the entire body of the function.

8.3 FUNCTION REFERENCES

A user defined function may be referenced 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. This includes integer and real parameters. If the function definition requires an integer parameter, the value passed to the function must be an integer. The same rule applies to real and string parameters.

A function must be defined prior to a reference to the function.

Prior to calling the function, the current value of each expression is substituted for the dummy variable in the definition.

Examples:

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

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

WHILE FN. ALTITUDE(CURR. ALT%) > MINIMUM.SAFE
    CURR. ALT% = INP(ALT. PORT%)
WEND
```

For example:

```
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 routine shows how this would be done:

```
FLD.S% 20
BLK$ =  11
PHONE$ "213-355-1063"
PRINT USING"#&"; RIGHT$ (BLK$ +PHONE$, FLD.S%)
```

which would output:

```
#   213-355-1063
```

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

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:

[<stmt number>] PRINT USING <format string> [<file reference>] <expression list>

A format string may be any string expression. This allows the format to be determined during program execution. An error occurs if the format string is numeric; a runtime error occurs if the expression evaluates to a null string.

The expression list consists of expressions separated by commas or semicolons. The comma does not cause automatic tabbing 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 starting at the beginning of the string.

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, if while outputting a string a numeric data field is encountered, the characters that make up the numeric data field will be treated as literal data. I there is no data field within the format string of the type required, an error wil 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 I-PRINTER 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 then written as a record in the selected file. Chapter 10 discusses in more detail the use of PRINT USING with disk files.

```
DEF FN.COUNT%(INDEXI%)
    COUNT% = 0
    FOR 1% = I TO INDEXI%
        COUNT% = COUNT% + ARRAY(I%)
    NEXT 1%
    FN.COUNT% = COUNT%
    RETURN
FEND
```

9.2 STRING CHARACTE 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$ = "Lynn": M.NAME$ = "Marion": L.NAM$="Kobi"
PRINT USING F. NAME$, M. NAME$, L. NAM$
```

would output:

```
L. M. Kobi
```

In this example, the period is treated as literal data. Since there are two expressions in the list, the format string is reused when processing 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, i necessary, padded on the right with blanks. A string, which is longer than the data field, is truncated on the right.

For example:

```
FOR I$ = "THE PART REQUIRED IS 5 . . . . 0 . . . .51"
PART.DESCRP$ = "GLOBE VALVE, ANGLE"
PRINT USING FORI$; PART.DESCRP$
```

will output:

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

The use of the periods and numbers between the slashe s makes it easy to verify that the data field is 16 characters long. They have had no effect on the output.

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.

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 rinted 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. When 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 I00E-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. When 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 forniat 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-0 I
```

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 field allows only 9 digits to be output. Using the second field, 10 digits may be output.

If the exponent option is used, commas are not printed; when commas occur in the held, they are treated as pound signs.

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. When 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 i the number is less than I 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$ = "####.####           ###.# ###11
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
```

9.6 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 NUMBEI

X = 132.71
```

PRINT USING X,X

will output:

% 132.71 132.7

10. Files

10.1 HOW CP/ AINTAINS FILES

CBASIC uses the CP/M file accessing routines to store and retrievedata from diskette files. This section will provide a brief introductionto 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. All CBASIC file accesses are performed using CP/M system calls.

The CBASIC statements used to access diskette files will now be discussed. Three statements are used to activate a file, OPEN, CREATE, and FILE. Once a file has been activated, READ and PRINT statements may access and write files respectively. An active file may be deactivated with either a CLOSE or DELETE statement. Chapter I I 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:

```
[<stmt number>] OPEN <expression> [RECL <expression>]
    AS <expression> [BUFF <expression> RECS <expression>]
```

1, <expression> [RECL <expression>
AS <expression> [BUFF <expression> RECS <expression>]

For instance:
If random access is to be used with a file, the BUFF expression, if present, must evaluate to 1; otherwise a runtime error will occur.

Both expressions must be numeric; a string value will cause an error. Rea values are converted to integers.

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 and closing them when they are no longer needed.

Examples:

```
555 OPEN "TRANSTIL" AS 9

OPEN FILE.NAME$ AS FILE.NO% BUFF 26 RECS 128

OPEN WORK. FILE. NAME$(CURRENT. FILE%)
RECL WORK.LENGTH% AS CURRENT.FILE% BUFF BS% RECS 128
```

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:

[<stmt number>] CLOSE <expression> 1, <expression>l

Each expression refers to the identification number of an active file. The fil 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 activated with a CREATE, OPEN or FILE statement.

If an IF END statement is currently associated with the identification number for the file being closed, the IF END will no longer be in effect.

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 a runtime error occurs.

Each expression must be numeric in the range I to 20. Real values are converted to integers. A string value will result in an error.

The first expression represents the name of a file on diskette. The name ma 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 expression must be of type string; an error occurs if it is numeric. The following examples show valid file names:

ACCOUNT.MST
CBASIC.COM
B:INVENTOR.BAK

The third example shows a reference to a file on drive B.

The directory on the selected drive is searched and the named file is opened. I 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. When a drive reference is present, it is the programmer's responsibility to insure such a drive is available on the system being used.

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 is not between 1 and 20, a runtime error occurs. The expression must be numeric; real values are converted to integer. A string value will cause an error.

When the optional RECL expression is present, the file will consist of fixed length records. A runtime error occurs if the record length is negative or zero. A file may be accessed randomly or sequentially when a record length has been specified; otherwise only sequential access is allowed. The RECL expression must be numeric; real values are converted to integer. A string value will cause an error.

The BUFF and RECS expressions are optional. If used, th ey both must be present. The expression following BUFF specifies the number of disk sectors from the selected file to maintain in memory at one time. If the expression is omitted, a value of one is assumed. The expression following RECS must be present when the BUFF expression is used, but the value of the expression is ignored. For possible future use, the value should be the size of a disk sector. Thi is normally 128 bytes.

If an IF END statement is currently associated with the identification number for the file being deleted, the IF END will no longer be in effect.

Examples:

```
DELETE I

DELETE FILE.NO%, OUTPUT.FILE.NO%

1% = 0
WHILE 1% < NO.OF.WORKFILES%
    1% = 1% + I
    DELETEI%
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

[<stmt number>] IF END # <expression> THEN <stmt 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 statement number following the THEN. If no IF END statement has been executed, a runtime error occurs.

The IF END statement must be the only statement on a line; it may not follow a colon nor be part of a statement list.

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. However, if a DELETE or CLOSE statement is executed, any IF END associated with the identification number is no longer effective.

The expression must be numeric in the range I to 20. Real values are converted to integers. A string value will cause an error.

When a condition exists which results in the transfer of control to the statement associated with an IF END statement, the stack is restored to the condition that existed prior to the statement which caused activation of the IF END. Thus if the statement which resulted in transfer was in a subroutine, a return must be executed after processing the end of file condition.

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 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 13, control will be transferred to statement 500.5. After a successful OPEN, an end o 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% BUFF 6 RECS 128
IF END # MASTER.FILE.NO% THEN 500
```

An IF END statement may also be used when writing to a file. I n this case control is transferred to the statement associated with the IF END when an attempt is made to write to the file and there is no disk space available. Part of the record being created may have been written to the file. When using fixed files, the last record may be rewritten after additional space is freed.

10.7 FILE STATEMENT

[<stmt number>] FILE <variable> [(<expression>)]1, <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, an error occurs. I the expression enclosed in parentheses is present, the value of the expression i the record length. The record length must be numeric. Real values are converted to integers. A string value will cause an error. The variable must not be subscripted and it must be of type string. It may not be a literal or an expression,

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 INPUT statement to access data from the console. The last two forms are similar to the INPUT LINE statement.

The general form of the sequential read is

[<stmt number>] READ # <expression> ; <variable>1 , <variable> 2

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 integer, floating point, or string values, and they are delimited by commas.

The expression, which selects the file, must be numeric. Real values are converted to integer. A string value will cause an error. In addition, the value must refer to an active file. Otherwise, a runtime error occurs.

Examples:

READ # 7; STRING$, NUMBER

READ # FILE.MASTER%; NAME$, ADDRESS$,CITY$, STATES

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

[<stmt number>] READ # <expression> , <expression> [<variable> 1, <variable> 2]

The second expression selects the record to be read. 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. To use this form of the read, the file must have been activated with the RECL option specified.

The second expression must be numeric. If the value is a string, an error will occur. Real values are converted to integers. The record number may not be zero; if it is, a runtime error will occur. The expression is treated as a sixteen bit unsigned binary number. This allows record numbers in the range of I to 65,535.

A random read with no variables specified will position the file to the selected record. A subsequent sequential read will access the selected record.

Example:

READ # FILE.NO%,REC.COUNT%; 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:

[<stmt number>] READ # <expression> ; LINE <variable>

This statement r eads 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. If the variable is not of type string, an error occurs.

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

[<stmt number>] READ # <expression> , <expression> LINE <variable>

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

The READ LINE statement permits CBASIC to acce ss 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 NEXT. LINE. OF. TEXT$

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

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

[<stmt number>] PRINT # <expression> <expression> 1, <expression>l

[<stmt number>] PRINT # <expression> ,<expression> ;<expression> 1, <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 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 expression following the pound sign must be numeric. A real value will be converted to an integer. A string value will cause an error. In addition the value must refer to an active file; otherwise a runtime error will occur.

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 sam fonnat 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 second expression must be numeric. If the value is a string, an error wil occur. Real values are converted to integers. The record number may not be zero; if it is, a runtime error will occur. The expression is treated as a sixteen bit unsigned binary number. This allows record numbers in the range of I to 65,535.

Examples:

```
PRINT # 3; "JONES, BILL"
PRINT #FILE.NO%; NAME$, ADDR$, SALARY
PRINT #PAY%,EMPLNO%; EMPL.NM (EMPLNO%),HOURS (EMPLNO%)
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:

[<stmt number>] PRINT USING <expression> # <expression> ; < xpression> 1, <expression> l

[<stmt number>] PRINT USING <expression> # <expression> , <expression> <expression> 1,
<expression>l

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 expression following USING must be of type string. An error occurs if the expression is numeric. If the string is a null string, a runtime error occurs. The expressions following the pound sign must follow the same rules as for unformatted printing to files.

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 list device. Numbers may be formatted with commas and decimal points; asterisks and dollar signs may be floated. Records containing embedded quotation marks or commas may also be written to a disk file with the PRINT USING statement.

For example:

```
cents.wanted% = TRUE%
editl$ = "$$##,###.##"
edit2$ = "$$##,###"
if cents.wanted% then
edit$ = edit 1 $
else edit$ = edit2$
print using "The ... &.... costs - + edit$;
#file.no%; product$, price
```

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

The -X-RAY MACHINE" costs $91,327.44crif

The use of two adjacent quotation marks in the string constant results in a single quotation mark being output to the file.

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, and then printing additional records.

An example of appending to a file is shown below:

```
true% = -I
if end # 3 then 200 rem process file not found
open "master" as 3 buff fre/128 - I recs 128
if end # 3 then 100 rem eof on process file
while true%
read # 3; dummy
wend
100 print # 3; "this added to end"
stop
200 print "file not found"
stop
```

This process may be made more efficient if the file was built with the RECL option specified. The SIZE predefined function is used to find the number of blocks in the file. The number of bytes in the file is calculated and then the number of records is determined. A random read is executed to this record and then the file is read until an end of file is detected. The following multiple line function will perform this:

```
DEF FN. GET. TO. END%(FILE. NAME$,REC. SIZE%,FILE. NUM%)
FN.GET.TO.END% = FALSE%
FILE.SIZE% = SIZE(FILE.NAME$)
IF FILE. SIZE% = 0 THEN \ REM FALSE IF NO FILE
RETURN\
ELSE FN.GET.TO.END% = TRUE%
IF END # FILE.NUM% THEN 100
READ # FILE.NUM%, (FILE.SIZE% * 1024) /REC.SIZE%;
WHILE TRUE%
READ # FILE.NUM%; DUMMY
WEND
100 RETURN
FEND
```

Except for the case of adding to the end of a file, 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 re-initialize its internal diskette usage map to accommodae the diskette being inserted. If this is not done, valid data may be overwritten.

Diskettes should never be changed while any files are open. If a file has bee written to and not closed and then an INITIALIZE statement is executed, all the new data could be lost. This means that user programs must close all active files before executing an INITIALIZE statement.

The INITIALIZE statement will re-initialize the disk usage maps for all disks inserted into logged-in drives. The general forin of the INITIALIZE statement is:

[<stmt number>] INITIALIZE

The drive selected prior to executing an INITIALIZE statement remains selected after the initialization is complete.

Insure that diskette changes are complete prior to executing the INITIALIZE statement.

Examples:

```
10 INITIALIZE
   INITIALIZE
```

The INITIALIZE statement is equivalent to the CALL 264 provided in version I of CBASIC.

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 forinat 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 as a hierarchy. The lowest level of the hierarchy is called a field. Groups of fields form records, and a fil consists of one or more records.

A field can contain either string or numeric data. A string field is surrounded by quotation marks. A numeric field is never enclosed by quotes, and it may contain any valid number as described in Chapter 3. Fields are separated from one another by either commas or a carriage return followed by a line feed.

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 organizati is used. Accessing is performed 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.

A portion of a stream file containing only string fields may look like this:

```
"first field", "second field",crlf
"third"," 126.89",crlf
11xxxl23yyy,crlf
```

There are six fields in the above example. The fourth field is a null string. The following example shows a file which contains both numeric and string data

```
"John", 798642764, "California"crl
"Tom Jones", 1234.56, "Iowa"crlf
```

CBASIC will read files in which strings are not enclosed in quotation marks. In this case, commas serve as the delimiters. Therefore, no commas may be included within the string, but a quotation mark embedded in the string would be treated as a character in the string. Strings written to files by CBASIC will always be enclosed in quotes. An attempt to write a string that contains a quotation mark to a file will result in a runtime error.

The PRINT USING statement does not insert delimiters between fields; each record will be terminated by a carriage return followed by a line feed.

11.4 Fixed Organization

Fixed organization of files provides a logical structuring of the data that pertains to a specific application.

A file is defined to be of fixed organization if the record length option is used with the CREATE, OPEN, or FILE statements. Each individual item of data in fixed files is written as a single field delimited by a comma, as with stream organization, but with the added concept of a fixed size record. A record is always delimited by a carriage return and a line feed.

One record is written each time any PRINT statement is executed. Each record always contains the number of bytes specified by the RECL parameter regardless of the number or size of the component fields. This implies that, while a given field may be any length, the combined length of all fields in the record must be less than the record length by at least two bytes to allow room for the carriage return and line feed. The last field in a record is not followed by a comma.

For example:

```
CREATE file.name$ RECL 25 AS file.no%
a$ = "one"
b$ = "record one"
```

```
c$ = -3
d$ = ....

e$ = "five"
f$ = - abc 123def"
PRINT #file.no%; a$,b$
PRINT #file.no%; c$,d$,e$
PRINT #file.no%; f
```

produces the following file:

```
"one", "record one" crlf
-3 ..... "five" crlf

I I abc 123def - crIf
```

The record delimiters, carriage return and line feed (crif), always occupy the last two bytes of the record and must be included in the specified record length. In the above example the linefeed is in the 25th position of each record. 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. For example:

```
    IF END #file.no% THEN 100
    WHILE TRUE%
        READ #file.no%; field$
        PRINT fiel
     WEND
100 STOP
```

Using the data from the previous example, the following will be printed console:

```
One
3
abcl23def
```

The fixed organization of files implies a well-defined structure to the accessed data. The processing program can 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.

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 a specific field in a record, either a zero or nul string should be written into the field. This will allow, for example, the fifth field of a sales transaction file to represent the amount of the sale, even if some or al 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 c haracters in length. By use of the STR$, function, numbers can be converted to strings and then padded, thus allowing 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
"11", -5-,-xxxl23yyy-crI
```

The required statements are:

```
OPEN file.name$ as file.no%
WHILE TRUE%
READ #file.no%; field$
PRINT fiel
WEND
```

The output on the console would be:

```
first field
second field
third

5
xxxl23yyy
```

The fourth line on the console is blank because the first field in the second record is a null string.

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 quotation marks of a string field it is permissible to have any character except a quotation mark.

When accessing a stream file, every field on the tile 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$,fiel
    PRINT fielda$, fiel
WEND
```

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

```
first field    second field
third
5       xxxl23yyy
```

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 field written by each PRINT statement wil be delimited by a carriage return and line feed instead of a comma.

For example:

```
a$= "number one
b$ = "two"
c$ = -3 -
d$= ....

e$ = "five"
f$ = "variable six"
PRINT #file.no%; a$,b$
PRINT #file.no%; c$
PRINT #file.no%; d$,e$,f
```

will put the following data in the file referenced by file.no%

```
"number one","two"crlf
-3-crlf
1411, "five", "variable six" crif
```

On files that are read or written using the stream organization, it does not matter which field delimiter is used. The crlf assumes significance when accessing files with fixed organization or when using the READ LINE statement described below.

When using the CP/M TYPE command to display a CBASIC file, the carriage return and line feed result in the output from each separate PRINT statement appearing on a separate line.

11.7 RANDOM ACCESS

In random access the program is not limited to accessing the next record or field. Any record on the file is as accessible as any other. Each record, or position where a record may be placed, is referenced by its relative record number. Each record may contain multiple fields.

Randomly accessed files must use the fixed organization. CBASIC locates each record on a randomly accessed file by taking the relative record number specified in the program, subtracting one from the number, and multiplying it by the length of a record. The result is the byte displacement of the record measured from the beginning of the file. If the records were of varying length, the displacement could not be calculated in this manner.

Normally random access files will be created sequentially and then read or updated using random access. An examle of this type of processing is an employee file for a small business. If the business has twenty employees, each would be assigned a number ranging from 1 to 20. Each employee might have a record on file with fields containing their name, social security number, and rate of pay. The twenty records would be placed on the 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% = -I
OPEN "employee. mst" RECL 50 AS 3
IF END # 3 THEN 500.1
WHILE TRUE% rem loop until eo
    INPUT "enter employee number"; employ.no%
    READ # 3, employ.no%; na   e$,ssn$,pay
    PRINT USING -&'s pay rate is ###.##"; name$, pay
```

```
    WEND
500.1 STOP
```

To summarize, the READ statement used with a stream organized file wil always access the next available field regardless of the field length or which delimiter is used. In a fixed organization file, each READ statement will access the next record. A record is 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 a file as it is when writing to the console. If the file is fixed, the single field written by each execution of the PRINT USING statement will be padded with blanks to the specified record length. 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 "SPEED=#####.### KPH"; #OUT.FILE,TIME;
    VELOCITY(TIME)

EDI$= "&
ED2$ = - $$, ###. ##
PRINT USING EDI$+ED2$+EDI$+ED2$;#17,TRANS.NO~
    "PRINCIPAL: ",PRIN, "INTEREST: -,INTR

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

PRINT USING -/2345/-;#WORK.FILE,REL. REC.NO;SORT. KEY$

IN$ = -X
WHILE IN$< >
    INPUT "ENTER DATA";LINE IN$
    PRINT USING      5 .... 0 .... 5 .... 0.3"; #4; 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 fi!e accessed with the READ LINE.

For example, if the following file exists:

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

and the following statements are 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 quotation marks and commas are considered part of the data, but the data does not include either the carriage return or the line feed.

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. There may not be a line number preceding the percent sign.

If characters on the same line following the directive are not a part of the directive, they are ignored by the compiler.

12.2 Listing Control Directives

%LIST

%NOLIST

%PAGE <constant>

%EJECT

The %LIST and %NOLIST 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. In addition, output to the disk and printer is controlled by the %LIST and %NOLIST directives.

The %PAGE directive sets the length of a page output to the printer. The constant must be an unsigned integer. If it is negative or zero, an error occurs. Initially the page length is set at 64.

As many %PAGE directives as desired may appear in a program. An error occurs if no constant is present.

The %EJECT directive positions the listings directed to the printer and the disk to the top of the next page. This is performed by outputting a formfeed character.

%INCLUDE <filename>

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

%INCLUDE b:readi

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

The %INCLUDE directive allows the programmer to build a library o common routines. This reduces programming time. System standards, such as 1/0 port assignments, can be put in included routines. If the programs are moved fro one system to another, the include routine is 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 to be changed, it can be made in one common file instead of several application programs. It is particularly valuable to code these routines as multiple line functions.

It should be noted that a program segment may compile without errors when compiled separately, but when combined with other routines, compiler errors ma occur. These errors should be predictable and will usually result from using the same line number in more than one module.

12.4 %CHAIN DIRECTIVE

%CHAIN <constant>, <constant>, <contstant>, <constant>

The %CHAIN directive is used to set the size of the main program's constant, code, data, and variable areas. This is required when chaining to insure that a program chained will not overwrite a portion of the data area being passed by the previous program. The compiler forces each of the four areas to be at least as large as the respective constant in the %CHAIN directive.

Each constant must be an unsigned positive integer. The first constant is the size of the area reserved for real constants. The second constant is the size of the code area. The third constant is the area used to store value from data statements. The final constant is the size of the area used to store variables.

The constants may be expressed as hexadecimal numbers by appending the letter H to the number. If the area to be reserved is greater than 32,767 the constant must be written as a hexadecimal number.

The values to use in the %CHAIN directive are determined by compiling each of the programs to be chained together and using the largest value of each area. The compiler lists the size of each area at the end of a compilation. For instance, if three programs are to be chained and the CODE SIZE for the programs are 789, 1578, and 4917 bytes, the second constant in the %CHAIN directive would be 4917.

The %CHAIN directive is only required in the main or first program executed. For more information refer to the discussion on the CHAIN statement.

12.5 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.

An END statement may not begin with a percent sign. It need not begin in column one, but it must be the first statement on the line.

A branch to an END statement is equivalent to executing a STOP statement.

Examples:

```
500 END
END
```

13. Operational Considerations

13.1 SYSTEM REQUIREMENTS

CBASIC operates with any CP/M based floppy disk system having at least 24K 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 48K of memory i recommended. If CBASIC is executed in a system smaller than 24K a CP/M LOAD ERROR may occur.

CBASIC will operate with CP/M version 2 and MP/M systems. A special configuration of the runtime package is available to take full advantage of the advanced features of CP/M version 2 and MP/M.

13.2 CBASIC COMPILE-TIME TOGGLES

Compiler toggles are a series of switches that can be set when compiling a program. 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. Toggles may only be set for the compiler.

Examples of the use of compiler toggles are

```
CBAS2 ACCOUNT3 $BGF

B:CBAS2 A:COMPARE $GEC
```

CBAS2 PAYROLL $B

CBAS2 BNALIDATE $E

Toggle B suppresses the listing of the program on the console during compilation.

If an error is detected, the error message is 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.

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 rovide 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'.

If toggle D is set, all keywords must be capitalized.

Initially toggle D is off.

Toggle E is useful when debugging programs. If this toggle is set, it will cause the runtime program to accompany any error messages with the CBASIC line number in which the error occurred. 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 be in effect.

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 i toggle F is set. Each page of the listing has a title printed and the pages are numbered. Formfeeds are used to advance to the top of a page.

Initially toggle F is off.

Toggle G will cause the compiled output listing to be written to a disk file. The file containing the compiled listing has the same name as the source file, and 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 produced.

Normally the disk listing will be placed on the same drive as the source file. The operator may select another drive by specifying the desired drive, enclosed in parenthesis, following the G toggle as shown below:

```
CBAS2 B:TAX $G(A:)
```

Initially toggle G is off.

13.3 Compiler Output

CBASIC does not require that each statement of a program be assigned a statement number. The only statements that must be given a statement number are those that have control passed to them by the GOTO, GOSUB, ON or I statements. During compilation, CBASIC assigns a sequential number to each line independent of the statement number which may be used by the programmer. The CBASIC assigned line number is the one referred to in error messages (i toggle E is specified) and when using the TRACE option. The line 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 seen read in from a disk file with a %INCLUDE directive. The asterisk (*) is used when the statement contains a user assigned statement 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 pfintrtn    rem rtn to print
7: 40  rem ----- rtn to print ----------
8: print name$
9: return
10: END
```

In the example, statement 3 has an asterisk because the '10' is not referenced at any place in the program. This can be useful during debugging or to help understand 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.

When an error is detected, the compiler prints a two letter error code, the line number the error occurred in and the position of the error relative to the beginning o the source line. The position assumes tab characters have been expanded.

13.4 TRACE

CRUN2 <filename> [TRACE [<Inl> [,<ln2>]

The TRACE option is used for run-time debugging. It will print the line number of each statement as it is executed. The output is directed to the console even when a LPRINTER statement is in effect. The line number printed is the number assigned to each statement by the compiler. Consider the following program:

```
AMOUNT = 12.13
TIME = 45.0
PRINT TIME * AMOUNT
```

In the above program was compiled using the following command:

```
CBAS2 TEST $E
```

and then executed with the trace option

```
CRUN2 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 (<Inl>) is used to specify the line number where the trace is to begin. The second number (<ln2>) specifies where the trace is to stop. If no line numbers are included in the command, the entire program is traced; if only the first line number is present, tracing starts at this line number and continues for all line numbers greater than the first number < I n I >.

13.5 Cross Reference Lister

In addition to CBAS2.COM and CRUN2.COM, a utility program XREFCOM is supplied with version 2 of CBASIC. XREF produces a disk file which contains an alphabetized list of all identifiers used in a CBASIC program. The usage of the identifier (function, parameter, or global) is provided, as well as a list of each line in which that identifier is used. The listing places all functions first with parameters and local variables associated with a function immediately following the function. The functions are in alphabetical order. The output is normally directed to the same disk as the source file. The file created has the same name as the CBASIC source file and is of type XRF. The standard output is 132 columns wide. The following command is used to invoke XREF

XREF <filename> [disk refl [$<toggles>] ['<tide>']

The filename must be a CBASIC source program with a filetype of BAS. The disk reference is optional and specifies the disk on which to place the cross reference file. If the disk reference is not present, the listing is placed on the same drive as the source. It is specified as A:, B: etc. For example:

XREF PAYROLL A:

will put the cross reference listing for PAYROLL.BAS on drive A. At least one blank must separate the filename and the disk reference.

Toggles may be used to alter the standard output of XREF. At least one bla must separate the dollar sign from the portion of the command line to the left. The toggles follow the dollar sign. They may be either lower or upper case letters. A, B, C, D, E, F, and G are valid toggles. Any other characters following the dollar sign, and before the title field or end of the command line, are ignored.

The A toggle causes a listing to be output to the list device as well as to a disk file.

The B toggle suppresses output to the disk. If only the B toggle is specified, no output is produced.

The C toggle suppresses the output to the disk and pen-nits output to the list device. The C toggle has the same effect as specifying both the A and B toggles.

The D toggle causes the output to be produced eighty columns wide instead o using 132 columns.

The E toggle produces output with only the identifiers and their usage. No line numbers are printed. The E toggle might be used to help document a program. The programmer would write the use of each identifier on the listing provided by XREF. Also the file created by XREF could be edited and made into a large remark with comments pertaining to each variable name. By including this file with the source program, additional documentation would be provided.

The F toggle allows the user to change the default page length of 60 lines per page. The desired number of lines per page is enclosed in parenthesis and must follow the F. There may be no imbedded blanks. Formfeed characters are used to position the printer and are also placed in disk files.

The G toggle suppresses printing of the heading lines and suppresses al formfeeds. This toggle might be used when building a disk file which will then be printed by a user utility.

The H toggle suppresses translation of lower case letters to upper case. This allows using XREF with programs compiled with compiler toggle D.

The following command: XREF GL $CD

produces a cross reference listing on the list device. The listing is 80 columns wide.

XREF ACCT$REC B: $EAF(40)

creates a disk file on drive B and a listing on the list device of all the identifiers and their usage. No line numbers would be provided. Pages are limited to 40 lines.

The optional title field must be the last field in the command line. All characters following the first apostrophe on the command line up to the second apostrophe, or until the end of the command line, become the title. The title i printed on the heading line of each page of output. The title is truncated to thirty characters if the listing is 132 columns wide and to twenty characters if the D toggle is specified.

The following command demonstrates the use of the title field:

XREF NAMES B: $AD 'version 2: 1 AUG 78

Appendix A

COMPILER ERRORS

NO SOURCE FILE: <filename>.BAS

The compiler could not locate a source file on the specified disk.This ile was used in either the CBAS 2 command or a %INCLUDE directive.

OUT OF DISK SPACE

The compiler has run out of disk space while attempting to write either the INT file or the LST file.

OUT OF DIRECTORY SPACE

The compiler has run out of directory en tries while attempting to create or extend either the INT file or the LST file.

DISK ERROR

A disk effor occurred while trying to read or write to a disk file. This message may vary slightly in form depending on the operating system being used. See your CP/M documentation for the exact meaning of this message.

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 STATE MENT(S)

There are n WHILE statements for hich a WEND could not befound.

PROGRAM CONTAINS I UNMATCHED DEF STATEMENT

A multiple line function was not terminated with a FEND statement. This may cause other errors in the program.

WARNING INVALID CHARACTER IGNORED

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

INCLUDE NESTING TOO DEEP NEAR LINE

An include statement near line n in the source program exceeds the maximu level of nesting of include files.

Other errors detected during compilation cause a 2 letter error code to be printed with the line number and position of the error. The error message normally follows the line in which the error occurred.

The possible error codes are:

BF

A branch into a m ultiple line function from outside the function was attempted.

BN

An invalid numeric constant was encountered.

CF

Common statement must be in the first line.

CI

An invalid file name was detected in a %INCLUDE directive. The file name may not contain a ?, *, or: (except as part of a disk reference where a colon may be the second character of the name).

CS

A COMMON statement, which was not the first statement in a program, was detected. Only a compiler directive such as %CHAIN, a REMARK statement, or blank lines may proceed a COMMON statement.

CV

An improper definition of a subscripted variable in a common statement. Possibly the subscript count is not a constant or there is more than one constant. Only one constant may appear in parenthesis. It specifies the number of subscripts in the array being defined.

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. Errors such as not defining functions prior to use and, in some cases, if the DIM statement does not proceed all references to an array, a DL error will result.

DP

A variable dimensioned by a DIM statement was previously defined. It either appears in another DIM statement or was used as a simple variable.

FA

A function name appears on the left side of an assignment statement but is not within that function. In other words, the only function name that may appear to the left of an equal sign is the name of the function currently being compiled.

FD

The same function name is used in a second DEF statement.

FE

A mixed mode expression exists in a FOR statement which the compiler can not correct. Probably the expression following the TO is of a different type than the index.

FI

An expression which is a subscripted 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, or the function was never 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 stnng is required.

IP

An input prompt string was not surrounded by quotes is.A subscripted variable was referenced before it wasdimensioned.

IT

An invalid compiler directive was encountered. A parameter required by the directive may be out of range or missing. Or the directive may be misspelled.

LU

A variable defined as an array in a DEF statement is used without subscripts.

MC

The same variable is defined more than once in a COMMON statement. Each variable may only appear in one COMMON statement.

MF

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

MM

An invalid mixed mode has been detected. Most likely variables of type string and type numeric are combined in the same expression.

NIS

A numeric expression was used where a string expression is required.

ND

A FEND statement was encountered without a corresponding DEF statement. This error could be the result of an improper DEF statement.

NI

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

NU

A NEXT statement occurs without an associated FOR statement.

OF

A branch out of a multiple line function from inside the function was attempted.

OO

More than 40 ON statements were used in the program. CBASIC has an arbitrary limit of 40 ON statements in a single program. Notify Digital Research if this limit causes problems.

PM

A DEF statement appeared within a multiple line function. Functions may not be nested.

RF

multiple line function may not call itself.

SD

second SAVEMEM statement was encountered. A program may have only one SAVEMEM statement.

SE

The source line contained a syntax error. This means that a statement is not properly formed or a keyword is misspelled.

SF

A SAVEMEM statement uses an expression of type numeric to specify the fil to be loaded. The expression must be a string. Possibly the quotation marks were left off a string constant.

SN

A subscripted variable contains an incorrect number of subscripts, or a variable in a DIM statement has been used previously with a different number of dimensions.

SO

The statement is too complex to compile. It should be simplified. Consider making the expression into two or more expressions. Please send Digital Research a copy of the source statement.

TO

Symbol table overflow has occurred. This means that the program is too large for the system being used. The program must be simplified or the amount of available memory increased. Smaller variable names reduces the amount o symbol table space used. Digital Research is interested in being informed i programs generate this error.

UL

line number that does not exist has been referenced.
string has been terminated by a carriage return rather than by quotes.

VO

Variable names are too long for one statement. This should not normally occur! If it does, please send a copy of the source statement to Digital Research. Reducing the length of variable names and reducing the complexity of the expression within the statement may eliminate the error.

WE

The expression immediately following a WHILE statement is not numeric.

WN

WHILE statements are nested to a depth greater than 12. BASIC has an arbitrary limit of 12 for nesting of WHILE satements.

WU

A WEND statement occurred without an associated WHILE statement.

Appendix B

RUN-TIME ERRORS

NO INTERMEDIATE FILE

A file name was not specified witbt he CRUN2 command, or no file of type INT with the specified file name was found on the disk specified.

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 values required by the input statement must be reentered.

Other errors detected 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. If an error occurs with a code consisting of an asterisk followed by a letter such as '*R' the runtime package has failed. Please notify Digital Research of the circumstance under which the error occurred.

The possible codes are listed below:

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 o the argument is returned.

NE

A negative number was specified before the raise to a power operator ( ). The absolute value of the parameter is used in the calculation. When using real variables a positive number may be raised to a negative power, but a negative number may not be raised to a power.

OF

A calculation using real variables 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.

ERROR CODES

AC

The string argument in an ASC function evaluated to a nullstring.

AE

An attempt was made to access an array element before the array DIM statement had been executed.

BN

The value following the BUFF option in an OPEN or CREATE statement is less than I or greater than 52.

CC

A chained program's code area is larger than the main program's code area. Use a %CHAIN directive in the main program to adjust the size of the code area.

CD

A chained program's data area is larger than the main program's data area. Use a %CHAIN directive in the main program to adjust the size of the data area.

CE

The file being closed could not be found in the directory. This could occur if the file name had been changed with the RENAME function.

CF

A chained program's constant area is larger than the main program's constant area. Use a %CHAIN directive in the main program to adjust the size of the constant area.

CP

A chained program's variable storage area is larger than the main program's variable storage area. Use a WHAIN directive in the main program to adjust the size of the variable storage area.

CS

A chained program reserved a different amount of memory with a SAVEMEM statement than the main program.

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 for which no IF END Statement has been executed. This may occur 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 maximu record size specified in the OPEN, CREATE or FILE statement for this file number.

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.

IF

A file name was invalid. Most likely an invalid character was found in the file name. A colon may never appear imbedded in the name proper. Question marks and asterisks may only appear in ambiguous file names. This error will also result if the file name was a null string.

IR

A record number of zero was specified.

IV

An attempt was made to execute an INT file created by a version I comp ler. To use CRUN2 a program must be recompiled using the version 2 compiler, CBAS2. This error will also result from attempting to execute an INT file which is empty.

IX

A FEND statement was encountered prior to executing a RETURN statement. All multiple line functions must exit with a RETURN statement.

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.

NC

Source program contains a real constant o utside the range of CBASIC rea numbers.

NF

The file number specified was less than I or greater than 20, or a file statement was executed when 20 files were already active.

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 but there are no DATA statements in the program, or all data items in all DATA statements have already been read.

OE

An attempt was made to OPEN a file that didn't exist and for which no I END statement had been executed prior to executing the OPEN statement.

OI

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

ONI

The program ran out of memory during execution. Space may be conserved by closing files when they are no longer needed and by setting strings to a null string when they are no longer required. Also by not using DATA statements, but rather reading the constant information from a file, space will be saved. Large arrays may be dimensioned with smaller subscripts when the array is no longer required.

QE

An attempt was made to PRINT a string containing a quotation mark to a file. Quotation marks can only be written to files when using the PRINT USING option of the PRINT statement.

RB

Random access was attempted to a file activated with the BUFF option specifying more than one buffer.

RE

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

RF

A recursive function call was attempted; recursion is not supported in CBASIC.

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.

SO

The file specified in a SAVEMEM statement could not be located on the referenced disk. The expression specifying the file name must include the type i one is present. A type of COM is not forced.

SS

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

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 or an escape char (\) was the last character 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.

Appendix C

KEY WORDS

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

TABLE OF CONTENTS