CB-80™

LANGUAGE

REFERENCE MANUAL

DIGITAL RESEARCH™

CB-80™ Language

Reference Manual

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The "CB-80 Language Reference Manual" was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press/Monterey.

****
* Second Printing: March 1982 *
****

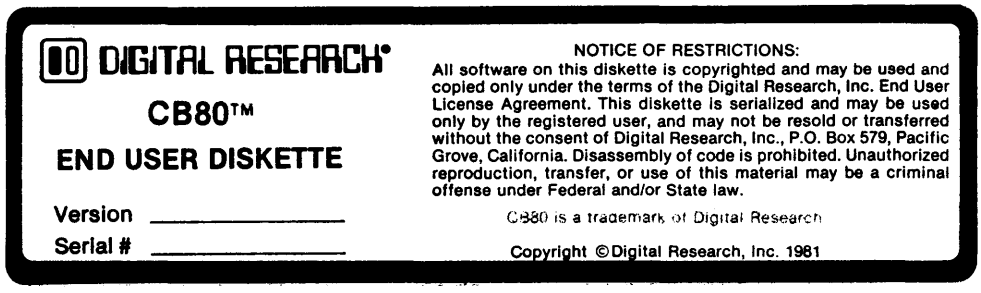
Foreword

CB-80™ increases the performance of commercial applications software on Z80® and 8080 based microcomputer systems. CB-80 consists of the CB-80 compiler, the CB-80 library and a link editor, LK-80™. With CB-80, you can compile statements into a module consisting of relocatable machine instructions. You can link a number of separately compiled modules into a single relocatable module with LK-80. A relocatable module linked with the CB-80 library by LK-80 produces an executable program. You can also generate overlays to allow one program to chain to another. These features ensure that very large programs can be compiled and that performance does not degrade when applications become large. Additional CB-80 features support operation in a multi-user enviornment.

CB-80 maintains compatibility with CBASIC®. This allows you to take existing applications and convert them to CB-80. The result is much faster execution and more flexibility when using assembly language.

This manual is a reference guide to CB-80. It defines the structure, functions, and statements of the CB-80 language and then describes the operation of the compiler and the linker. This manual does not teach programming principles and in general, assumes that you are familiar with programming in one or more high level languages. If you are familiar with CBASIC scan this manual for new statements that have been added to CB-80. Read Sections 4.5, 4.6, 7.11, 11, and 12 in detail.

CB-80 and associated programs are distributed by Digital Research or by dealers licensed by Digital Research to distribute CB-80. A diskette containing an authorized copy of CB-80 has a label like the one shown below.



A copy of Digital Research's CB-80 Licensing Guide accompanies each copy of CB-80. If you do not have a Licensing Guide or if your disk does not have an end user label, please contact Digital Research at (408)649-3896.

Digital Research is very interested in your comments on our documentation and programs. Problem report forms are included with your distribution disk. Please use them to help us provide you with a better product.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Table of Contents
(continued)

Table of Contents
(continued)

Appendixes

Section 1
Introduction to CB-80

A CB-80 source program is a text file consisting of ASCII characters. Groups of characters form one of the following program primitives: identifiers, reserved words, constants, special characters, or remarks. Section 1.1 defines the CB-80 character set that forms these primitives. The following sections describe each program primitive. Section 1.5 explains the notation this manual uses to illustrate each statement in the language.

1.1 CB-80 Character Set

Any ASCII character can appear in a CB-80 program. The CB-80 language uses the alphanumeric characters and the following special characters:

```
! " # $ % & ( ) = - ^ \ * : + ; ? / > . < ,
```

You can insert any number of blanks between program primitives. Except in a string constant (discussed in Section 1.3), a consecutive group of blanks is treated as one blank. For example the following primitives are the same.

```
PRINT X
```

and

```
PRINT                    X
```

A physical line of source code is terminated by the end of line character which is a carriage return followed by a line-feed.

You can use tab characters in source programs; CB-80 treats them as blank characters. In listings, CB-80 expands tabs to the next column that is a multiple of eight.

Except in string constants, CB-80 converts lower-case alphabetic characters to the corresponding upper-case alphabetic characters. The following primitives are the same:

```
PRINT
```

and

```
print
```

The backslash character (\) has a special meaning in CB-80. Unless it is contained in a string constant, explained in Section 1.3, the backslash signifies that the next line is a continuation of

the current line. This allows a line oriented language like Basic to have statements extend over many physical lines. CB-80 ignores any characters following the backslash on the same physical line. (Section 2 details the use of continuation characters.)

1.2 Identifiers and Reserved Words

An identifier is a program primitive that is a group of alphanumeric characters and decimal points. Identifiers represent program elements, defined by the programmer, such as variables, function names and labels. Subsequent sections explain the use of these elements. In general, the same identifier cannot be used for two different elements. Reserved words are identifiers that have specific meaning in the CB-80 language. In the remainder of this manual, the term identifier refers to identifiers other than reserved words or compiler directives. Appendix A contains a list of CB-80 reserved words.

The first character in an identifier must be alphabetic, a question mark (?), or a percent sign (%). CB-80 permits a question mark as the first letter of an identifier to allow access to a routine in the CB-80 run-time library. Identifiers that start with a percent sign are called compiler directives. (Section 2 explains these directives. Appendix A contains a list of all compiler directives.) Identifiers can end with a percent sign or a dollar sign.

CB-80 always converts lower-case letters in an identifier to the corresponding upper-case letters.

Identifiers can be of any length. CB-80 allows long identifiers so that you can choose names that have meaning. This makes programs easier to develop and maintain. The amount of space CB-80 requires during compilation of a program is related to the length of identifier names. If you have long identifiers, you might need more memory space to compile your program. However, the size of the identifiers does not affect the amount of executable code CB-80 produces.

A specific implementation of CB-80 might limit the number of significant characters in an identifier. CB-80 does not set this limit to less than 31 characters. Linkage editors might truncate names of functions that are public or external to less than 31 characters. If an identifier is truncated due to an implementation length restriction, a terminating dollar sign or percent sign is not retained. This could change the meaning of the identifier. (See Appendix E for current implementation limits.)

You can imbed decimal points in an identifier to enhance readability. You can use any number of decimal points, and they can appear as the last character of the identifier. For example:

```
MASTER.ACCOUNT.NUMBER.
```

```
FILE.NUMBER%
```

The decimal point is part of an identifier and must be present in all references to that identifier. Therefore, the identifiers:

```
INV.NO
```

and

```
INVNO
```

are different identifiers.

The following list shows valid identifiers:

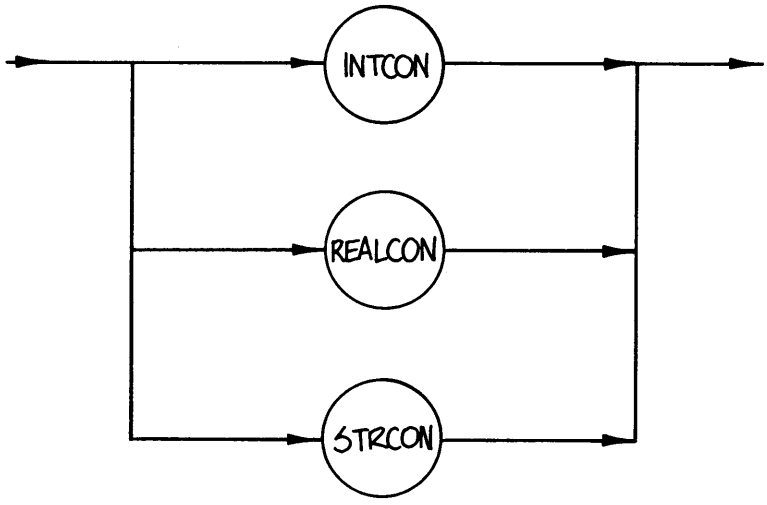
```
AMOUNT                          FN.ANGLE
PAYMENT.DUE.DATE                ORDER.QTY
INDEX%                          I%
DaTe$                           ACCOUNT.101
income.source.code              A1.B2.C3.
?GETS                           I
```

The following identifiers are invalid:

A$B	CB-80 only allows a dollar sign at the end of the identifier.
SIN	SIN is a reserved word (see Appendix A).
7IJ	Valid identifiers must start with a letter (I7JK is valid).
A?B	Question marks can only appear at the beginning of an identifier.
$	Valid identifiers must start with a letter or ?.
.A.B	Decimal points cannot start an identifier.
A B C	Spaces cannot appear in identifiers.

1.3 Constants

A constant is a program primitive that does not vary during the execution of a program. There are two types of constants: string and numeric.



A string constant is a group of characters enclosed within quotation marks. The maximum number of characters allowed in a string constant is implementation dependent (see Appendix E for current limits). In all cases, CB-80 permits at least 255 characters.

The compiler treats two consecutive quotation marks within the string as one quotation mark which is a character in the string. For example:

```
"He said ""The time has come"" before he left"
```

represents the following string constant:

```
He said "The time has come" before he left
```

The following examples also use imbedded quotation marks:

```
""""
"this is a quotation mark """
```

The first example is a string consisting of one quotation mark. The string constant:

```
""
```

is a null string. A null string is a string with a length of zero.

The following are examples of valid string constants:

```
"This is a valid string constant"

"ABC Development Company"

""

"PAYMENT DUE DATE: "

"..!#$%&'()=~^|\{[}]*:+;<..."
```

Numeric constants are either integer or real constants. Integer constants are stored as two byte signed binary integers with a maximum magnitude of 32767. Real constants are stored as eight byte binary coded decimal digits. The first byte is the sign and exponent; the remaining seven bytes represent the mantissa. You can express real constants in either decimal or floating point format. The compiler converts numeric constants to an internal format.

The following are examples of valid numeric constants:

1	0	32767
5478	12345	21
12.83	1267.	54.0E 01
1.11E-21	0.01E63	1.23E+61

A blank can appear following the E in a numeric constant. No other blanks can be used in a constant.

Numeric constants are always positive. If you append a sign to a constant, CB-80 treats the sign as an unary arithmetic operator. (See Section 5 for a discussion of arithmetic operators.)

The following numeric constants are invalid:

3.2E	Missing exponent.
1.23E+99	Exponent out of range.
12,734	Commas are not permitted within constants.
0.11.2	Only one decimal point is permitted.
12 .34	A blank is not permitted in the number.

If a numeric constant does not contain a decimal point or an exponent, the compiler treats the constant as an integer unless the magnitude of the constant exceeds 32767, the maximum magnitude of CB-80 integers. If the constant exceeds 32767, CB-80 treats the constant as a real constant. In other words, 30000 is an integer but 300000 is a real constant. 30000.0 is also a real constant because it contains a decimal point.

Integer constants can also be expressed as hexadecimal or binary constants. A binary constant is a group of 0's and 1's ending in the letter B. Hexadecimal constants consist of a group of numeric characters and the letters A through F. A hexadecimal constant ends with the letter H.

In binary constants the letter B, and in hexadecimal constants the letters A through F and the letter H, can be either lower-case or upper-case. The first character of a hexadecimal constant must be a digit.

The following list contains examples of valid binary and hexadecimal constants:

1100b	0101010101B	8000h
7ABCH	7abch	1B
07fffh	0000H	0FfFfH
0h	111b	0ABCDH

Unlike decimal integer constants, binary and hexadecimal constants are not converted to real constants if their magnitude exceeds 32767. This allows bit patterns up to 16-bits long to be represented as constants. This means that while CB-80 treats

```
65535
```

as a real constant,

```
0FFFFH
```

is a hexadecimal integer constant.

The following binary and hexadecimal constants are invalid:

fa3eh	Does not start with a digit.
7ABCD	Missing H at end of constant.
0FFFFFH	Exceeds the range of integers.
010201b	Binary contains digit other than 0 or 1.
0 111 0B	Spaces not permitted in constants.
1011,1111	Comma not permitted in constants.

1.4 Remarks

You can add remarks to a source program to increase the readability of your program, but the compiler ignores remarks. A remark starts with the reserved word REMARK or REM, and terminates with the physical end of the line or with a backslash if the remark continues to the next line.

Remarks can appear anywhere in the source program with the following restrictions.

- A remark always terminates a statement. (Statements are described in subsequent sections).
- Remarks cannot be imbedded in other program primitives.
- Remarks are not permitted as part of a DATA statement. (Section 3 explains DATA statements).

CB-80 treats a blank line as a remark. You can use any number of blank lines within a program. In the example below, the blank line becomes part of the remark, but the third line is not a continuation of the remark.

```
REM   THIS IS A REMARK CONTINUED \

      BUT THIS IS NOT PART OF THE REMARK
```

The following examples show valid remarks.

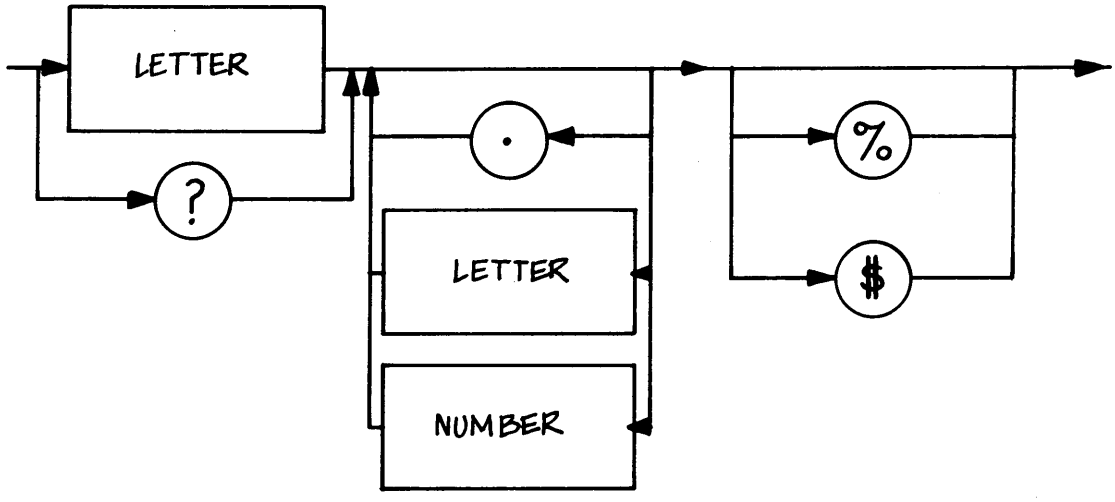
```
REM Any Characters

REMARK ACCOUNTS PAYABLE
```

```
REMARK \
       \PAYROLL
       \PROGRAMMED BY TIM SMITH
       \LAST MODIFIED 28 JUNE 1981
         VERSION 1.03
```

In the last example, the backslashes indicate that the five physical lines are one remark. Thus, the backslash has specific meaning even as part of a remark. A remark cannot contain a carriage return because a carriage return terminates the remark. The carriage return is not part of the remark.

1.5 Notation

This manual uses syntax diagrams to illustrate the syntax of each statement in the language. A syntax diagram shows the permissible constructs for each statement. For example, the syntax diagram for an identifier is:



- A rectangular box indicates a program element that is further defined by another syntax diagram. In this example, a syntax diagram could be drawn to show that a letter is an A, B, C etc.
- The circle indicates a reserved symbol or token in the language.
- Arrows represent the flow of control that indicates permissible alternative forms of the program element.
- For clarity, program examples in this manual use upper-case letters for both reserved words and identifiers. However, you can also write any of the identifiers or reserved words in lower-case without altering the program.

Note: the CB-80 Reference Manual is independent of the operating environment wherever possible. However, when filenames must be shown, CP/M® filenames are used. Digital Research's CP/M and its derivatives MP/M™ and CP/NET™ are the standard operating systems for 8-bit microprocessors using CB-80. If you are using CB-80 with an operating system other than CP/M, file specifications might differ from those shown in this manual.

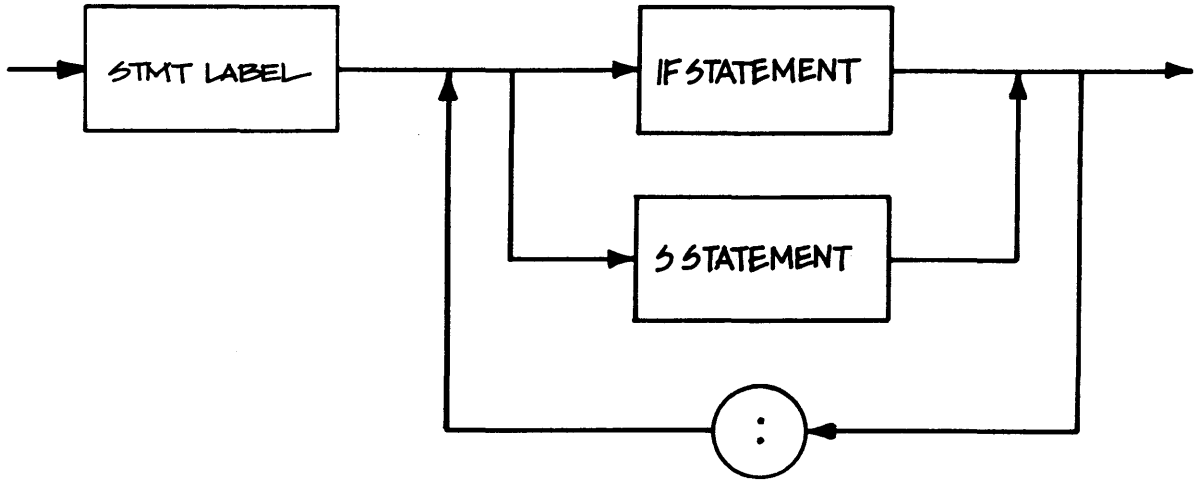
End of Section

Section 2
CB-80 Program Structure

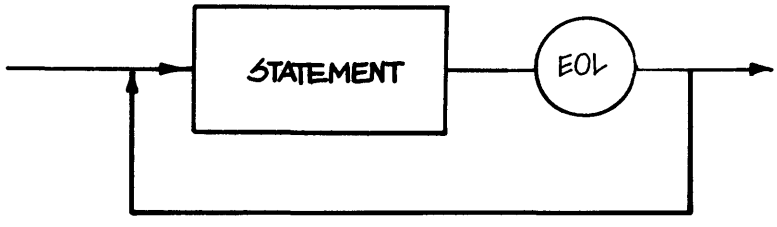
Section 1 defines program primitives from which CB-80 programs are built. This section describes the overall structure of CB-80 programs, which consist of a declaration group followed by a statement group. Section 2.2 describes compiler directives that provide information to the compiler during compilation.

2.1 CB-80 Statements

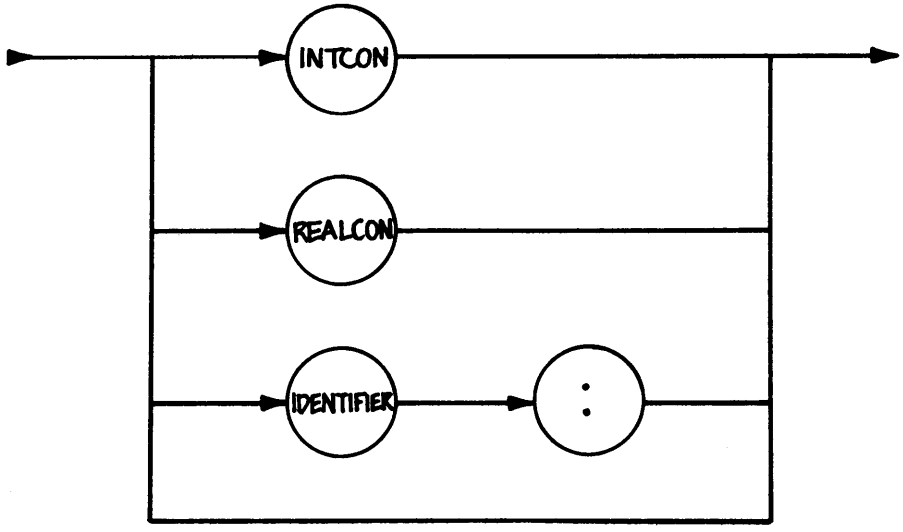
A CB-80 statement consists of an optional statement label and one or more statements separated by colons. The statement terminates with the end of a physical source line. With the exception of some assignment statements, all statements start with a reserved word.



A number of statements is called a statement group.



A statement label can be an integer or real constant or an identifier with a colon appended to the end of the identifier.



When you use an identifier for a label, you cannot use it again in another context within the program. This means it cannot be used as a variable or a function name. (See Section 4 for a discussion of local variables and labels within multiple line functions for an exception to this rule.)

The following list contains valid statement labels:

```
100                 2300.00             2222
GETRECORD:          PROCESS.COMMAND:    A:
200E03              100.00              0.001
```

The following statement labels are invalid:

```
100H      Hexadecimal constants are not permitted.
XYZ       Colon is missing.
1#2       Invalid constant;  pound sign is not permitted.
stop:     Stop is a reserved word.
```

When an alphanumeric label is referenced, the colon is not part of the label. (Section 7 explains statements that reference labels.)

When you use a numeric constant as a label, the characters making up the label determine the uniqueness of the label, not the value of the label itself. The labels 100.0 and 100.00 are different labels even though they have the same numeric value.

Section 1 explains that CB-80 uses the backslash character (\) as a continuation character to allow statements to extend over many physical source statement lines. For example:

```
PRINT X, Y, Z
```

can be written as:

```
PRINT    \
      X, \
      Y, \
      Z
```

A continuation character causes CB-80 to ignore all characters beginning with the continuation character and including the first end of line.

```
PRINT \ ALL THIS IS IGNORED
      X
```

A continuation character can appear anywhere that a blank can separate program primitives. Thus, the continuation character can separate two primitives:

```
PRINT \
      X
```

A continuation character cannot split a primitive. The following example shows an invalid use of the continuation character:

```
PRI\
NT X
```

CB-80 treats a backslash within a string constant as a character within the string rather than as a continuation character. For example:

```
"AB\CD"
```

is a valid string constant that contains 5 characters.

Because CB-80 ignores all characters following the continuation character on the same physical line, the characters following a continuation character can be used to document a program.

```
PRINT \   NOW PRINT THE TOTAL
    ACCOUNT.TOTAL
```

A remark terminates a statement. Thus, the statement:

```
PRINT REMARK NOW PRINT THE TOTAL
    ACCOUNT.TOTAL
```

is not the same statement, and is in fact an incorrect CB-80 statement.

At times it is necessary to form a group of statements. Normally, this is used in conjunction with the IF statement described in Section 7.2.

The special character colon (:) indicates that two consecutive statements are part of a statement group. For example:

```
PRINT X : PRINT Y
```

To prevent confusion with a label, the colon must not be adjacent to an identifier.

All statements in a group must be part of one logical statement line. This means that if the statement group is spread over multiple source lines, you must use the continuation character. For example:

```
PRINT X :\
PRINT Y
```

associates both statements in the same group. But

```
PRINT X :
PRINT Y
```

does not. In this last example, the first line is a group of two statements consisting of a print statement followed by a null statement. The second line is another print statement not part of the statement group in the first line.

A colon allows multiple statements on one line. In conjunction with the continuation character, the colon allows groups or blocks of statements to be continued over many physical source lines.

2.2 Compiler Directives

Compiler directives are reserved words that provide information to the compiler; they are not translated into executable code. The following sections define the different compiler directives.

All compiler directives begin with a percent sign (%). For example:

```
%LIST
```

There must not be any blanks between the percent sign and the remainder of the directive. The compiler directive can appear anywhere within a source line but no other statements can appear on the same line with the directive. CB-80 ignores any characters on the same line with the directive unless they are required by the directive.

Only blanks or tab characters can precede the directive; it cannot have a label.

A compiler directive cannot be continued to another line with a continuation character.

2.2.1 Listing Control Directives

There are four compiler directives that affect the format of the listing product by CB-80: %LIST, %NOLIST, %EJECT, and the %PAGE directives. Compiler toggles, explained in Section 11, also affect listings.

The %NOLIST directive stops listing the source file and interlisting code. The %LIST resumes listing the source file.

```
%LIST

%NOLIST
```

The %EJECT directive continues the listing on the top of the next page. The %EJECT directive is only in effect if the listing is being directed to the printer. The %EJECT directive is ignored if %NOLIST is in effect.

```
%EJECT
```

The %PAGE directive sets the page length of a listing directed to the printer. The desired length must be an integer constant following the %PAGE. The following example sets the page length to 40 lines:

```
%PAGE 40
```

2.2.2 %INCLUDE Directive

The %INCLUDE directive allows source code in a disk file to be included into the source program during compilation. The following directive includes source statements from the file CONDEF.BAS.

```
%INCLUDE CONDEF
```

The file CONDEF.BAS is read from the CP/M default drive. The filetype of the file specification defaults to "BAS". However, you can specify any filetype. For example, the following directive includes the file CONDEF.INC into the source program.

```
%INCLUDE CONDEF.INC
```

You can specify that an include file be read from a drive other than the default drive. One method is to directly specify the drive, as shown below.

```
%INCLUDE D:CONDEF.INC
```

Another method uses a compiler toggle to read include files from a drive other than the one containing the source program. (Section 11.2 explains compiler toggles.)

Include files can be nested. The maximum depth of such nesting is implementation dependent. (See Appendix E for the current limitations.) You can assume that the maximum allowable depth is always at least four, however some operating environments limit the number of files that can be open at one time. Extensive use of %INCLUDE files, especially when nested, decreases the speed of compilation.

The included text is incorporated into the source directly after the %INCLUDE directive. CB-80 treats the first character of the included text as the next character in the source program. The physical line containing the %INCLUDE directive is not a part of the statement being compiled.

A %INCLUDE directive can "split" a statement.

```
PRINT \
     %INCLUDE RECDEF.INC
```

If file RECDEF.INC contains the following source line:

```
NAME$ \
ADDRESS$
```

the %INCLUDE forms with the following statement:

```
PRINT \
      NAME$ \
      ADDRESS$
```

The statement that is actually compiled is obtained by replacing the entire source line containing the %INCLUDE directive with all source lines in the file specified in the directive.

End of Section

Section 3
Data Types and Declarations

CB-80 provides a variety of data types to support the requirements of programmers implementing commercial applications. There are three kinds of CB-80 data: numeric, string, and label. A specific data type can be either a constant or a variable. Constants do not change value during execution of a program, while variables can assume different values during program execution. This section explains the properties of CB-80 data items.

3.1 Numeric Data

Numeric data represents arithmetic and logical quantities. Numeric data falls into two classes: integer and real. Integer quantities are represented as two's complement binary numbers. Each integer requires two bytes for storage. If you assign an integer a value outside the defined range of 15 binary digits (-32768 to 32767), the results are undefined.

Integer data is processed more efficiently than real data because the hardware processes integers directly. In addition, integers use less memory than real data. You should use integers whenever possible to decrease execution time and to reduce the amount of memory used.

The compiler stores real numeric data as packed decimal digits in an eight byte floating point format. The first byte contains both the exponent and the sign of the number. The first bit is the sign of the number. The remaining 7 bits are the exponent.

The mantissa is seven bytes long and contains 14 digits. Values are always stored in a normalized format as 4 bit decimal digits. There are two digits stored in each byte of the mantissa.

The dynamic range of real numbers is 1.0E-64 to 9.99999999999999E+62. Both the accuracy and dynamic range of CB-80 numbers are significantly greater than that found in most binary implementations of real numbers.

The internal representation CB-80 uses for some real numbers is shown below:

NUMBER	EXPONENT	MANTISSA
1.0	41H	00H 00H 00H 00H 00H 00H 10H
-1.0	C1H	00H 00H 00H 00H 00H 00H 10H
0.0123	3FH	00H 00H 00H 00H 00H 23H 10H
0.0	00H	(not significant if exponent byte 0)
largest positive number	7FH	99H 99H 99H 99H 99H 99H 99H
smallest nonzero positive number	01H	00H 00H 00H 00H 00H 00H 10H

3.2 String Data

String data consists of variable length strings of characters. A string can have a maximum length of 32767 bytes. Space for string variables is allocated dynamically and released when the string is no longer required.

The first two bytes of a string represent the length of the string. The first byte of the length is the high-order byte and the second byte is the low-order byte. This is contrary to the normal storage of sixteen bit quantities in 8080 microprocessors. The string "SAMPLE" is stored internally in eight bytes:

LENGTH	BODY OF STRING
00H 06H	53H 41H 4DH 50H 4CH 45H

The system uses the left most bit (bit 7) of the first byte of the length to recover temporary strings. This bit must be ignored when accessing the string length. Thus, the string length is actually the low-order 15 bits of the first two bytes of a string.

3.3 Label Data

Labels reference statements and functions and are always constants. Section 3 explains statement labels; Section 4 explains functions.

Labels within the main executable block of a program must be unique. All labels within a function (Section 4) must also be unique, but a label within a function can be the same as a label in another function or the main executable block.

3.4 Data Structures

CB-80 supports two data structures: simple variables and arrays. Simple variables are single values associated with a variable name. Simple variables can be of three types: integer, real, or string. For example, the following identifiers represent simple variables:

```
AMOUNT          PAYMENT.DUE.DATE$       FIRST.FLAG%

INDEX%          ANGLE                   I
```

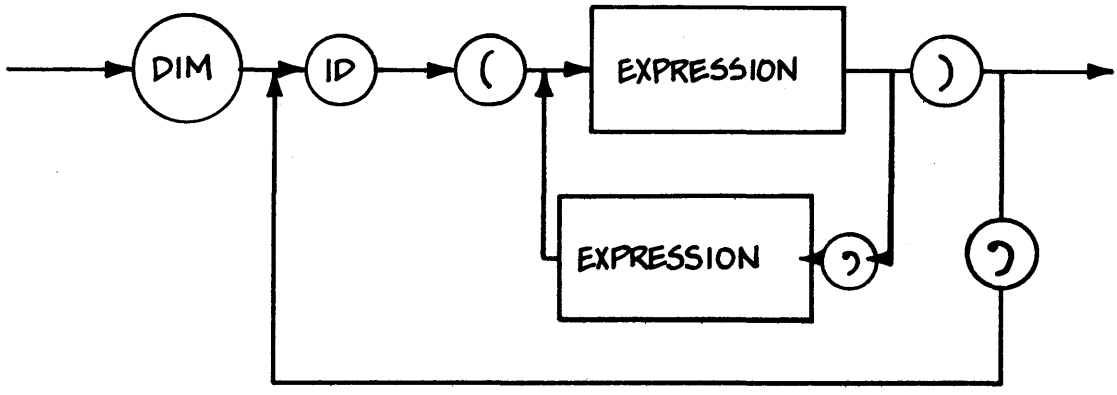
Integers are stored in two bytes of memory; real variables require eight bytes of storage. Strings are assigned two bytes of permanent storage that store the address of the dynamically allocated string.

Arrays are the other data structure CB-80 provides. An array associates a group of simple variables to one variable name. A particular element is identified by providing subscripts to select one variable in the array. In the following example, MATRIX is the array name. The values in parentheses are subscripts selecting a specific element of MATRIX. MATRIX is a two dimensional array because there are two subscripts.

```
MATRIX(2,3)
```

Arrays can have any number of dimensions, and the value of dimensions can be expressions determined during the execution of the program. A particular implementation of CB-80 might limit the number of dimensions allowed in an array. (Refer to Appendix E for current limitations.)

The DIM statement dynamically allocates space for an array. That is, the memory the array requires is not reserved until the DIM statement is executed.



The expressions specify the upper bound for each subscript. Section 5 defines expressions. The lower bound is always zero. For example:

```
DIM X(25)
```

allocates an array with 26 elements, X(0), X(1), through X(25). The statement:

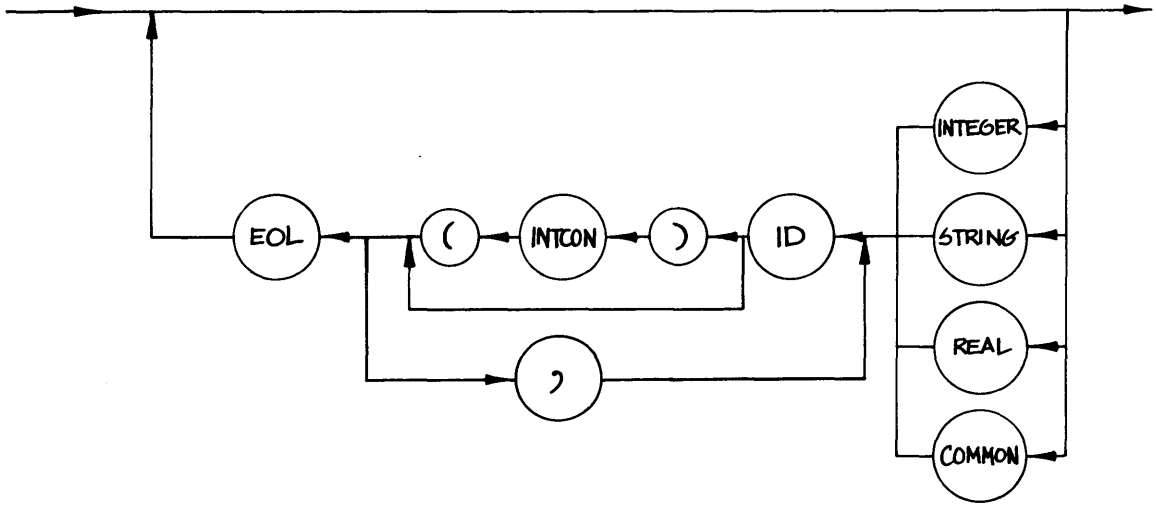
```
DIM ACCOUNTS(I,J)
```

creates space for (I+1) * (J + 1) elements.

The actual method of allocation is undefined in CB-80. CB-80 does not define the order in which elements are stored in memory for a specific array. The method of allocation can vary from implementation to implementation. This approach allows allocation methods that give efficient access to array elements on machines without hardware multiply.

3.5 Declarations

Declarations allow you to specify whether a specific variable or function name represents an integer, real, or string data type. Declarations also indicate that a variable is in COMMON.



The following statements are valid declarations:

```
INTEGER I,J,LOOP.COUNT
REAL A, AMOUNT.DUE, C
STRING NAME,PART.DISCRIP
```

In the statements above, the identifiers I, J, and LOOP.COUNT represent integer data items and the identifiers A, AMOUNT.DUE, and C represent real data items. NAME and PART.DISCRIP are strings.

If the identifier represents an array, the number of subscripts are in parentheses following the identifier name.

```
INTEGER MAX(2), Y(1)
STRING NAMES$(1)
```

The statement above declares MAX to be a two dimensioned integer array, while Y and NAMES$ each have one dimension. This declaration does not result in allocation of space for the array. You must execute a DIM statement for an array prior to referencing any elements in the array.

Any statement in a declaration block can have a label. CB-80 ignores the label except that it is assigned the address of the first executable statement in the statement group that follows.

The following declarations are invalid:

`INTEGER I,J K`	Missing comma.
`REAL X(15,40)`	Arrays have number of dimensions in parentheses.
`STRING POS`	POS is a reserved word.
`REAL X : INTEGER I`	Colon cannot be used to group declarations.

In addition to the INTEGER, REAL, and STRING statements, a declaration group can contain blank lines, REM statements, COMMON statements, and DATA statements. For example:

```
        INTEGER FLAG1, FLAG2 REM FLAGS FOR FILE I/O

100     REMARK FOLLOWING VARIABLES USED FOR CALCULATIONS

        REAL AMOUNT, BALANCE, PAYMENT
```

You can place any program variable in COMMON. This allows data to be shared by two or more programs. (See Section 7 for a discussion of CHAINING.) The following COMMON statement places three variables in COMMON:

```
COMMON X, Y, Z
```

When a variable is subscripted, then the number of subscripts is placed in parentheses following the variable name. For example:

```
COMMON A(2)
```

specifies that the variable A is a two dimensioned array. The statement order of variables placed in COMMON statements must be the same in all chained programs using these variables.

The same variable can appear in a declaration statement and a COMMON statement. For example:

```
STRING X
COMMON X, Y(1)
REAL   Y(1)
```

You can place any number of COMMON statements in a declaration block. However, if a declaration block is used in a multiple line function (Section 4), no COMMON statements are permitted.

3.6 Default Declarations

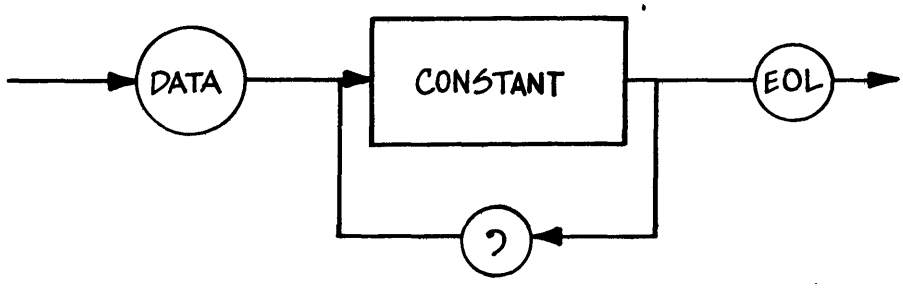
CB-80 provides default declarations for variables that do not appear in an INTEGER, REAL or STRING declaration statement. Variable names that end with a percent sign (%) default to integer variables, while variables ending in a dollar sign ($) default to string variables. Other variables default to real variables.

For example, CB-80 treats the variable X as a real variable, while A$ is a string. INTEGER, REAL, or STRING statements can override the default declarations. The following statement declares A$ to be an integer.

```
INTEGER A$
```

3.7 DATA Statements

A DATA statement is not executable but defines a list of constants that can be assigned to variables using a READ statement. (READ statements are explained in Section 8.) Any number of DATA statements can occur anywhere in a program, either in the declaration group or in an executable group. However, CB-80 treats all DATA statements, whether they occur as connective statements or not, as one list of constants available during execution.



The following examples show valid DATA statements:

```
        DATA 1,2,3,4

100     DATA "APPLE", GRAPE, "ORANGE"

        DATA "$$$$$", "#####", """""""""""", \
             "!!!!!", "\\\\\"
```

In the last example, the continuation character continues a DATA statement to another line. However, backslashes can appear in string constants enclosed in quotation marks.

Strings do not have to be enclosed in quotation marks, in which case they are optionally delimited by commas. A field must be terminated with a comma or the end of line character.

The following DATA statements are invalid:

`DATA 12, ,13`	Missing field.
`DATA "ABC`	Missing quotation mark.
`DATA 1,2 REM VALUES`	A REMARK not allowed here.
`DATA "AB" "CD"`	Comma missing between strings.

DATA statements cannot appear in lines containing other statements. A DATA statement cannot be part of a statement group.

Labels are optional on DATA statements. Because a DATA statement is not executable but rather defines a list of constants that are available during execution, the label actually addresses the first executable statement following the DATA statement. Thus the following example:

```
START.EXEC: DATA 10,20,30
            PRINT X
```

is equivalent to:

```
            DATA 10, 20, 30
START.EXEC: PRINT X
```

3.8 Identifier Usage

Unless its scope is different, you cannot use an identifier for two different elements even if the usage is not ambiguous. An identifier used as a function name or as a label cannot be used as a variable. In addition, the same identifier cannot be used as both a subscripted and non-subscripted variable.

The following example is invalid:

```
ACCOUNT: ACCOUNT = 3
```

The identifier ACCOUNT cannot be used as both a label and a simple variable. The next example is also invalid:

```
X = X + X(3)
```

The identifier X cannot be used as both a subscripted and simple variable name.

Section 4 discusses the scope of variable names. It is possible for the same identifier to have two different uses when the scope of the identifiers is different.

End of Section

Section 4
User Defined Functions

A function allows you to execute the same group of statements from various points in a program. Functions can be included in the program that references them, or they can be in separate modules. If the functions are in separate modules, each module is compiled and the modules are then linked together. CB-80 provides two types of functions: user defined functions and predefined functions. This section describes user defined functions. Section 6 describes predefined functions.

4.1 Introduction to User Defined Functions

Functions perform operations that have limited and controlled interaction with the remainder of the program. CB-80 supports two types of user-defined functions: single line and multiple line functions.

Both types of functions can have zero or more formal parameters. A function contains a list of the formal parameters that are assigned a value when the function is accessed. An actual parameter is an expression that is passed to the function when the function is referenced, and substitutes for a formal parameter. (See Section 5 for a discussion of expressions.)

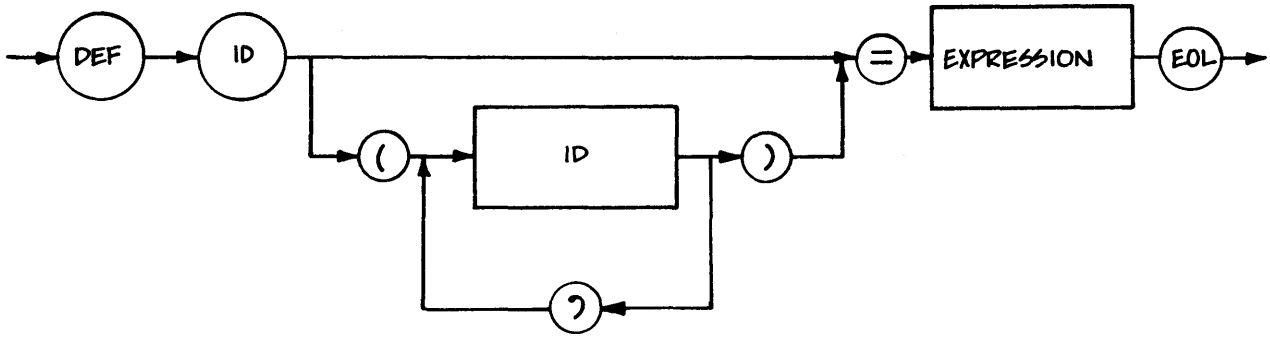
When a function is accessed, the number of formal and actual parameters must agree. In addition, if the formal parameter is a string, then the actual parameter must evaluate to a string expression; if the formal parameter is numeric, the actual parameter must be numeric. An integer expression can be passed to a real formal parameter, and an integer formal parameter can accept a real actual parameter. The appropriate conversion occurs. The implementation can limit the maximum number of parameters allowed in a function. (See Appendix E for current implemention limits.)

All parameters in CB-80 are passed by value. This means that the actual parameter is evaluated before the function is executed. The value of the actual parameter is then passed to the function and becomes the initial value for the corresponding formal parameter. This method of passing parameters assures that changing a value of a formal parameter does not change the value of a variable outside the function.

Both single line and multiple line functions can be elements in an expression; a multiple line function can also be invoked through a CALL statement. (CALL statements are explained in Section 7.6.)

4.2 Single Line Functions

Single line functions evaluate an expression and return the value of the expression. A single line function is similar to Fortran's statement function.



The ID following the reserved word DEF is the function name. The expression following the equal sign can be any valid expression. If the expression is of type string, the function name must be of type string. (Section 5 explains expressions.)

You access a single line function by using its name in an expression. The following function calculates the average of two integers:

```
DEF AVERAGE%(A%,B%) = (A% + B%)/2
```

A% and B% are formal parameters. When you reference a function, actual parameters are substituted for the formal parameters and then the expression is evaluated.

The following statement uses the single line function AVERAGE% to determine the average of two expressions.

```
PRINT AVERAGE%(TEST.1% + 2,TEST.2%)
```

TEST.1% + 2 and TEST.2% are the actual parameters substituted for A% and B%.

The identifier used as a function name defines the type of value returned. The function AVERAGE% defined above returns an integer.

```
DEF CONVERT(A%) = A%
```

This function returns a real value since CONVERT is a real identifier.

```
DEF CAT$(A$,B$) = A$ + B$
```

The function CAT$ returns a string.

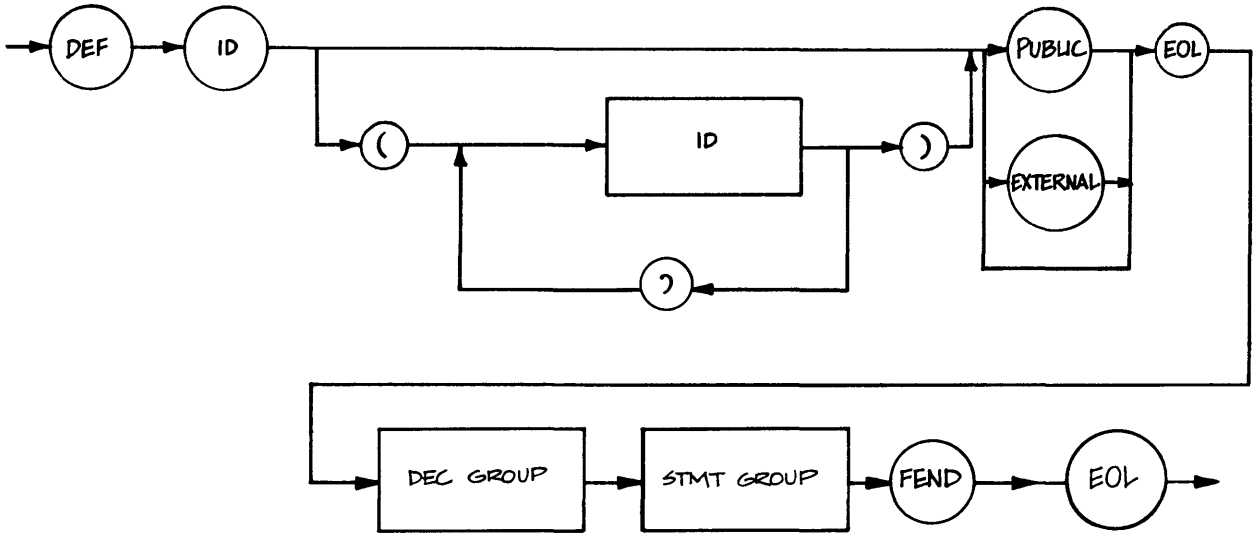
You cannot place the names of single line functions in a declaration. For example, the following statements are not correct:

```
STRING CAT
DEF CAT(A$,B$) = A$ + B$
```

4.3 Multiple Line Functions

Multiple line functions consist of a function definition followed by a declaration block and an executable block. The FEND statement indicates the end of a multiple line function.

Multiple line functions are equivalent to Fortran subroutines and functions, or PL/I procedures.



Section 4.5 explains EXTERNAL and PUBLIC functions. They permit linkage with separately compiled modules.

```
DEF FN.NAME(F,M,L)
     STRING F,M,L,FN.NAME

     FN.NAME = F + " " + M + " " + L
FEND
DEF MEAN(X,Y)

     MEAN = (X + Y)/2.0
FEND
```

The declaration group cannot contain a COMMON statement. Array variables can be declared but each execution of the DIM statement results in a new array being dimensioned. Array names cannot be passed as parameters; individual array elements can be used as actual parameters.

The executable block can contain any CB-80 executable statements. However, function definitions cannot be nested. A multiple line function cannot contain another multiple line or a single line function definition. In addition, recursive references are not supported.

Multiple line functions are invoked either with a CALL statement explained in Section 7, or by using the function as an element in an expression. If the function is used as part of an expression, the function returns a value. The type of the value returned is the same as the type of the function name.

```
DEF A%
      .....
FEND

PRINT A%
```

The function A% returns an integer value. This value is the last value you assign to the function name prior to returning from the function. A function returns when the reserved word FEND is reached or when a RETURN statement is executed. (Section 7.7 explains RETURN statements.)

```
DEF GREATER(A,B)
     STRING GREATER, A, B

     IF A > B THEN \
          GREATER = A \
     ELSE \
          GREATER = B
     RETURN
FEND
```

The function GREATER returns a string that is equal to the greater of the two parameters. The function GREATER can also be called, with no value being returned. But in this example, it is of little practical value.

```
CALL GREATER
```

A RETURN statement in a function results in a return from the most recently executed GOSUB or function reference. (See Section 7 for a discussion of the GOSUB and RETURN statements.)

4.4 Scope of Variables

All formal parameters and any variables you declare in the declaration block are local to the function. In addition, labels defined within a multiple line function are local to that function. This means that they are unknown or undefined outside the function.

```
INTEGER A,B,C,D
DEF TESTIT(A,B)
     INTEGER TESTIT,C

     C = A + B
     D = A / B
FEND
```

In the program above, the function TESTIT has 3 local variables. They are the formal parameters A and B, and the locally defined variable C. Note that the function name TESTIT is also declared as an integer within the function. The variables A, B, and C defined before the function are different variables from the three local variables A, B, C.

In the example above, the variable D is not local to the function TESTIT. However, TESTIT accesses and changes the value of D. A multiple line function can access and alter any variable that is available to the main program. That is, a variable that is not defined in a different multiple line function.

Changing D in the function TESTIT is a side effect of the function. These side effects can often cause unexpected results.

The following example shows a function with a local label MORE called by a program with a statement group using the same label MORE. The two labels are different; no confusion results from their use.

```
DEF LOOP(MAX)
     INTEGER MAX

     MORE:
          IF A < MAX THEN \
               A = A + 1 :\
               GOTO MORE
FEND

MORE:
     CALL LOOP
     GOTO MORE
```

4.5 Public and External Functions

Multiple line functions can be compiled separately, forming a module. This module can be linked with another CB-80 module or a module created by a relocatable assembler such as RMAC™. (RMAC is available from Digital Research.)

Note that when combining modules to form a program, only one of the modules can contain executable statements in its executable group. The other modules must only contain multiple line functions.

A function that can be referenced in another module is called a PUBLIC function.

```
DEF THIS.IS.A.FUNCTION PUBLIC
     INTEGER THIS.IS.A.FUNCTION

     PRINT "I AM A PUBLIC FUNCTION"

FEND
```

THIS.IS.A.FUNCTION is a public function. If a module contains this function, and it is linked with another module, the second module can reference THIS.IS.A.FUNCTION. The following program can access function THIS.IS.A.FUNCTION:

```
DEF THIS.IS.A.FUNCTION EXTERNAL
     INTEGER THIS.IS.A.FUNCTION
FEND
CALL THIS.IS.A.FUNCTION
```

In the example above, no code is generated for the EXTERNAL function THIS.IS.A.FUNCTION. The compiler generates the required information so that the linkage editor, LK-80, links the call to function THIS.IS.A.FUNCTION with its definition in another module.

If two modules are linked together only those functions that are public in one module and external in another are linked. Each module can use the same name for functions that are not PUBLIC or EXTERNAL without confusion.

Parameters can be passed to external functions in the same manner as they are passed to a procedure defined in the same module in which they are accessed. No type checking is performed when parameters are passed to an external procedure. It is your responsibility to ensure that corresponding parameters agree in type.

```
DEF ADD(A,B) PUBLIC
     STRING A,B,ADD
```

```
        ADD = A + B
FEND
```

ADD is a public procedure. It can be accessed from another module by using the following external function definition:

```
DEF ADD(STR1$,STR2$) EXTERNAL
     STRING ADD
FEND
```

Note that the parameter names do not have to be the same. However, the function names must be the same and the types of the parameters must agree. The following is an equivalent definition:

```
DEF ADD(S1,S2) EXTERNAL
     STRING ADD,S1,S2
FEND
```

Some linkage editors might restrict the length of external names. (See Appendix E for current restrictions.)

4.6 Linkage With Assembly Language Routines

An external function does not have to be generated by another CB-80 program. It can be an assembly language program. The only requirement is that the assembly language program must observe the CB-80 parameter passing conventions. All parameters are passed on the stack. Integers and real numbers are placed on the stack directly. In the case of strings, a pointer to the string is placed on the stack.

Integers and strings each occupy two bytes on the stack. The values are stored as 16-bit addresses with the low-order byte first. Real numbers are stored as eight byte quantities. The top byte on the stack is the exponent. The eighth byte is the most significant byte of the mantissa. Section 2 explains the format of integer and real numbers.

If the address corresponding to a string parameter is zero, the string is a null string. Otherwise, the address points to the string. The first two bytes of the string represent the length of the string, with the high-order byte first.

If the high-order bit of the string length is a one, the string is a temporary string. The space for temporary strings must be released prior to returning from an assembly language function. The method of releasing strings is machine dependent. Section 12, on LK-80, provides information on releasing strings.

You can use the SADD function, explained in Section 6, to pass the location of a string without having to worry about whether a string is temporary.

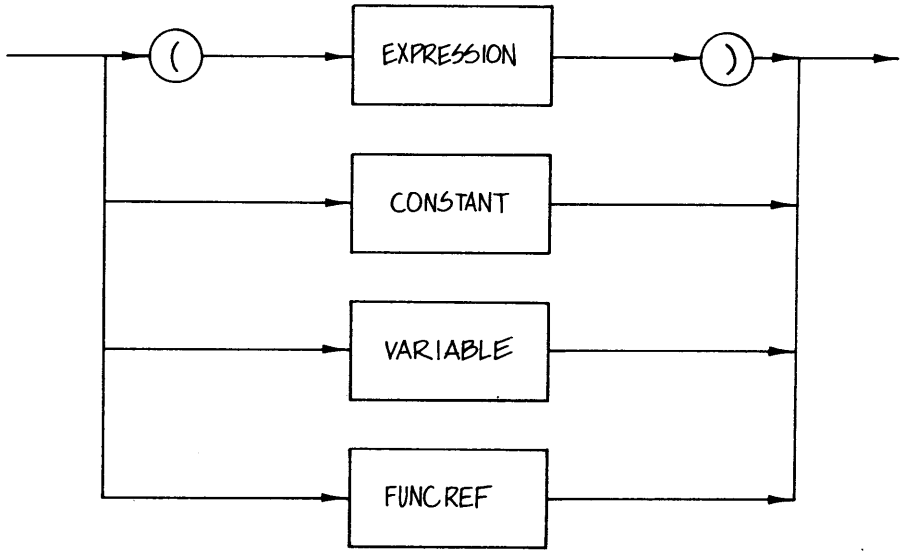
End of Section

Section 5
Expressions and Assignments

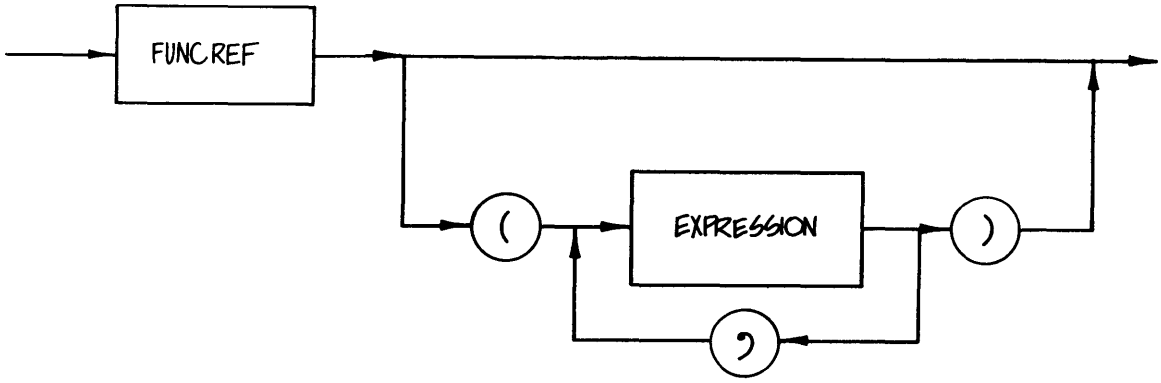
An expression is a combination of operands and operators that evaluate to a single value. Operands are variables, constants, or function references. Logical, relational, and arithmetic operators combine operands. The value of an expression can be saved by assigning it to a variable.

5.1 Operands

An operand is a variable, constant, function reference or an expression enclosed in parentheses.

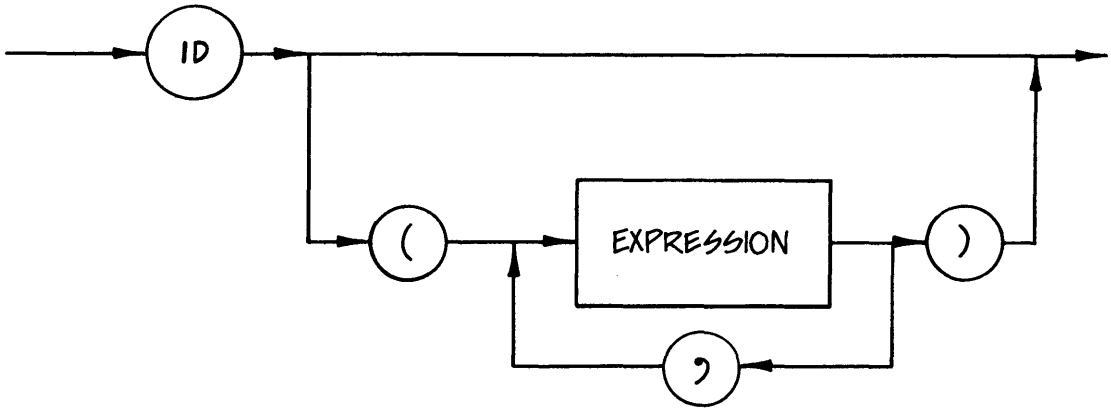


Section 1 discusses constants. There are two types of functions: user defined functions and predefined functions. Section 4 discusses accessing user defined functions. Section 6 explains predefined functions.



This section discusses accessing variables. A variable is a quantity that can change during program execution. Variables are assigned values by assignment statements, explained in this section, or by READ and INPUT statements, explained in Section 8.

The value of a variable is the last value assigned to the variable. If no value has been assigned to a variable, the value is undefined. Some implementations may assign initial values to variables but this is not required. (Refer to Appendix E.)



Variables can be simple variables or subscripted variables. A subscripted variable selects a specific element in an array and treats the variable as a simple variable. Before an array element can be accessed, you must use a DIM statement to allocate space for the array.

The following list shows valid variables:

```
X                       MAT(I,3)
ACCOUNT.NO              SIZE%
SCREEN$(I)              INDEX.MAIN%
?SPACE                  NAMES$(K%)
```

The following variables are invalid:

3RRRRR	Variable names must be identifiers.
X(-3,J)	A subscript cannot be negative.
FINISH:	This is a label.
STOP	A reserved word cannot be a variable.

5.2 Operators

Operators perform unary and binary operations on operands. CB-80 provides three types of operators: logical, relational, and arithmetic. Table 5-1 lists the precedence of operators in CB-80.

Table 5-1. Operators

Operator	General Class
(Nested parentheses)	
^	arithmetic
*, /	arithmetic
+, -, concatenation, unary + and -	arithmetic
<, <=, >, >=, =, <>	relational
NOT	logical
AND	logical
OR, XOR	logical

A higher precedence operator is evaluated before a lower precedence operator. If two operators are of equal priority, they are evaluated left to right. For example the expression:

```
X + Y * Z
```

is evaluated by first multiplying Y by Z and then adding the result to X. This is because multiplication (*) has a higher precedence than addition (+).

In the next expression, the division is performed first because multiplication and division are of equal precedence.

```
X / Y * Z
```

Note: you can alter the order of evaluation by using parentheses.

If the type of two operands differs, CB-80 requires conversion to a common type. The following table lists the rules for converting operands. For example, if the operand on the left is an integer and the operand on the right is real, the integer is converted to a real value.

		Right Operand		
		REAL	INTEGER	STRING
Left Operand	REAL	NO CONV	REAL	ERROR
	INTEGER	REAL	NO CONV	ERROR
	STRING	ERROR	ERROR	NO CONV

NO CONV indicates that no conversion is required; ERROR indicates that operands of those types cannot be used together. An attempt to combine these types of operands results in a compiler error.

Concatenation (+) combines or adds together two strings. It is the only arithmetic operator that can be used with strings.

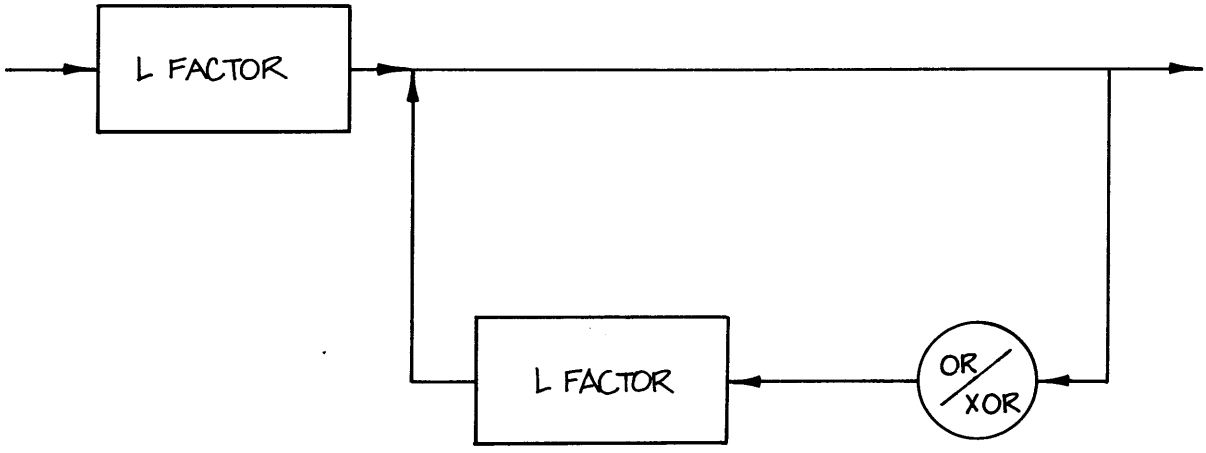
5.2.1 Logical Operators

CB-80 provides logical operators AND, OR, XOR, and NOT. NOT is a unary operator; the others are binary operators. All logical operators require numeric operands. All logical operators treat an operand as a 16-bit binary quantity. If the type of an operand is real, it is converted to an integer prior to performing the logical operator.

The logical OR and XOR operators require two operands and perform a bitwise OR or XOR operation on the operands. The tables below define the OR and XOR operators:

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0



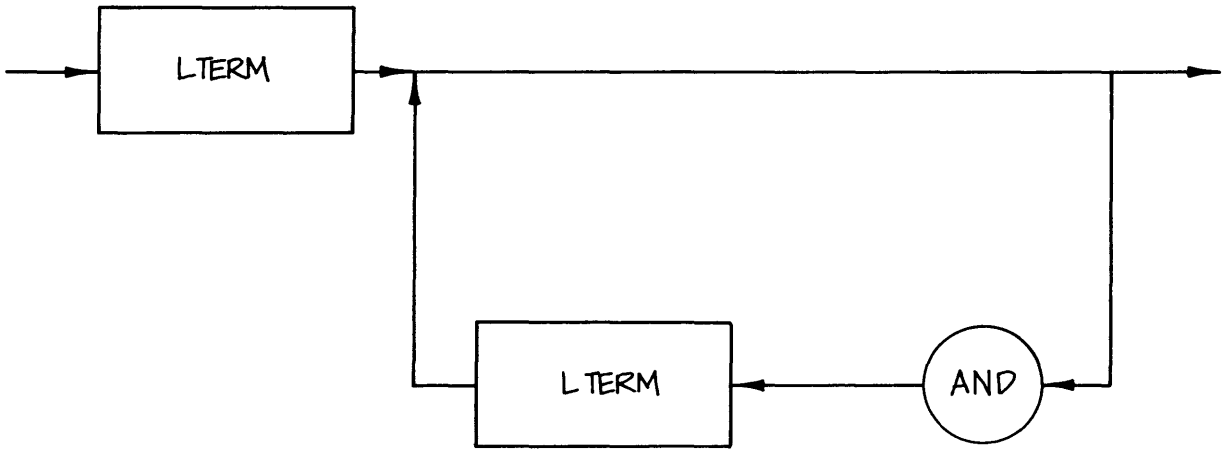
You can use the OR operator to "turn on" or set bits in an integer variable. For example:

```
FLAG% OR 700H
```

ensures that bits 9 through 11 are a 1 (on). The least significant bit is bit 0 and the most significant bit is bit 15.

The logical AND operator requires two operands and performs a bitwise AND operation on the operands. The table below defines the AND operator.

AND	0	1
0	0	0
1	0	1



The AND operator can "turn off" bits in an integer. For example:

```
FLAG% AND 80FFH
```

ensures that bits 9 through 14 are 0 (off).

The logical NOT operator requires one operand. The NOT operator inverts each bit of the operand. This results in the 1's complement of the operand.

The syntax diagram for the NOT operator is shown as part of the syntax diagram for relational operators.

5.2.2 Relational Operators

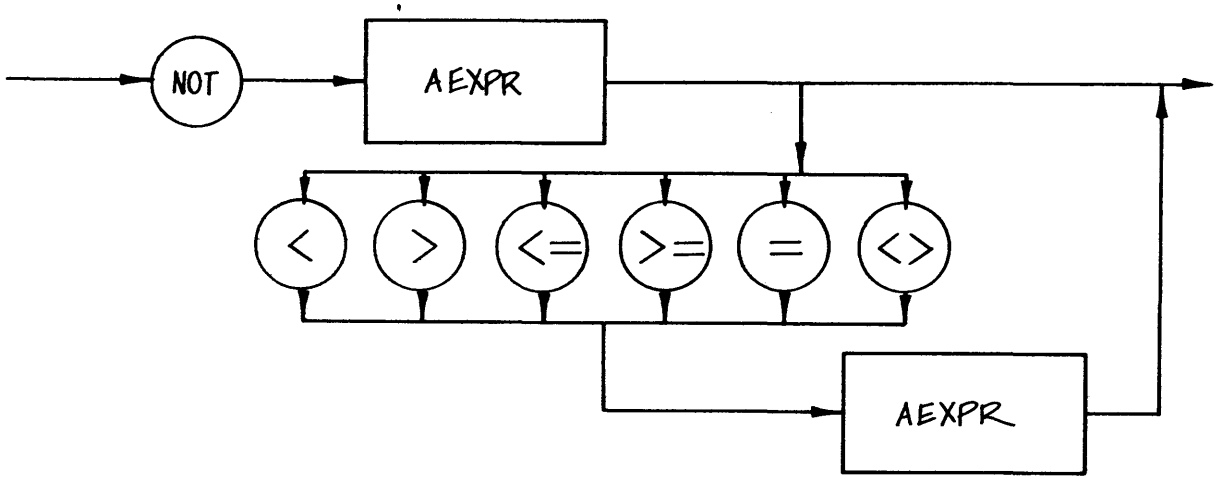
CB-80 has six postfix relational operators that appear in the table below. Relational operators compare two operands and produce an integer result. If the relationship is true, the result is a negative one (all 1 bits), otherwise it is a zero.

Table 5-2. Relational Operators

OPERATOR	RELATION
<	LESS THAN
<=	LESS THAN OR EQUAL
>	GREATER THAN
>=	GREATER THAN OR EQUAL
=	EQUAL
<>	NOT EQUAL

The value resulting from a relational operator is either true (the relationship holds), or false (the relationship does not hold). True is a value of 0FFFFH, and false is zero. This ensures that not true is false.

Expressions containing relational operators are most frequently used with WHILE loops and IF statements. (See Section 7 for a description of the IF and WHILE statements.)



The operands must both be numeric or of type string. If one operand is real and the other is an integer, the integer is converted to a real value before performing the comparison.

The following examples show relational operators with real, string, and integer operands. In each case, the result of the operation is an integer value. In the final example, INDEX% is converted to a real value before the comparison is performed.

```
A < B

ANSWER$ = "STOP"

(I% <= J%) OR (X > Y)

INDEX% <> ANGLE
```

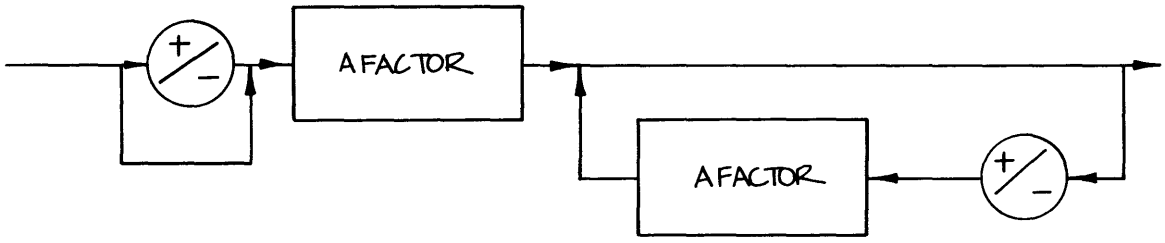
The following expressions show invalid uses of relational operators:

`A$ < B%`	Cannot compare a string and an integer.
`I% >< B%`	Not a valid relational operator.
`X NOT = Y`	Invalid syntax (use <>).

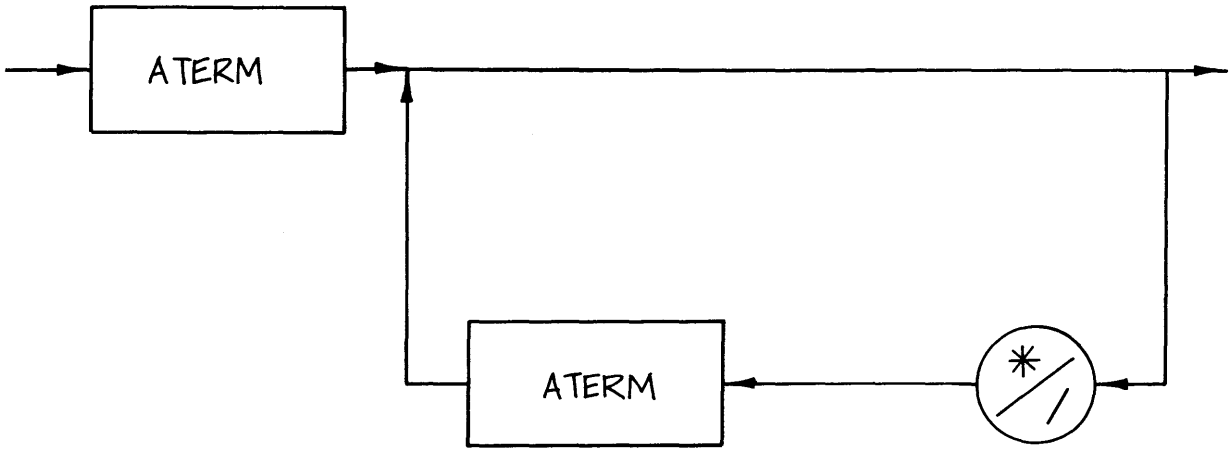
5.2.3 Arithmetic Operators

CB-80 provides five arithmetic operators: addition, subtraction, multiplication, division, and exponentiation. Addition and subtraction can be used as unary or binary operators; the others can only be used as infix operators.

Addition and subtraction can be performed on both integer and real operands. If one operand is real and the other is an integer, the integer is converted to a real value prior to performing the operation. The binary operator for addition (+) concatenates strings.

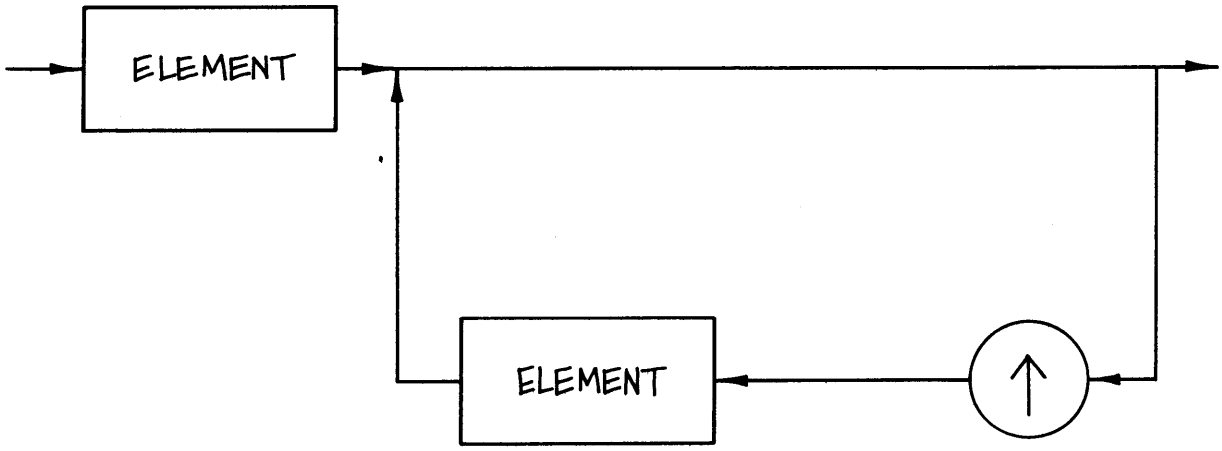


Multiplication and division can be performed on both integer and real operands. If one operand is real and the other an integer, the integer is converted to a real value prior to the multiplication or division.



Exponentiation, the final arithmetic operator, is also performed on both integer and real operands. The first operand is raised to the power represented by the second operand. If one operand is real and the other is an integer, the integer is converted to a real value prior to performing the exponentiation.

A negative real value cannot be raised to a power. An execution error occurs if the operand on the left of the operator is negative.



5.2.4 Expression Overflow

It is possible for some arithmetic operators to overflow the maximum magnitude permitted for the type of operand involved. If the operands are integers, overflow is ignored. If the operands are real values, an execution error occurs when overflow is detected. In the following example, the addition overflows the maximum magnitude of 32767 allowed for integer values.

```
INTEGER X,Y,Z

X = 30000
Y = 30000
Z = X + Y
```

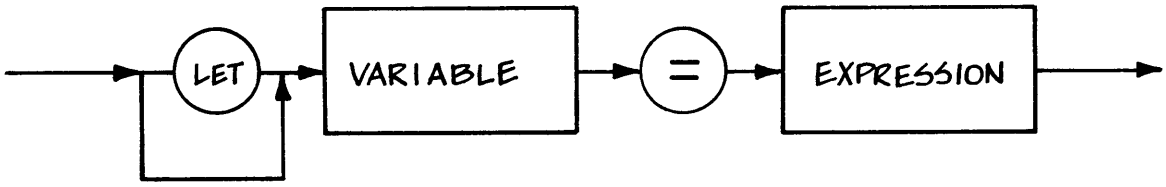
String overflow also causes an execution error. For example, if two strings, each with a length of 20,000 characters, are concatenated, the new string has to be 40,000 characters long. This is greater than the maximum string length and results in an execution error.

Division of a real value by zero results in an execution error, but division of an integer by 0 produces an undefined result.

Overflow of integer calculations is not required to be checked because of the substantial reduction in performance that results on 8-bit microprocessors when such checks are made. A particular implementation might check for these conditions.

5.3 Assignment Statements

The assignment statement sets a variable equal to the value of an expression.



The value of the expression is assigned to the variable at the left of the equal sign (=).

```
LET X = Y + X

LET A$(I,J) = B$ + C$
```

The reserved word LET is optional; normally it is not used, as in the following example.

```
X = A + 1.0
```

If the type of the variable on the left of the equal sign is a string, the expression on the right must evaluate to a string. When the variable is numeric, the expression must also be numeric. The expression is converted to the type of the variable, either integer or real, as shown in the following example.

```
A$ = B$ + C$(I%)

LET X = W * Y + 1.0

I% = X
```

The last expression above causes the variable X to be converted to an integer, and then assigned to the variable I%. If a real value is greater than the maximum magnitude of integers, the result of the conversion is undefined.

The following assignment statements are invalid:

`A$ = X + 1`	Numeric expressions cannot be assigned to a string variable.
`X,Y = A + 1`	Only one variable is allowed on the left of the equal sign.

5.4 Evaluation of Expressions

Expressions are evaluated so that the hierarchy of operators is preserved and that normal algebraic properties (such as commutativity) are retained.

```
X + Y and Y + X
```

These expression always evaluate to the same value (assuming X and Y are variables and not functions). You can use parentheses to control the order of evaluation.

```
X * (Y + Z)
```

The expression above performs the addition of Y and Z prior to multiplying by X. But the expression:

```
X * Y + Z
```

performs the multiplication first.

To provide the maximum opportunity for optimization, no other order of evaluation is implied. In particular, if operations are commutative, CB-80 might use this property to rearrange the expression. This might result in two different implementations giving different values to the same expression. Normally, side effects resulting from the evaluation of functions cause this. In the following example, setting W equal to 2 in the function X causes Y and Z to have different values.

```
DEF X

        W = 2
        X = 4
FEND

W = 1
Y = A + X + W
Z = A + W + X
```

Note: you can combine operators into complex expressions; however, for any implementation, there is a limit on the complexity of expressions. This should not affect most programs. If a compiler error occurs because an expression is too complex, break the expression into two expressions.

The following list shows valid expressions:

```
AMOUNT * (QTY.ONHAND + QTY.ONORDER)
```

```
((I2^2) * R2 * (1.0 - S)) / 746.0

(CINDEX = 2) OR (CINDEX = 5) OR (CINDEX = 6)

I + SIN(X<Y) OR B/C

(((((X + Y)))))
```

The following expressions are invalid:

X + A$	Invalid operands (string and real).
I% - J% K%	Operator missing between J% and K%.
- A$	Unary minus not allowed with string operand.
(X * Y))	Parentheses are not matched.

5.5 Mixed Mode Expressions

Mixed mode expressions are expressions in which a binary operator has an integer and a real operand. In general, mixed mode expressions generate more code and execute more slowly than expressions that do not use mixed mode.

The following assignment has a mixed mode expression.

```
A = X + Y%
```

The operand X is real, and the operand Y% is an integer. The expression:

```
X = X + 2
```

is also mixed mode since the constant 2 is an integer constant. If the expression is written as:

```
X = X + 2.0
```

it is not mixed mode. These last two examples are an exception to the rule that mixed mode generates more code. In these examples, the first expression generates less code than the second one because the real constant (2.0) takes eight bytes to store.

End of Section

Section 6
Predefined Functions

Section 6 describes numeric, string, and other miscellaneous predefined functions. A predefined function returns a value that is used as an operand in an expression. The type of the actual parameters must match the usual convention that integer and real values can be used interchangeably.

In this section, an X parameter represents a real numeric expression. I% represents an integer expression, and an A$ represents a string expression.

6.1 Numeric Functions

Numeric functions calculate commonly used arithmetic and trigonometric functions. The following sections describe each numeric function.

6.1.1 The ABS Function

```
ABS(X)
```

The ABS function returns the absolute value of the argument X. The argument must be numeric and is converted to a real value if it is an integer. ABS returns a real value.

6.1.2 The ATN Function

```
ATN(X)
```

The ATN function returns the arc-tangent or inverse-tangent of the argument X. The argument must be numeric and is converted to a real value if it is an integer. ATN returns a real value.

The ATN function is calculated using Chebyshev polynomials for maximum accuracy. The argument X is expressed in radians.

6.1.3 The COS Function

```
COS(X)
```

The COS function returns the cosine of the argument. The argument must be numeric and is converted to a real value if it is an integer. The COS function returns a real value.

The COS function is calculated using Chebyshev polynomials for maximum accuracy. The argument X is expressed in radians.

6.1.4 The EXP Function

```
EXP(X)
```

The EXP function returns the irrational constant "e" raised to the power of the argument. The argument must be numeric and is converted to a real value if it is an integer. EXP returns a real value.

The EXP function is calculated using Chebyshev polynomials for maximum accuracy.

6.1.5 The FLOAT Function

```
FLOAT(%)
```

The FLOAT function returns a real value equivalent to the integer argument. The argument must be numeric and is converted to an integer if it is a real value.

6.1.6 The INT and INT% Functions

```
INT(X)

INT%(X)
```

The INT and INT% functions convert their arguments to whole numbers. The argument must be numeric and is converted to a real value if it is an integer. Both functions truncate the argument to a whole number.

The INT function returns a real value, while the INT% function returns an integer value.

6.1.7 The LOG Function

```
LOG(X)
```

The LOG function returns the natural, or Naperian, logarithm of the argument. The argument must be numeric and is converted to a real value if it is an integer. LOG returns a real value.

The LOG function is calculated using Chebyshev polynomials for maximum accuracy.

6.1.8 The MOD Function

```
MOD(I%,J%)
```

The MOD function returns the remainder after dividing the first parameter by the second parameter. Both arguments must be numeric and are converted to integer values if either is a real value. MOD returns an integer value.

6.1.9 The SGN Function

```
SGN(X)
```

The SGN function returns an integer value that represents the algebraic sign of the argument. SGN returns a -1 if the argument is negative, 0 if it is 0, and a positive 1 if the argument is positive.

The argument must be numeric and is converted to a real value if it is an integer.

6.1.10 The SIN Function

```
SIN(X)
```

The SIN function returns the sine of the argument. The argument must be numeric and is converted to a real value if it is an integer. SIN returns a real value.

The SIN function is calculated using Chebyshev polynomials for maximum accuracy. The argument X is expressed in radians.

6.1.11 The SQR Function

```
SQR(X)
```

The SQR function returns the square root of the argument. The argument must be a numeric value and is converted to a real value if it is an integer. If the argument is negative, an execution error occurs. SQR returns a real value.

The SQR function is calculated using Newton's method.

6.1.12 The TAN Function

```
TAN(X) = SIN(X)/COS(X)
```

The TAN function is calculated using the identity above. The function returns the tangent of the argument. The argument must be numeric and is converted to a real value if it is an integer. TAN returns a real value.

The argument X is expressed in radians.

6.2 String Functions

This section describes string functions.

6.2.1 The ASC Function

```
ASC(A$)
```

The ASC function returns the ASCII numeric value of the first character of the string argument. The value returned is an integer.

6.2.2 The CHR$ Function

```
CHR$(I%)
```

The CHR$ function returns a one character string that is the ASCII character represented by the value of the argument modulo 256. The argument must be numeric; if it is a real value, it is converted to an integer.

6.2.3 The LEFT$ Function

```
LEFT$(A$,LEN%)
```

The LEFT$ function returns a string that includes the left most characters of the first argument. The length of the string returned is the lesser of the length of the first argument and the value of the second argument.

The second argument must be numeric; if it is a real value, it is converted to an integer. A null string is returned if the second argument is zero. An execution error occurs if the second argument is negative.

```
LEFT$("ABC",2) returns "AB"
```

If the second argument is longer than the length of the first argument, the first argument is returned.

```
LEFT$("ABC",5) returns "ABC"
```

6.2.4 The LEN Function

```
LEN(A$)
```

The LEN function returns the length of the string argument. Zero is returned if the argument is a null string.

6.2.5 The MATCH Function

```
MATCH(PATTERN$,TARGET$,I%)
```

The MATCH function has three arguments: a pattern string, a target string, and a numeric value. The MATCH function returns the position of the first occurrence of the pattern string in the target string or zero if no match is found. Searching starts at the position in the target string determined by the third parameter.

If the third parameter is real, it is converted to an integer. An execution error occurs if the third parameter is zero or negative.

A zero is returned if either the pattern string or the target string is a null string. The MATCH function provides special pattern characters for matching different classes of characters. The following table provides a list of these characters.

Table 6-1. Pattern Characters

Pattern	Corresponding Class of Characters
#	any digit
!	any lower-case or upper-case letter
?	any character

For example:

```
MATCH("##","ABC1A123",1) returns a 6

MATCH("##","ABC1A123",7) returns a 7

MATCH("?!#","3 people are in A1",1) returns a 16
```

Note: the preceding special definitions are ignored if a backslash (\) precedes a character in the pattern string and the next character is a #, !, or ?. The backslash is an escape that overrides the special pattern matching characters. Thus,

```
MATCH("ABC\#","12ABC#",1) returns a 3
```

but

```
MATCH("ABC#","12ABC#",1) returns a 0.
```

6.2.6 The MID$ Function

```
MID$(A$,START%,LEN%)
```

The MID$ function returns a string that is a segment of the first argument. The segment starts with the character position represented by the second argument. The third argument is the length of the segment.

The second and third arguments must be numeric. They are converted to integers if they are real. A null string is returned if the third argument is zero.

```
MID$("ABCD",2,2) returns "BC"
```

An execution error occurs if the second argument is zero or negative, or if the third argument is negative.

A null string is returned if the second argument is greater than the length of the first argument. The following example returns a null string.

```
MID$("ABCD",5,3)
```

6.2.7 The RIGHT$ Function

```
RIGHT$(A$,LEN%)
```

The RIGHT$ function returns a string that includes the right most characters of the first argument. The length of the string returned is the lesser of the length of the first argument and the value of the second argument.

The second argument must be numeric; it is converted to an integer if it is a real value. A null string is returned if the second argument is zero. An execution error occurs if the second argument is negative.

```
RIGHT$("ABC",2) returns "BC"
```

The first argument is returned if the second argument is longer than the length of the first argument.

```
RIGHT$("ABC",5) returns "ABC"
```

6.2.8 The STR$ Function

```
STR$(X)
```

The STR$ function converts the numeric argument to a string that is an ASCII representation of the number. The argument must be numeric; if it is an integer, it is converted to a real value.

The number is converted to a string just as unformatted output is printed to the console. The only difference between the string returned by STR$ and the string printed to the console is that STR$ removes all blanks from the number.

6.2.9 The UCASE$ Function

```
UCASE$(A$)
```

The UCASE$ function returns a string in which the lower-case characters in the argument have been translated to upper-case. Other characters are not altered.

The argument remains unchanged unless it is set equal to UCASE$(A$).

```
A$ = UCASE$(A$)
```

The example above alters the argument A$ but the following assignment does not change A$.

```
B$ = UCASE$(A$)
```

6.2.10 The VAL Function

```
VAL(A$)
```

The VAL function converts the argument into a floating point number. Conversion is identical to that used to input characters from the console.

Zero is returned if the argument is a null string.

6.3 Miscellaneous Functions

This section describes miscellaneous support functions, such as error handling and memory allocation.

6.3.1 The COMMAND$ Function

COMMAND$

The COMMAND$ function returns a string equal to the command line that was used when the program was executed. The command line does not contain the name of the executed program and has leading blanks removed. All lower-case letters are translated to upper-case.

Some operating systems might require that COMMANDS$ be implemented differently.

6.3.2 The ERR Function

ERR

The ERR function returns a two character string equal to the last execution error that occurred. The ERR function returns a null string if no error has occurred. Appendix D lists the possible execution error codes.

Use the ERR function in conjunction with the ON ERROR statement explained in Section 7.

6.3.3 The ERRL Function

ERRL

The ERRL function returns the line number of the last physical source line executed. The ERRL function returns an integer value.

The source program must be compiled with the N toggle, otherwise a zero is always returned.

```
IF ERR="FR" AND ERRL=256 THEN\
      GOTO FDEL
```

6.3.4 The FRE Function

FRE

The FRE function returns a binary value that is the total amount of unallocated or free dynamic memory space. When FRE returns a negative value, it represents a large positive number.

When you use the FRE function, you must ensure that "negative" values are interpreted correctly. In general, if FRE returns a negative value, there is ample space remaining in dynamic memory space. Use the following statement to determine that dynamic memory is at a low level.

```
IF (FRE > 0) and (FRE < MIN.MEMORY%) THEN \
      CALL LOW.MEMORY.WARNING
```

This also applies to the MFRE function described in the next section.

6.3.5 The MFRE Function

```
MFRE
```

The MFRE function returns an integer value that is the largest contiguous area of dynamic memory that is available. The value that MFRE returns is always less than or equal to FRE.

6.3.6 The SADD Function

```
SADD(A$)
```

The SADD function returns an integer value that is the address of the string argument. The address returned is a 16-bit quantity ranging from 0 to 65535. A zero value means that the argument is a null string. A null string can also have a zero length.

The SADD function does not accept an expression as an argument.

6.3.7 The VARPTR Function

```
VARPTR(<VARIABLE>)
```

The VARPTR function returns an integer value that is the permanent storage space assigned to the argument. The argument can be an integer, real, or string variable.

The VARPTR function accepts the following arguments:

Name of a simple variable	VARPTR(X)
Name of a subscripted variable	DIM A$(10) VARPTR(A$)
Element of an array	VARPTR(I%(2))

VARPTR does not accept an expression as an argument.

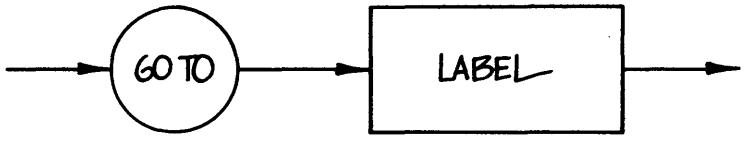
End of Section

Section 7
Flow of Control Statements

Normally, program statements are executed in the order they occur in the program. This section describes statements that alter this execution sequence.

7.1 GOTO Statements

The GOTO statement transfers execution to a statement label specified in the GOTO statement. The label referenced must be defined within the program but need not be defined before it is used in the GOTO statement.



If the GOTO statement is part of the executable group of a multiple line function, then the referenced label must be contained within that function. Likewise, a GOTO statement outside a function cannot refer to a label within the body of the function. In other words, a label within a function is local to that function; its existence is unknown outside the function.

As explained in Section 2, a colon is not part of a reference to an alphanumeric label. The following example shows both the label ENDLESS (with the colon) and a reference to ENDLESS.

```
ENDLESS: GOTO ENDLESS
```

If the label referenced in a GOTO statement is not part of an executable statement, the next executable statement after the label is executed. In the following example, the REM statement is not an executable statement. Thus, execution of the GOTO 100 results in the execution of the PRINT statement.

```
100  REM THIS IS NOT EXECUTED
     PRINT X
     GOTO 100
```

The following examples show valid GOTO statements:

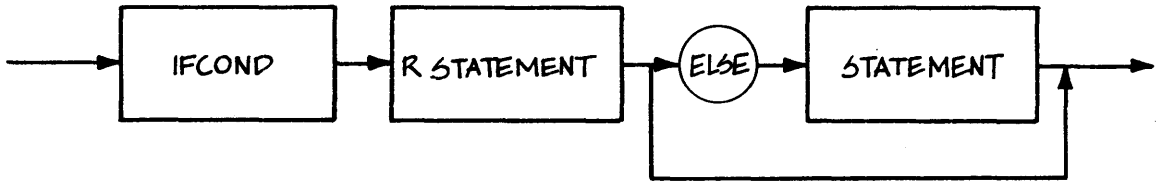
```
GOTO 100

GOTO START.OVER

GOTO 100E-01
```

The following GOTO statements are invalid:

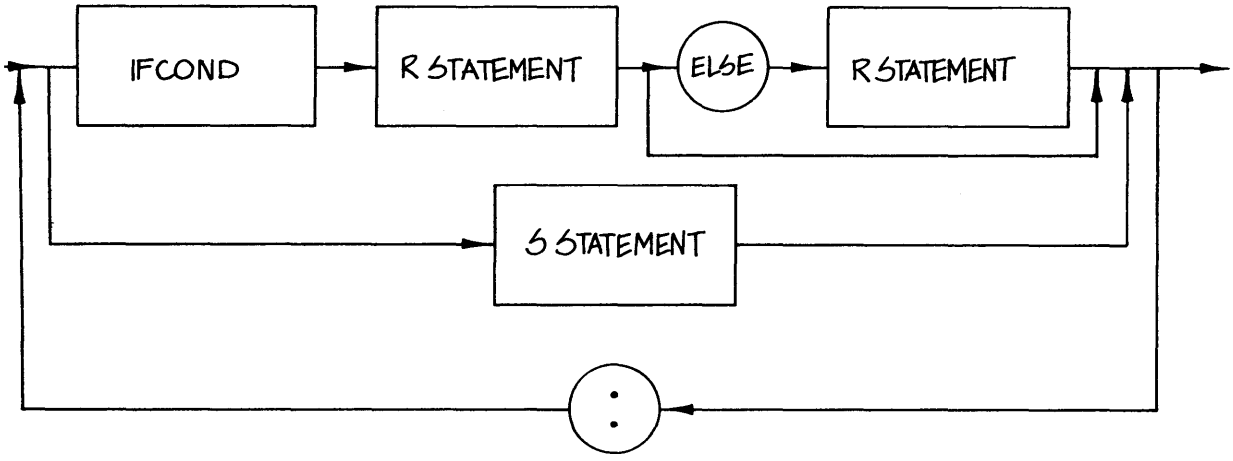
GOTO BEGIN:	The colon is not part of the label referenced.
GOTO 0FFFFH	The hexadecimal constants cannot be labels.
GOTO STOP	A reserved word cannot be a label.

7.2 IF Statements

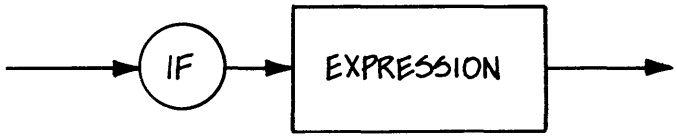
An IF statement allows for the conditional execution of one of two statement groups. The second statement group can be omitted allowing the conditional execution of one statement group.



The syntax diagram for RSTATEMENT is shown below.



The syntax diagram for an IFCOND is shown below.



The expression following the reserved word IF must be a numeric expression. The expression is a "logical expression", having either a true or false value. The expression is false if the value of the expression is zero (0); any other value is true.

The first statement group is executed when the logical expression is true. For example:

```
A = 2
B = 3
IF A < B THEN \
     PRINT "FIRST GROUP EXECUTED" \
ELSE \
     PRINT "SECOND GROUP EXECUTED"
```

In this example, "FIRST GROUP EXECUTED" is printed because the value of A is less than the value of B. If the expression is false, "SECOND GROUP EXECUTED" is printed.

A statement group can contain any executable statement except a function definition. Statement groups can contain any number of statements. Use the colon (:) to group statements together. As shown below, the continuation character (\) allows one statement group to be written over many lines.

```
IF PAGE.BREAK% THEN \
     PRINT FORM.FEED$ :\
     PRINT HEADER$ :\
     PAGE.NO% = PAGE.NO% + 1 :\
     LINE.NO% = 1
```

IF statements can be nested.

```
IF MORE.MASTER THEN \
      IF CURR.REC = M.REC THEN \
            IF MORE.TRANSACTION THEN \
                  PRINT PROCESS.TRANSACTION
```

In some cases, you must use empty or null statements to force the proper pairing of the "IF" statement group with the ELSE statement group.

```
IF I < J THEN \FIRST IF
     IF A = B THEN \SECOND IF
          IF MORE THEN \THIRD IF
               J + J + 1 \
          ELSE \THIS ELSE MATCHES THIRD IF
               I = I + 1 \
     ELSE \THIS ELSE MATCHES SECOND IF
ELSE \THIS ELSE MATCHES FIRST IF
     J = J + 1
```

An ELSE matches the "nearest" IF, as shown in this example:

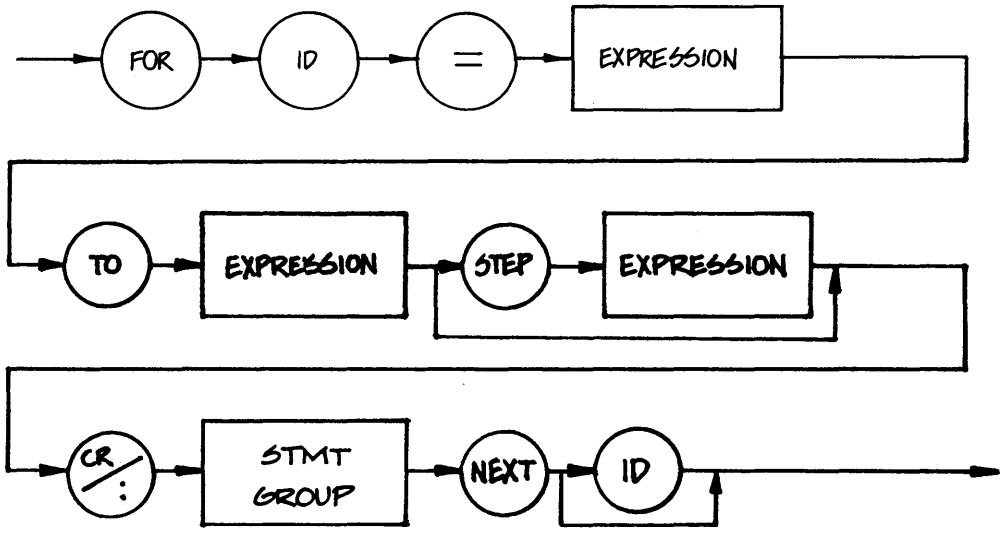
```
IF I < J THEN \FIRST IF
     IF K > L THEN \SECOND IF
          X = 3 \
     ELSE \THIS ELSE MATCHES SECOND IF
          Y = 2
```

The following IF statements are invalid.

```
IF A$ THEN GOTO 10     The expression must be numeric.

IF A < B PRINT X       THEN is missing.
```

7.3 FOR Loops

FOR loops are one of two looping constructs that CB-80 provides. (See Section 7.4 for a discussion of WHILE loops.) A FOR loop consists of a FOR loop header, a statement group, and a NEXT statement. The FOR loop executes the statements in a statement group zero or more times depending on the values in the FOR loop header.



On each iteration through the loop, the index is incremented by the value of the step expression. If the step expression is omitted, the index is incremented a value of 1 (the default value). The general form of a FOR loop header is shown below.

```
FOR index = <initial exp> TO <final exp> STEP <step exp>
```

The index must be an unsubscripted numeric variable. The type of the FOR loop, either integer or real, is the type of the index. Each of the three expressions are converted to the type of the loop. If the index of a FOR loop is an integer, the initial, final, and step expressions are converted to integers providing any of them are real expressions.

If the FOR loop index is real, any integer expressions are converted to real values, as in the following example. Because the index X is real, the final value J% is converted to a real value. The step, which in this example defaults to 1, becomes the real constant 1.0.

```
FOR X = 1 TO J%
```

Programs that use integer indexes and in which the initial, final, and step expressions are integers execute much faster and generate less code than FOR LOOPs with real indexes. In the following FOR LOOP header, no conversion is required because the index and final expressions are both integers.

```
FOR I% = 1 TO J%
```

The following sample program demonstrates the logic used to execute FOR loops.

```
index = <initial exp>
GOTO loop.end
loop.head:

    [FOR loop statement group]

index = index + <step exp>
loop.end:
     if <step exp> < 0 then \
          if index >= <final exp> then \
               GOTO loop.head \
          else \
     else \
          if index <= <final exp> then \
               GOTO loop.head \
          else

[continue execution with statement following NEXT]
```

As the preceding sample program shows, loop termination is based on the sign of the step expression. If the step is positive, then the loop body executes as long as the index is less than or equal to the final expression.

```
FOR I = J TO K STEP 1
   .........

NEXT I
```

The FOR loop statement group above executes K - J + 1 times. If J is greater than K, the loop body is not executed at all.

If the STEP expression is negative, the FOR loop statement group executes as long as the index is greater than or equal to the final expression.

```
FOR I = -5 TO -10 STEP -1
   .........

NEXT I
```

This loop executes 6 times, with I being assigned values of -5, -6, -7, -8, -9, and -10.

On each iteration of the FOR loop, the final and step expressions are reevaluated. The index can be changed within the loop. You can also use the GOTO statement to enter or exit the loop.

If the NEXT statement is followed by an identifier, the identifier must be the same as the index of the loop that the NEXT statement is terminating. The following FOR loops are equivalent:

```
FOR J = 2 TO K STEP 5          FOR J = 2 TO K STEP 5

NEXT                           NEXT J
```

FOR loops can contain any executable statements including another FOR loop.

```
FOR I% = 1 TO N%
     FOR J% = 1 TO M%
          A(I%,J%) = B(I%,J%) + C(I%,J%)
     NEXT J%
NEXT I%
```

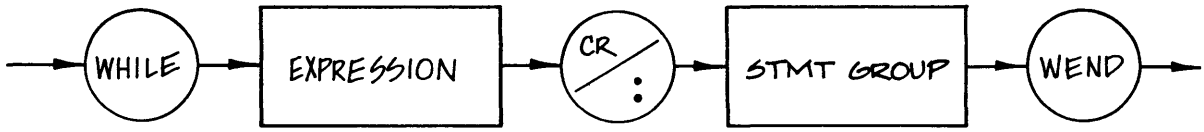
CB-80 does not limit the depth of nesting of FOR loops. However, in a specific implementation, memory constraints during compilation might result in a limit being placed on the number of nested FOR loops. (Refer to Appendix E for specific limits.)

The following FOR LOOPS are invalid:

```
FOR I%(1) = 1 TO N       The index must be a simple variable.
NEXT I%(1)

FOR J = K TO L STEP M    The NEXT identifier must match index.
NEXT K

FOR I = 1 STEP 3         The reserved word TO and the final
NEXT                     value expression are missing.
```

7.4 WHILE Loops

WHILE loops are the second type of looping structure CB-80 provides. A WHILE loop consists of a WHILE loop header, a statement group and a WEND statement. The WHILE loop executes the statements in a statement group zero or more times depending on the value of the WHILE loop header expression.



The expression must be numeric. As with the IF statement, the WHILE loop expression is treated as a logical expression. If the expression evaluates to zero, the statement following the WEND is executed. The statements in the statement group are executed if the value of the expression is other than zero. The expression is evaluated prior to each execution of the statement group.

The following sample program demonstrates the logic used to execute WHILE loops.

```
GOTO loop.end
loop.head:

          [executable group]

loop.end:
     if <expression> <> 0 then
          GOTO loop.head

[continue execution with statement following WEND]
```

The following loop executes indefinitely because the expression is always true.

```
INTEGER TRUE
TRUE = -1

WHILE TRUE

     ........

WEND
```

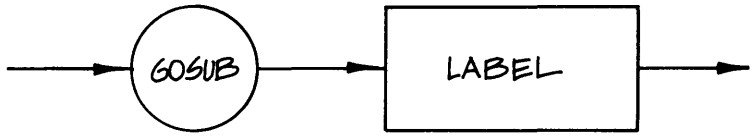
You can enter a WHILE loop by branching to any statement within the statement group. However normal practice is to enter WHILE loops at the loop header.

The following WHILE LOOPS are invalid:

```
WHILE           The expression is missing.
WEND

WHILE A$        The expression must be numeric.
WEND

WHILE A%        The statement group in a while
  DEF A         loop must not contain a function
  FEND          definition.
WEND
```

7.5 GOSUB Statements

The GOSUB statement transfers statement execution to a statement specified by a reference to a label. The address of the statement following the GOSUB statement is saved on a Last-In-First-Out (LIFO) stack so that statement execution can continue with (or return to) the statement following the GOSUB.



The label must be defined within the program but need not be defined prior to its use in the GOSUB statement. If the GOSUB statement is part of the statement group of a multiple line function, then the label must also be part of that statement group. Likewise, a GOSUB statement outside of a given function cannot refer to a label within the body of the function.

If the label is not part of an executable statement, the next executable statement after the label is executed.

```
GOSUB 100

GOSUB PROCESS.ONE.RECORD
```

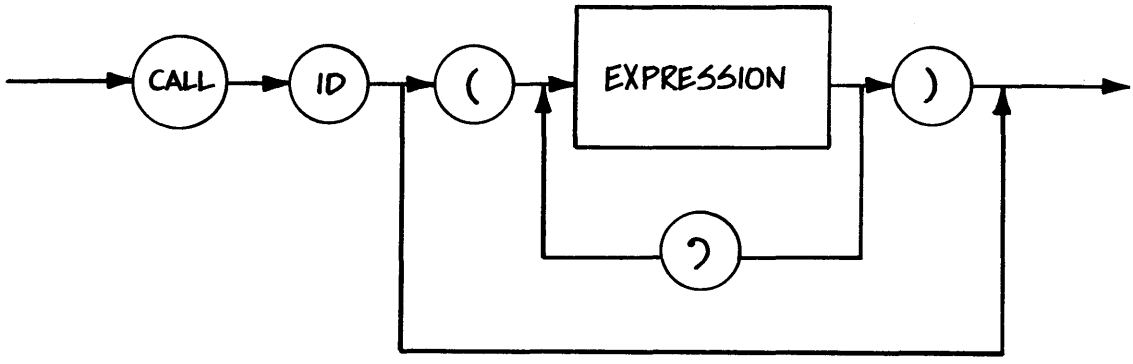
Use the RETURN statement, described later in Section 7.7, in conjunction with the GOSUB statement to continue with the statement following the GOSUB.

The following list contains invalid GOSUB statements:

GOSUB GET.RECORD:	Colon isn't in a reference to a label.
GOSUB 0101B	Binary constants cannot be labels.
GOSUB NEXT	A reserved word cannot be a label.

7.6 CALL Statements

CALL statements pass actual parameters to a multiple line function and then execute the function. The address of the statement following the CALL statement is saved on a Last-In-First-Out (LIFO) stack. So, statement execution can continue with (or return to) the statement following the CALL. A RETURN statement or a FEND statement returns execution to the statement following the CALL.



The number of parameters the CALL statement passes must be the same as the number of formal parameters in the definition of the multiple line function. When the formal parameter is a string, the actual parameter must be a string. However, numeric parameters are converted from integer to real (or real to integer) as necessary.

```
CALL FN.GET.RECORD
CALL GET.REC(FILE.NM$,REC.NO%,AMOUNT)
```

The following list contains invalid CALL statements:

CALL PRINT(REC.NO%)	Reserved word cannot be function name.
CALL FN.A X,Y	Parameters must be enclosed in parentheses.

```
DEF F(A)
  .....
FEND
CALL F(X,Y)
```

An incorrect number of parameters in CALL.

```
DEF F(A$)
  .....
FEND
CALL F(X)
```

A numeric value cannot be passed to a string formal parameter.

The multiple line function referenced in a CALL statement must be defined before it is used in a CALL statement. Use the DEF statement to define a function (see Section 4).

A CALL statement cannot call a single line function or a program label.

7.7 RETURN Statements

RETURN statements return the program to the statement following the last CALL statement, function reference or GOSUB statement. The statement returned to is the last address placed on the LIFO stack by a GOSUB or CALL statement or by a function reference.

If the RETURN statement is returning from a GOSUB or CALL statement, execution continues with the statement following the GOSUB or CALL, but a value is not passed back.

In the following example, the GOSUB statement transfers control to the label ROUTINE: and saves the address of the next statement, in this case the assignment to X. After the RETURN statement executes, the assignment, X=3, executes.

```
ROUTINE:
     Y=2
     Z=30

     .........

RETURN

GOSUB ROUTINE
X = 3
```

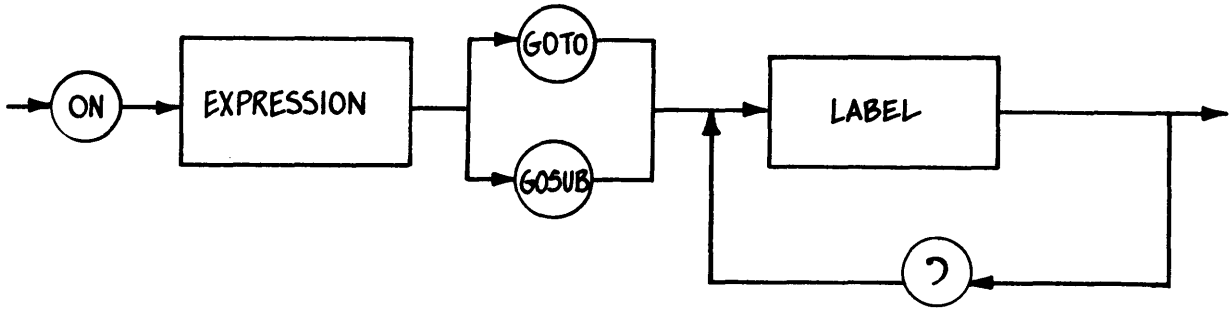
If the RETURN statement is returning from a function reference, the last value assigned to the function name is returned to the expression that referenced the function. In the following example, the function ADD.THEM returns a value and assigns it to the variable X.

```
DEF ADD.THEM(A,B)
     INTEGER A,B,ADD.THEM

     ADD.THEM = A + B
     RETURN
FEND

X = ADD.THEM(23,56)
```

If more RETURN statements execute than there are addresses on the LIFO stack, the results are undefined and an execution error does not occur.

7.8 ON Statements

ON statements transfer execution to one of a number of labels. Control can be passed using a GOTO statement or a GOSUB statement.



The ON statement is similar to the computed GOTO statement in FORTRAN. The expression is evaluated and is used as an index to select one of the labels in the list. The expression must be numeric; a real expression is converted to an integer.

The ON statement must have at least one label in the list; there is no limit on the maximum number of labels in an ON statement.

If the expression evaluates to 1, the first label is selected; if it evaluates to 2, the second label is selected, and so forth. In the following example, the value of I is 3 so control passes to LABEL3 where the PRINT statement prints the number 3. Since the ON

statement was an ON ... GOTO, no return value is retained.

```
I = 3
ON I GOTO LABEL1,LABEL2,LABEL3

LABEL1:
     PRINT 1
     STOP
LABEL2:
     PRINT 2
     STOP
LABEL3:
     PRINT 3
     STOP
```

The labels in an ON statement need not be defined before they are referenced in the ON statement and they can be in any order in the program. The next example shows an ON statement with one label before and one following it.

```
20   PRINT 1

        .......

     ON I GOTO 10, 20
10   PRINT 2
        .......
```

If the ON statement was an ON ... GOSUB, control can be returned to the statement following the ON statement by executing a RETURN statement.

```
I = 2
ON I GOSUB LABEL1,LABEL2,LABEL3
     STOP
     ........

LABEL1:
     PRINT 1
     RETURN
LABEL2:
     PRINT 2
     RETURN
LABEL3:
     PRINT 3
     RETURN
```

In the preceding example, the second label, LABEL2, is selected. When the RETURN statement is executed, control transfers to the STOP statement which is the next statement following the ON

... GOSUB.

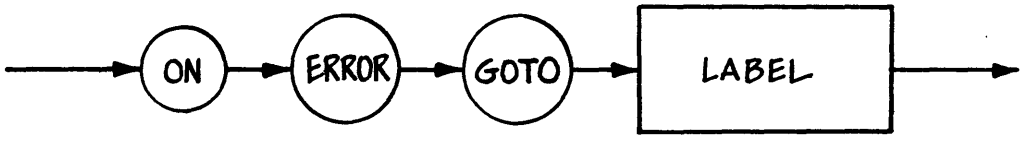
If the index is less than one or greater than the number of labels in the list, the results are undefined. No execution error occurs. Therefore you should always test the index value before executing an ON statement.

The following ON statements are invalid:

```
ON I GOTO 100 200      Comma is missing between labels.

ON B$ GOSUB 12, 23     Expression must be numeric.

ON K-1 10, 20          GOTO or GOSUB is missing.
```

7.9 ON ERROR Statements

The ON ERROR statement traps execution errors allowing the program to process them. The ON ERROR statement is an executable statement that must be executed prior to trapping errors.



When an execution error occurs and the program has executed an ON ERROR statement, execution continues at the first executable statement following the label referenced in the ON ERROR statement. In the following example, if an error occurs after the ON ERROR statement has been executed, the program continues execution at PROCESS.ERROR.

```
ON ERROR GOTO PROCESS.ERROR
PROCESS.ERROR:

        ......
```

When an error occurs, the execution stack is reset. This means that any return addresses are lost. For this reason, an ON ERROR statement must not be used in the statement group of a multiple line function.

If a program contains multiple ON ERROR statements, the last ON ERROR statement executed determines the label that is branched to.

The ON ERROR statement is normally used in conjunction with the ERR and ERRL functions explained in Section 6.

The following list contains invalid ON ERROR statements:

ON ERROR 100	Reserved word GOTO is missing.
ON ERROR GOSUB ERRQ	GOTO is required in place of GOSUB.

7.10 STOP Statements

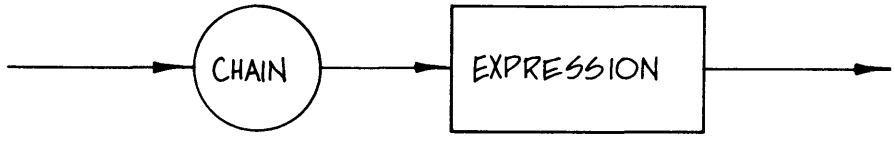
The STOP statement terminates execution of a program. Control returns to the operating system.

Prior to returning to the operating system, any open files are closed.

7.11 CHAIN Statements

The CHAIN statement loads and executes a new program. The CHAIN statement can load two types of programs: an overlay program created by the linkage editor (LK-80), or a directly executable core image (COM) file.

The information concerning the CHAIN statement is general, and examples apply to the CP/M and MP/M operating systems. For more detailed information on linking modules and programs, refer to the linkage editor (LK-80) documentation in Section 12.



The CHAIN statement expression, which must evaluate to a string, is the name of the program to be loaded. If no filetype is specified, a type of OVL is assumed. An execution error occurs if the file cannot be opened.

The following statement loads the file "RPTWRT.OVL" and then executes the new program. All OVL files loaded by a CHAIN statement must have been linked with the last COM file loaded.

```
CHAIN "RPTWRT"
```

The next statement loads and executes the file AR.COM. When a program is loaded, the variables in the data area are set to zero if they are numeric and to null strings if they are string variables. Any variables in the COMMON area remain as they were before the CHAIN statement was executed.

```
CHAIN "AR.COM"
```

If the program being chained to has a COM filetype, and the program has a different name than the last COM file loaded, the COMMON variables are also reset to zero or null strings. This allows a CHAIN statement to load and execute a completely new application.

A CHAIN statement can load a COM file created by languages other than CB-80. The COM files loaded need not be created by LK-80. However, all OVL files loaded must have been created by LK-80. In addition, if a COM file chains to an OVL file, both the COM and OVL files must have been created by LK-80.

The CB-80 run-time support system zeros the data area prior to executing a program. This means that assembly language modules linked with CB-80 modules cannot have initialized data in data segments.

End of Section

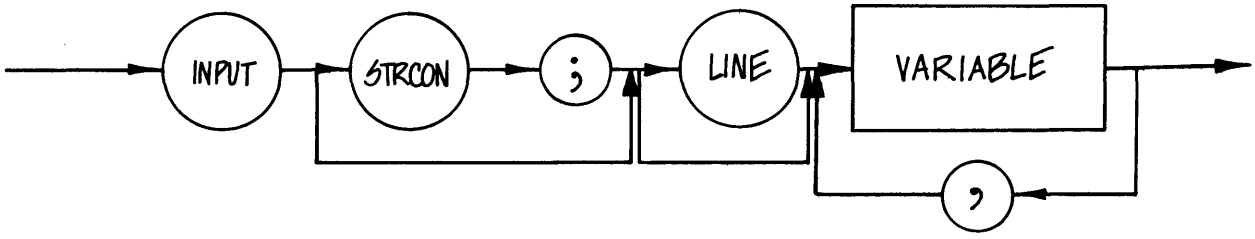
Section 8
Input/Output Processing Statements

Input/Output processing statements allow data to be transmitted between external devices and CB-80 variables. This section explains transfer of data to and from the console device and to the line printer.

This section also explains assigning data in DATA statements to CB-80 variables. In addition, the POKE, RESTORE, RANDOMIZE statements, and predefined functions associated with input and output operations are explained.

8.1 INPUT Statements

INPUT statements accept data from the console and assign the data to program variables.



The simplest form of an input statement accepts data from the console and assigns the data to a list of variables. The following statement inputs three data items from the console and assigns each data item to a variable.

```
INPUT A, B$, C%
```

The data input must contain exactly three data fields. When you enter data in response to this statement, separate the first two fields with a comma and terminate the last one with a carriage return. A field is a string or numeric constant followed by a comma or by the end of the input line.

When an INPUT statement is executed, the compiler prints a question mark (?) on the console followed by one blank space. Then you can enter characters in response to the input statement. The response terminates either with a carriage return or after you enter the maximum number of characters allowed. The maximum is at least 255 characters. (See Appendix E for specific implementation limits.)

All the characters you enter in response to an INPUT statement are echoed at the console. CB-80 supports the normal line editing input commands of the operating system.

Data you enter in response to an INPUT statement must contain a field for each variable in the list. In the example above, three fields are required. Except for the last field, fields are terminated with a comma. The following input statement requires two fields:

```
INPUT A, B%
```

The following is a proper response for this input statement:

```
? 123.45, 45
```

CB-80 prints the question mark and the blank space that follows. If you enter an incorrect number of fields, a warning message appears at the console and you must reenter all the fields.

You can enter strings enclosed in quotation marks. This permits any character except a carriage return to be included in the string. Double quotation marks within the string represent one quotation mark and do not terminate the string.

```
INPUT NAME$
```

The following is a valid response to the preceding statement:

```
"Jones, John"
```

If a string is not enclosed in quotation marks, the first comma ends the string. Any other character except a carriage return can appear in a field.

When a field is assigned a numeric variable, CB-80 converts the entire field to the internal representation corresponding to the class of the variable. If CB-80 encounters an unexpected character in the field, conversion to the internal form terminates.

```
INPUT X
```

The following response to the statement above results in X being assigned a value of 123.45. The character "Q" is not expected as part of a number. Thus, the remainder of the field is ignored. No error message is printed.

```
? 123.45Q+23
```

When you enter data for assignment to an integer variable, and the magnitude of the integer exceeds the maximum magnitude of CB-80 integers (32,767), the assigned value is undefined. As with all integer overflow, no error results.

You can use a prompt string in an INPUT statement. If a prompt string is present, CB-80 prints it in place of the question mark. CB-80 still prints a single blank prior to accepting input.

```
INPUT "Enter three numbers"; A, B, C
```

This statement prints the following prompt on the console:

```
Enter three numbers
```

Following the prompt, one blank is printed and then three fields are accepted as input.

If the prompt string is null, the INPUT statement operates the same as an INPUT statement without a prompt string except that no question mark is printed.

The INPUT LINE statement is a special form of the INPUT statement that accepts one line of input from the console and assigns it to a string variable. The statement:

```
INPUT "What is your name? "; NAME$
```

accepts any characters as input until you enter a carriage return. The entire line, excluding the carriage return, is assigned to the string variable NAME$.

Only one variable can appear in an INPUT LINE statement. If you enter only a carriage return in response to an INPUT LINE statement, a null string is assigned to the variable.

The following statements are valid input statements:

```
INPUT "Enter the data"; A,B,C

INPUT LINE X$
```

The following input statements are invalid.

```
INPUT LINE A        Must be a string variable.

INPUT "Enter" X     Semicolon is missing after prompt.

INPUT A$; C%        Prompt must be a string constant.
```

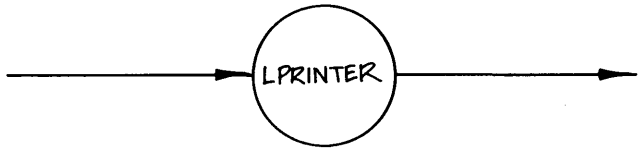
8.2 CONSOLE and LPRINTER Statements

During execution of a CB-80 program, a print control flag determines whether output from a PRINT statement is displayed on the list device or on the console. The print control flag is a special

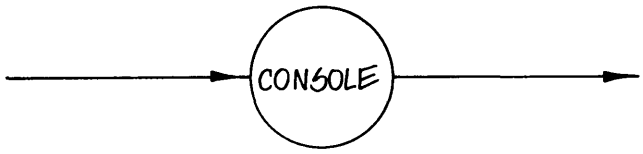
variable maintained by CB-80; you cannot directly access the control flag. The CONSOLE and LPRINTER statements set and reset this flag.

When the print control flag is reset or false, output from PRINT statements prints on the console. When the flag is set, the output goes to the list device. Initially, the flag is reset so the output appears on the console.

The LPRINTER statement sets the print control flag to true so information can be printed on the list device.



The CONSOLE statement resets the print control flag.



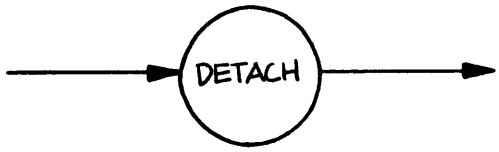
The print control flag does not affect output resulting from INPUT statement prompt strings. The toggle always appears on the console. When either a CONSOLE or LPRINTER statement is executed and the position in the current output line is not 1, a carriage return and line-feed are printed prior to changing the print control flag.

The following example uses the LPRINTER and CONSOLE statements.

```
IF LST.REQ THEN \
     LPRINTER   \
ELSE \
     CONSOLE

 ..............
```

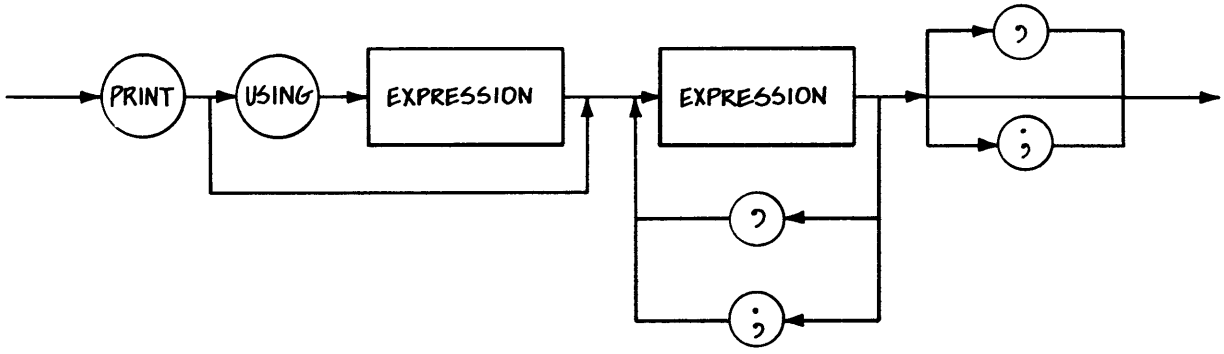
8.3 DETACH Statements

The DETACH statement detaches the printer currently assigned to the program. The DETACH statement is ignored unless MP/M is being used. Normally, DETACH is used in conjunction with the ATTACH function (described in Section 8.10.1).



8.4 PRINT Statements

The PRINT statement prints data on the console or line printer depending on whether the print control flag is false or true.



Section 10 explains the USING option of the PRINT statement which provides formatted output. This section discusses unformatted output.

Each expression in the list is printed on the console or the list device depending on the setting of the print control flag. The following statement prints three fields. The first field starts in column one; each of the remaining fields start at the next column after the last number printed that is a multiple of 20. A new line is started after the last field is printed.

```
PRINT X, Y$, I%
```

The comma forces automatic tabbing after the field has been printed. The tab positions are 1, 20, 40 etc. The next example:

```
PRINT 12,13.78,14
```

prints the following line on the console. In this section and in Section 9, the asterisk (*) marks column 1 and the symbol <NL> indicates that a new line starts.

```
*
12                  13.78                  14<NL>
```

Numeric expressions are printed in two formats depending on the value of the number. The value prints in a fixed decimal format if the number is greater than or equal to 0.01 and less than or equal

to 99,999,999,999,999. If the number is outside this range, the value is printed in scientific notation with one digit before the decimal point.

```
1.0E 32

7.218E-10
```

If a number is negative, a minus sign (-) is printed before the first digit. A positive number has a blank space preceding the first digit in place of the sign. One blank is printed after the number is printed.

Strings are printed as is; no leading or trailing blanks are output and the strings are not enclosed in quotation marks.

```
A$ = "HI"
PRINT A$
```

This statement outputs:

```
*
HI<NL>
```

If two expressions are separated by a semicolon (;), instead of a comma (,), no automatic tabbing takes place. One field follows directly after the last. Numeric fields are still separated by a blank because numbers always have a blank printed after them.

```
A = 3
A$ = "HI"
PRINT A;A$;A$
```

The preceding outputs:

```
*
3 HIHI<NL>
```

If the last expression in a PRINT statement is followed by a comma (,) or a semicolon (;), a new line is not started.

```
PRINT A+B, B-A, A-B,
```

If the last character is a comma as shown in the example above, the tabbing to the next column that is a multiple of 20 occurs but no carriage return is output.

```
PRINT "SAY HI",
```

This statement outputs:

```
*
```

```
SAY HI_____________
```

The underscore (_) indicates one blank was printed.

The trailing comma or semicolon causes the next PRINT statement to output on the same line as the PRINT statement with the trailing delimiter.

```
PRINT "THIS IS ";
PRINT "A SENTENCE"
```

outputs:

```
*
THIS IS A SENTENCE<NL>
```

The next example shows a loop printing a value and automatically tabbing to the next column.

```
FOR I% = 1 TO 3
     PRINT I%,
NEXT I%
PRINT
```

The output from this program is shown below.

```
*
1               2               3               <NL>
```

The following example does not use automatic tabbing.

```
FOR I% = 1 TO 3
     PRINT I%;
NEXT I%
PRINT
```

The output from this program is shown below.

```
*
1 2 3 <NL>
```

The following PRINT statements are invalid:

```
PRINT A+B C+D      Delimiter (, or ;) is missing.

PRINT A +          Expression is incomplete.

PRINT A,,B         An expression is missing.
```

A PRINT statement with no expression list can print blank lines.

```
PRINT
PRINT
```

The two preceding statements each start a new line. Thus two blank lines are printed. The two statements

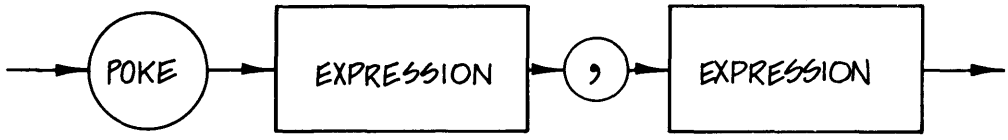
```
PRINT "HI THERE";
PRINT
```

are equivalent to the statement:

```
PRINT "HI THERE"
```

8.5 POKE Statements

The POKE statement places the value of the second numeric expression at an absolute memory location determined by the first numeric expression. The value placed in memory is one byte of data.



The first expression must evaluate to a valid address for the computer being used. However, CB-80 does not verify that the memory address is valid. The second expression, modulo 256, is placed at this memory location.

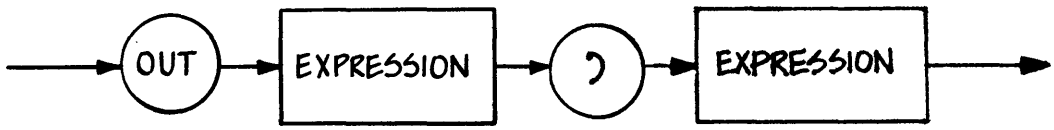
```
POKE MEM.LOC%,VALUE%
```

The absolute addresses assigned to the program code and data area are determined when a module is linked. When using the POKE statement, the effect of linking the program must be taken into account.

The expressions must be numeric; if either expression is real, it is converted to an integer.

8.6 OUT Statements

The OUT statement outputs an integer value to a hardware output port. This function is hardware dependent and might not have the same effect on different processors. In addition, the OUT statement can also interfere with the operating system you are using.

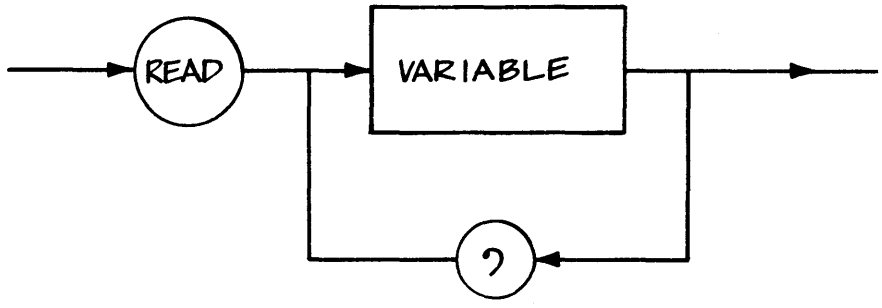



```
OUT PORT%, I%
```

The arguments must be numeric; if either is a real value it is converted to an integer. The first expression must evaluate to a valid port number for the processor being used. CB-80 does not verify that the port number is valid. The second expression, modulo 256, is output to the selected port.

8.7 READ Statements

READ statements accept data defined by DATA statements and assign the values to variables. DATA statements are explained in Section 3.



The following statements assign the value 10 to the real variable X, an integer value 20 to I%, and the string "HI" to the string variable A$.

```
DATA 10, 20, "HI"
READ X, I%, A$
```

The following statements are equivalent to those above:

```
DATA 10, 20
READ X
READ I%, A$
DATA "HI"
```

Each READ statement assigns the next field in the DATA statement to the variable in the next READ statement. All the DATA statements in a program are treated as one consecutive group of

fields.

If the variable in the READ statement is numeric, the field from the DATA statement is converted into the appropriate internal representation. When assigning values to variables with READ statements use the same rules as for the INPUT statement. The statements below assign a value of zero (0) to I%. This is because string "XYZ" is not a valued integer. However, K% is assigned a value of 71.

```
DATA "XYZ", "71"
READ I%,K%
```

If you attempt to read a field past the last field in the last DATA statement in the program, an execution error occurs. Executing the following statements results in an execution error unless there are other DATA statements in the program.

```
DATA XYZ
READ A$, B$(I)
```

The following READ statements are invalid:

```
READ A B          Comma is missing.

READ I(J%);V%     Variables must be separated by commas.

READ A,,B         Variable name is missing.
```

The RESTORE statement, explained in the next section, allows the DATA statements to be reused.

8.8 RESTORE Statements

The RESTORE statement repositions the pointer into the data area, so the next value read with a READ statement will be the first item in the first DATA statement in the program.

The following is an example of a RESTORE statement:

```
RESTORE
```

8.9 RANDOMIZE Statements

The RANDOMIZE statement seeds the pseudo-random number generator so the RND function (see Section 8.10.8) generates random numbers.

On operating systems that do not provide a time of day function, the seed is generated using the time taken to respond to INPUT statements. If the time of day is available, it generates a random seed.

Thus, on operating systems that do not have the time of day available, it is necessary to execute an INPUT statement prior to using the RANDOMIZE statement. In any event, a RANDOMIZE statement must be used prior to using the RND function to generate a different pseudo-random series each time the program is executed. Section 8.10.8 explains the RND function.

8.10 Input/Output Predefined Functions

8.10.1 The ATTACH Function

The ATTACH function returns an integer that is true if the selected printer can be attached by the program. Otherwise, ATTACH returns a false value.

```
ATTACH(PRINTER.NO%)
```

8.10.2 The CONSTAT% Function

The CONSTAT% function returns an integer set equal to the console status. If a character has been entered at the console but not yet read, CONSTAT% returns "true", which is a negative one. Otherwise, CONSTAT$ returns a false or zero value.

```
CONSTAT$
```

8.10.3 The CONCHAR% Function

The CONCHAR% function returns the ASCII integer value of the next character typed at the console and displays that integer on the screen.

The lower eight bits of the returned value are the binary representation of the ASCII character read from the console. The high-order eight bits are always zero.

CONCHAR% always reads one character from the console. If no character has been entered, CONCHAR% waits until a character is entered at the console.

```
CONCHAR%
```

For example, if you enter an upper-case letter "A" at the console, the CONCHAR% function returns a value of 65.

8.10.4 The INKEY Function

The INKEY function returns an integer equal to the next character entered at the console. Unlike the CONCHAR% function, INKEY does not echo the character at the console.

The lower eight bits of the returned value are the binary representation of the ASCII character read from the console. The high-order eight bits are always zero.

```
INKEY
```

INKEY is useful when control characters or other special characters might be input and you do not want these characters to be printed. INKEY can also accept passwords. Some operating systems might require that INKEY be implemented identically to the CONCHAR% function.

8.10.5 The INP Function

The INP function returns an integer equal to an 8-bit value input from the I/O port selected by the argument. This function is hardware dependent and might not have meaning on certain processors. In addition, the INP function might interfere with the operating system being used.

The argument must be numeric; if it is a real value, it is converted to an integer. The argument must evaluate to a valid port number for the current processor. CB-80 does not check the validity of the port number.

```
INP(PORT%)
```

8.10.6 The PEEK Function

The PEEK function returns an integer equal to the value of the memory location selected by the parameter. The memory location must be valid for the computer being used. However, CB-80 does not check on the validity of the memory address. The parameter must be

numeric; if it is real, it is converted to an integer.

```
PEEK(MEM.LOC%)
```

8.10.7 The POS Function

The POS function returns the current position in the output line. The value returned is an integer.

POS returns the number of characters plus one that has been output to the console or list device since the last carriage return. In other words, POS returns the next position in which a character will be printed.

Output to the console can be generated by PRINT statements or by INPUT statement prompts.

```
POS
```

The following statements

```
PRINT
PRINT POS,POS
```

print the numbers 1 and 20 starting in columns one and twenty, respectively. The value POS returns might not be valid if you output characters that change the cursor position, such as clearing the screen. The CONCHAR% function can invalidate the value POS returns.

8.10.8 The RND Function

The RND function returns a real value that is a uniformly distributed pseudo-random number between 0 and 1.

```
RND
```

8.10.9 The TAB Function

The TAB function prints blank characters until the value POS returns is equal to the argument. If the value of the argument is less than or equal to the current position to be printed, a new line starts and then the TAB function executes.

The argument must be numeric; if it is a real value, it is converted to an integer. A zero or negative argument causes an execution error.

```
TAB(I%)
```

If the console cursor position has been changed with special control characters, or if the position has been changed with the CONCHAR% function, the TAB function does not provide the desired results.

You can analyze the TAB function in PRINT statements. The following statement prints the string "HI" starting in column 19.

```
PRINT TAB(19), "HI"
```

You can blank out a portion of a line with the following PRINT statement:

```
PRINT TAB(20);
```

End of Section

Section 9
File Processing Statements

A file is a collection of data items stored on an external device such as a floppy disk or Winchester hard disk. CB-80 is not concerned with the physical storage of the data but rather with the logical organization of the data. This section explains input and output between the file system and CB-80. Sections 9.2 through 9.5 explain the statements that open or create files, access files, and close or delete files. In addition, Section 9.6 explains predefined functions that involve file accessing.

9.1 File Description

In general, CB-80 files are made up of ASCII characters. This allows the file to be conveniently displayed on the system display using operating system utilities. Binary files can be built and accessed with certain restrictions explained in this section.

CB-80 supports two types of files: stream and fixed. In a stream file, information is placed in the file as a stream of fields with no record structure. The file is a continuous stream of individual data items. There is no implied relationship between the data items.

Either a comma or a new line character separate each field in a stream file from the next field. With most operating systems, the new line characters are a carriage return followed by a line-feed.

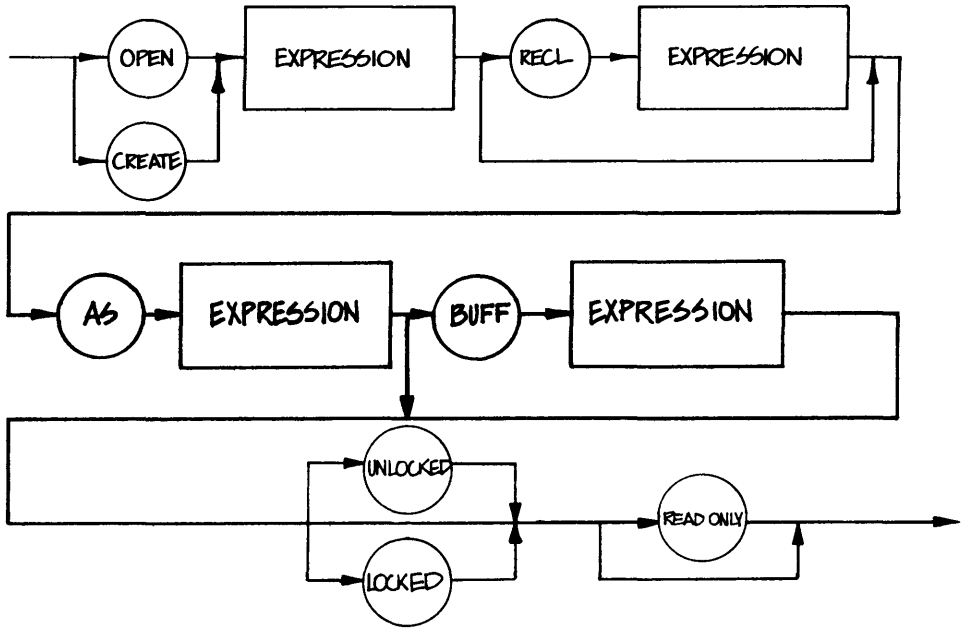
Fixed files also have fields of data separated by commas. However, the fields are grouped into fixed length records. Unused space in records is padded with blanks. The new line characters terminate the record.

In fixed files, the new line characters are part of the record. Thus, the minimum record size is two bytes. Of course, no information can be stored in a file with a record length of two. The maximum record length must be expressed as an integer value.

9.2 OPEN and CREATE Statements

Before data items can be written to a file or read from a file, an interface must be established between CB-80 and the operating system. Characteristics such as the device selection, filenames, and buffer requirements must be defined.

CB-80 provides two statements to define files: the OPEN and CREATE statements. The OPEN statement accesses existing files; the CREATE statement creates a new file with no data in it.



The first expression in an OPEN or CREATE statement is a string expression that evaluates to a valid filename for the operating system. A particular operating system might restrict the characters in the filename and the length of the name.

The expression following the reserved word AS assigns a CB-80 file identification number to the file. All future references to the file use this number. The file identification number can be any numeric expression. If the expression evaluates to a real value, it is converted to an integer. An execution error occurs if the value is zero or negative, or if it is greater than the maximum number of files that can be open at one time. (See Appendix E for current limits.)

A file is open when it has been assigned a file identification number by an OPEN or CREATE statement. You cannot use the same file identification number for two files open at the same time. An execution error occurs if the file identification number in an OPEN or CREATE statement is currently assigned to another file. The number of files that CB-80 allows to be open at one time depends on the implementation. (See Appendix E for current limits.) Some operating systems might impose further restrictions on the number of files that can be open at one time.

```
OPEN "TEST" AS 4

CREATE W.DISK$ + W.NAME$ AS WORKFILE%
```

The filename and file identification number must appear in every OPEN and CREATE statement. The other information is optional.

If a file has fixed length records, specify the record length following the reserved word RECL. The following statement opens a file named MASTER with a record length of 700 bytes.

```
OPEN "MASTER" RECL 700 AS 1
```

MASTER is assigned a file identification number of 1. The record length can be any numeric expression. Real values are converted to integers.

```
CREATE NAME$ RECL FIELD1% + FIELD2% + 2 AS J%
```

When a file is opened with the RECL option, the file is a fixed file.

The reserved word BUFF specifies the number of internal buffers to maintain for the file. If no buffers are specified, a value of one is assumed. The size of a buffer depends on the implementation, but is normally chosen so that it is the amount of data that can be accessed by one call to the operating system. (See Appendix E for current specifications.)

```
OPEN A$ AS 4 BUFF 10
```

The statement above opens a file with 10 buffers assigned for its use. Multiple buffers are always stored consecutively in memory. Use the MFRE function to determine the amount of available memory prior to choosing the number of buffers.

The BUFF option cannot specify more than one buffer when the file is accessed randomly. Random access is explained Section 9.3.

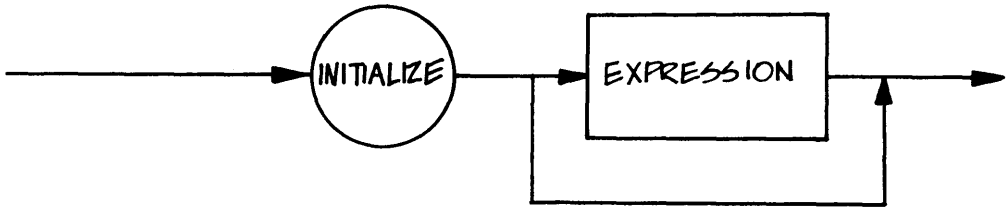
The amount of buffer space CB-80 requires is independent of the record length. CB-80 does not require that the complete record be held in memory at one time. CB-80 provides all the deblocking necessary to support large records in a system with limited memory.

The following OPEN and CREATE statements are invalid:

`OPEN "TEST"`	File identification is missing.
`CREATE A$ AS 1 RECL 100`	Reserved words are in wrong order.
`CREATE 3 AS 2`	Filename must be of type string.
`OPEN FN$ BUFF 10 AS 2`	Reserved words are in wrong order.

The INITIALIZE statement resets the operating system after diskettes or other storage media have been replaced. This prevents the operating system from writing data to the wrong place on the

storage media.



The INITIALIZE statement must not be executed until the swapping is complete and the devices on which the media is placed are ready.

If the expression is present, it is used as a bit pattern to select which drives to reset. For example, the statement

```
INITIALIZE 11B
```

initializes only drives A and B. The expression is only required in multi-user systems where other users might prevent resetting all the drives.

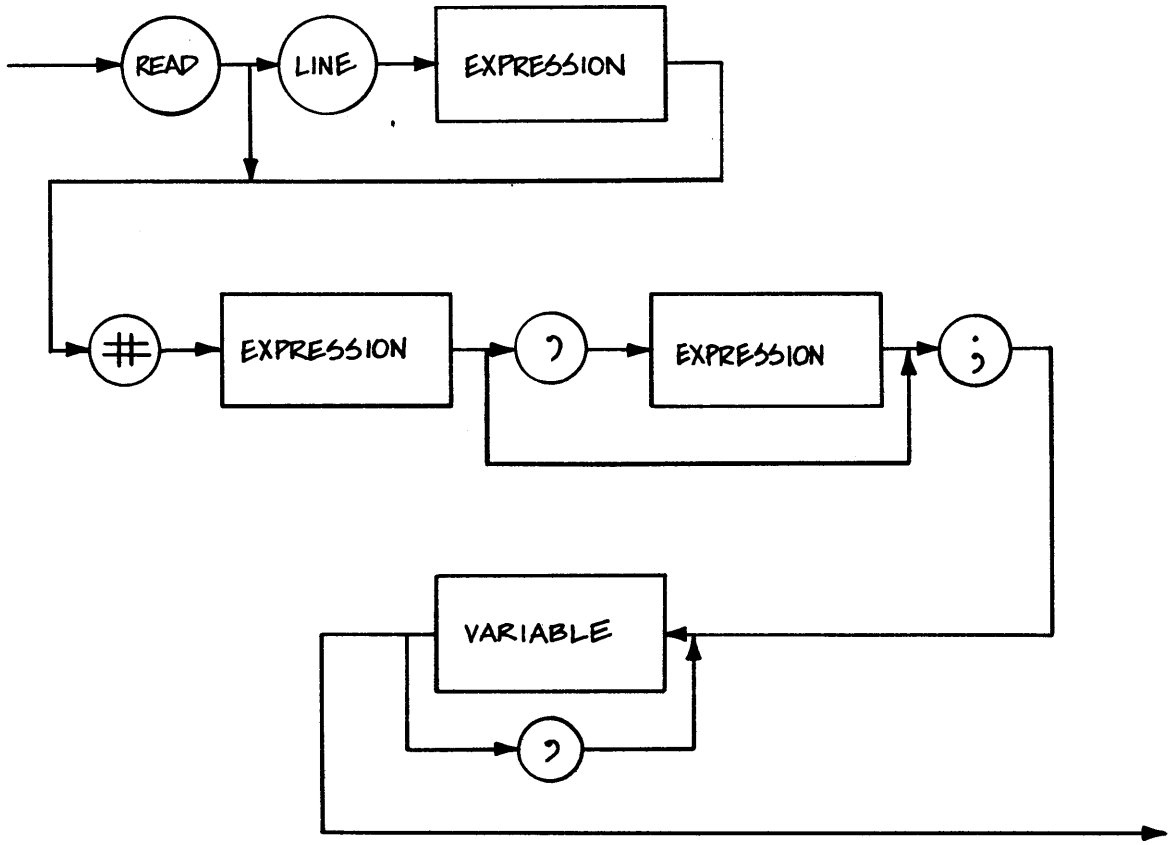
9.3 File Accessing Methods

You can access files in three ways: sequentially, randomly, or one byte at a time. You can use these methods interchangeably with the provision that only fixed files can be accessed randomly.

The following sections explain the file READ, file PRINT, and PUT statements.

9.3.1 Reading Files

Files can be read sequentially and randomly using the file READ statement.



A file READ statement specifies the file identification number for the file to be read, and a list of variables to which data items read from the file are to be assigned. Optionally, you can specify a record number to select the record to be read. You can only use this option with fixed files.

The file identification number and optional record number can be any numeric expression. If either expression is real, it is converted to an integer value. An execution error occurs if the file identification number does not evaluate to an integer assigned to an open file.

```
READ # 1; A, B$, C%
```

This statement reads the next sequential record from a file with a file identification number of 1 and assigns the first three fields to the variables A, B$, and C%. In the case of variables A and C%, the fields are interpreted as numbers and converted to the internal format for real and integer variables, respectively.

There is a fundamental difference in the way fixed record length files and stream files are treated when the files are read. If a file has fixed records, any fields that the READ statement did not read are skipped. The next sequential read reads a new record even if fields were left unread in the previous record.

With a stream file, one field is read after another and no logical organization is assumed. Consider a file with the following records:

```
1,2,3,4CRLF

5,6,7,8CRLF

9,10,11,12CRLF
```

You can use the following READ statements to access this file:

```
READ # 1; A,B,C

READ # 1; D

READ # 1; E,F
```

If the file is a stream file (no record length was specified when the file was opened), the variables A through F are assigned the following values:

```
A = 1   B = 2   C = 3
        D = 4
    E = 5   F = 6
```

However, if the file was opened with a record length specified, the variables are assigned the following values:

```
A = 1   B = 2   C = 3
        D = 5
   E = 9    F = 10
```

Note: if the records are fixed length records, they are padded with blanks.

Another difference between reading fixed length files and stream files occurs when a carriage return is encountered as a field delimiter. If the file is fixed and an attempt is made to read past a carriage return, an execution error occurs. When reading a stream file, a carriage return is treated just like a comma. Thus, when

reading a fixed file, one READ statement reads one record assigning the fields in the record to variables in the variable list.

A READ statement can select a specific record to read instead of reading the next sequential record. The file being read must be a fixed record length file. This type of access is called random access.

```
READ # 1, 12; A, B, C(I,J)
```

The statement above reads the twelfth record from file 1. The first three fields in the record are assigned to the variables A, B, and C(I,J). If a record in this file has less than three fields or the file was a stream file, an execution error occurs.

The first record in a file is record one. An execution error occurs if a READ statement uses a record number of zero. The record number is treated as an unsigned sixteen bit integer. This means that "negative" record numbers can be used for record numbers greater than 32767.

If an attempt is made to read a file past the last record in the file, CB-80 reports that the end of file has been reached. The section on file exception processing explains how you can process an end of file condition. An end of file exception also occurs when a random read attempts to read a record that does not exist.

Sometimes it is necessary to position to a specific record in a file and then read the file sequentially. The following statement positions file 1 to the beginning of record 7. No data is read from the file.

```
READ # 1, 7;
```

An execution error occurs if the file is a stream file.

The READ LINE statement is similar to the INPUT LINE statement explained in Section 8. The READ LINE statement reads one complete line of data from a file and assigns the information read to a string variable. Only one variable can be used after the reserved word LINE and it must be a string variable.

The following statement reads the next sequential record from the selected file and assigns the entire record up to but not including the new line characters to the string variable D$.

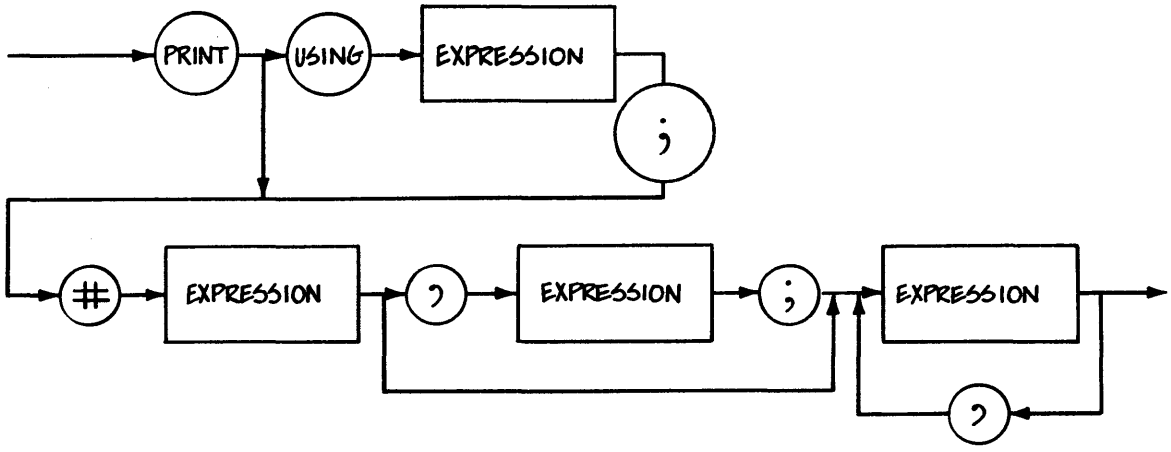
```
READ #FILE.NO%; LINE D$
```

The READ LINE statement can also read a random record, as shown below.

```
READ #F%, R%; LINE X$
```

9.3.2 Writing to Files

Files can be sequentially or randomly written to using the file PRINT statement. This section describes unformatted output to files. Section 10 explains formatted output.



A file PRINT statement specifies the file identification number for the file being printed to, and a list of expressions that are evaluated and output to the file. An optional record number can be specified to select the record to output. Use this option only with fixed files.

The file identification number and optional record number can be any numeric expression. If either expression is real, it is converted to an integer value. An execution error occurs if the file identification number does not evaluate to an integer assigned to an open file.

```
PRINT # 1; A%, B, C$
```

This statement prints three fields to the next sequential record in the file with a file identification number of 1. The first two fields are separated by commas and the last field is followed by new line characters.

When a string is output to a file, it is enclosed in quotation marks. Numbers are output to a file following the same formatting rules used for output to the display.

With a fixed file, sufficient blank characters are output after the last field and before the new line characters. This ensures that each record is the length that was specified when the file was opened. If the data output to the file results in a record length that exceeds the fixed record length, an execution error occurs.

```
OPEN "MASTER" AS 3
X = 21.73
Y = .00007
I% = -72
A$ = "THIS IS A FIELD"
PRINT # 3; X, Y, I%, A$
```

Execution of the program above writes the following record to the file "MASTER":

```
*
21.73,7E-04,-72,"THIS IS A FIELD"<NL>
```

In the program above, substitute an OPEN statement with a record length of 40.

```
OPEN "MASTER" RECL 40 AS 3
```

The record that is output with the substituted OPEN statement is shown below:

```
*
21.73,7E-04,-72,"THIS IS A FIELD"____<NL>
```

An execution error occurs if the record length is less than 34.

A file PRINT statement can direct output to a specific record in a file. This type of access is called random access. To use random access, the file must be a fixed record length file.

```
PRINT #3, 4; C(I), A$+B$
```

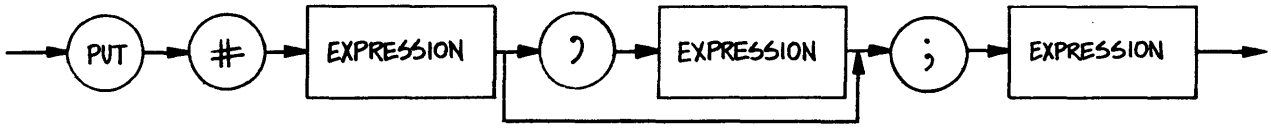
The statement above outputs record four to the file using three as a file identification number. The record contains two fields.

A file exception occurs if a file PRINT statement attempts to output to a file and the file system has insufficient space. Section 9.5 explains how you can trap this condition.

The following file PRINT statements are invalid:

```
PRINT 3; A          Pound sign is missing.

PRINT #I, J;        Expression list is missing.

PRINT # 2; A+1; B   Commas must separate expressions.
```

The PUT statement writes one byte to the selected file. The byte can be any value between 0 and 255.



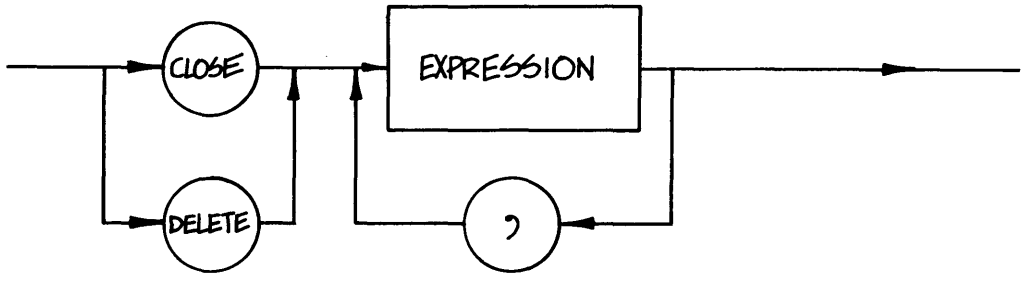
Both expressions must be numeric; if one of them is a real expression, it is converted to an integer.

The PUT statement allows binary data to be written to a file. No delimiters or other characters are added to the data output.

```
PUT 3, I%
```

9.4 Terminating Access to Files

CB-80 provides two statements that terminate access to files: the CLOSE and the DELETE statements. To use these statements, the file must be open.



The CLOSE statement tells the operating system that no further access to the file is required. Any interfaces established by the OPEN or CREATE statement are terminated. All information in the file is retained.

```
CLOSE 3

CLOSE TEMP1%, TEMP2%
```

The DELETE statement instructs the operating system to remove the file from the system directory. No information about the file is retained.

```
DELETE 1

DELETE INDEX% + 3
```

The expressions in CLOSE or DELETE statements must be numeric; if they evaluate to real values, they are converted to integers. Each expression must evaluate to a valid file identification number and that number must refer to an open file. Otherwise, an execution error occurs.

After the CLOSE or DELETE is complete, you can use the file identification number again. If an IF END statement, explained in the following section, is associated with the file identification number, the association terminates.

9.5 File Exception Processing

The IF END statement traps file system exceptions and allows you to take appropriate action.

The label reference must refer to a label defined within the scope of the IF END statement. The label need not be defined prior to its use in an IF END statement.

The IF END statement is an executable statement. It must be executed before it can trap file exceptions. A given IF END statement only applies to the one file that the expression selects.

The expression selects the file identification number of the file for which you want exception processing. The expression must be numeric; a real expression is converted to an integer.

```
IF END # 1 THEN 200

IF END # WORK.1% THEN FILE.EOF
```

The IF END statement traps the following types of exceptions:

- READ PAST END OF FILE
- DISK OR DIRECTORY FULL DURING PRINTING TO A FILE
- ATTEMPT TO OPEN A FILE THAT DOES NOT EXIST

If any of these exceptions occur, the file processing system determines if an IF END statement has been executed for the file identification number of the file. If an IF END statement is in effect, execution continues at the statement with the label referenced in the IF END statement. Otherwise an execution error occurs.

```
IF END # 3 THEN 200
       .......

200 REM PROCESS EXCEPTION FILE 3
```

When transferring control to the exception processing routine, all return addresses saved on the LIFO stack are retained.

The following IF END statements are invalid:

`IF END 7 THEN 200`	Pound sign is missing.
`IF END # 7 THEN GOTO QUITE`	GOTO not allowed.
`IF END # I% THEN PEOF:`	Label reference cannot have colon.

A program can have any number of IF END statements for the same file identification number. The most recently executed IF END for a given identification number is the IF END statement in effect when an exception occurs.

An IF END statement can use a file identification number that is not currently being used by an open file. This allows the IF END to trap exceptions when an OPEN statement is executed.

9.6 File Predefined Functions

9.6.1 The GET Function

```
GET(FILE.ID%)
```

The GET function accepts one byte of data from the file selected by the parameter. The parameter must be numeric; if it is real it is converted to an integer.

GET returns binary data from a file. The value returned by the GET function is an integer between 0 and 255.

9.6.2 The LOCK Function

```
LOCK(FILE.ID%,REC%)
```

The LOCK function locks a record in the file selected. Record locking prevents other programs from updating that record. The LOCK function returns the value returned by the operating system when an attempt is made to lock the record. Normally, a zero means that the record was successfully locked.

Both arguments must be numeric; if either evaluates to a real value, it is converted to an integer.

If your operating system does not support record locking, no action occurs and LOCK returns a value of zero.

9.6.3 The RENAME Function

```
RENAME(NEW$,OLD$)
```

The RENAME function renames a file. The file being renamed, which is the first parameter, must not be open. The arguments must both be string expressions. The value returned by RENAME is an integer value that is true (-1) if the rename was successful and false (0) if the new filename already exists.

An execution error occurs if the new filename already exists.

9.6.4 The SIZE Function

The SIZE function returns the size of the file specified by the parameter. The value returned is an integer equal to the number of 1024 byte blocks contained in the file.

The argument must evaluate to a string. The string represents the filename.

```
SIZE(FILE$)
```

The file does not need to be open.

```
SIZE("NAME")

SIZE(TEMP1$ + ".$$$")
```

Some operating systems support wildcard selections for files. For instance, CP/M allows the asterisk (*) and question mark (?) to represent matches with a variety of characters. The asterisk matches any filename or filetype while the question mark matches any one character in the filename or filetype. For example "*.BAS" refers to all files with a filetype of BAS. The SIZE function accepts wildcard specifications when operating systems such as CP/M support this feature.

```
SIZE("*.TMP")

SIZE("CB-80.OV?")
```

If the file contains no data or if the file does not exist, SIZE returns a zero.

9.6.5 The UNLOCK Function

```
UNLOCK(FILE.ID%,REC%)
```

The UNLOCK function performs the opposite action as the LOCK function. The parameters evaluate to a file identification number and a record number. UNLOCK attempts to unlock the selected record. The UNLOCK function returns the value returned by the operating system where an attempt is made to unlock the record. Normally, a zero means that the record was unlocked successfully.

Both arguments must be numeric; if either evaluates to a real value, it is converted to an integer.

If your operating system does not support record locking, no action occurs and a value of zero is returned. A zero is also returned if the record was already unlocked when the UNLOCK function was executed.

End of Section

Section 10
Formatted Output

CB-80 allows output generated by a PRINT statement to be formatted under the control of a Using format string. This form of a PRINT statement is called a PRINT USING statement. It can be used with output and can be directed to a disk file, the console, or the line printer.

Section 10.1 explains Using strings and format field characters. Sections 10.2 through 10.4 give detailed examples for Using format field characters. Section 10.4 recapitulates the use of escape characters and Section 10.5 explains formatted output to files. The LPRINTER and CONSOLE statements, explained in Section 8, control output to the console or line printer.

10.1 Using Strings

A print statement that has the reserved word USING followed by an expression and a semicolon is a PRINT USING statement. (The syntax diagrams in Sections 8 and 9 show this form of the PRINT statement.) The Using string must be a string expression that consists of literal characters, numeric fields, and string fields. The following example shows Using strings:

```
PRINT USING A$; A,B,C

PRINT USING USING.STRING$(3); #1, REC%; A$,B$

PRINT USING "The amount owed is $$#,###,###.##"; BALANCE
```

When the program executes, it evaluates the next expression in the expression list. The Using string is then scanned. Literal characters are output as they are encountered. When a field is located that matches the type of the expression, the expression is output in the format dictated by the format field.

No delimiters, automatic spacing, or other characters are output. At the end of the print statement, a new line is started unless the expression list ends in a comma or semicolon. In the case of a disk file with fixed length records, the record is padded with blanks if necessary prior to outputting the carriage return and line-feed.

If the expression list contains a string expression, there must be at least one string field in the Using string, otherwise an execution error occurs. Likewise, there must be a numeric field in the Using string if there is a numeric expression in the expression

list.

The numeric and string fields consist of combinations of these characters. The backslash acts as an escape character to force the next character to be treated as a literal character instead of a field character. This does not cause a conflict with continuation characters because the compiler treats a backslash character within a string constant as a character in the string. For example:

```
PRINT USING "The part is \# #####"; \
                        MASTER.PART.NUMBER%
```

Table 10-1 lists the format field characters that CB-80 supports.

Table 10-1. Format Field Characters

Field Character	Function
#	digit position in a numeric field
$$	float a dollar sign in a numeric field
**	asterisk fill a numeric field
-	leading or trailing sign in a numeric field
,	place commas every third digit before decimal point in a numeric field
.	decimal point position in a numeric field
^^^^	exponent position in a numeric field
&	variable length string field
/..../	fixed length string field
!	single character string field
\	escape character (treat next character as a literal)

Sections 10.2 through 10.4 give detailed explanations of format field charaters and their functions.

10.2 Numeric Fields

A pound sign (#) indicates one numeric position. For example the following statement:

```
PRINT USING "###"; I%
```

defines a field of three positions in which to print I%. If I% is set equal to 3, then the result is printed as:

```
__3
```

In this example, two blanks and then the numeral three is printed. The value is right justified in the field and filled with leading blanks.

Note: the underscore (_) in this section indicates that a blank is printed in the space.

The following examples list the results that occur with other values of I%. The Using string remains "###".

I%	RESULT
10	_10
999	999
-10	-10
1000	%1000
-999	%-999

The last two examples show numbers that do not fit into the field. In these cases, the overflow is indicated by printing a percent sign (%) followed by the number in the print format that is used with printing without a Using string. Another example of field overflow is shown below.

```
PRINT USING "###"; 10E10
```

The output from this statement is:

```
%1.0E 11
```

One decimal point can appear in a numeric field. The following examples show the use of a decimal point in numeric fields.

VALUE	FIELD	RESULT
10.10	##.##	10.10
100.789	####.##	_100.79
945.673	####.##	_945.67

Note: values are rounded to fit the field to the right of the decimal point.

If no digits exist before the decimal point and there are one or more digit positions in the format string before the decimal point, a leading zero is printed.

VALUE	FIELD	RESULT
0.78	.###	.780
0.78	#.#	0.8
0.999	#	1

If one or more commas appear in the numeric field, the results are printed with commas inserted every third digit before the decimal point. Each comma in the numeric field serves as a digit position specifier and each comma that actually is printed uses one of the available digit positions. The following examples show the use of the comma in numeric fields.

VALUE	FIELD	RESULT
1000.0	#,###.##	1,000.0
100.0	#,###.##	__100.0
7654321	##,###,###	_7,654,321
7654321	#,#######	7,654,321
7654321	#,######	%7654321

The commas do not have to be placed where they occur in the output and only one comma causes all the necessary commas to be printed. However, the number of pound signs and the number of

commas determine the total number of positions available in the field.

Numeric expressions can be printed in an exponential format by appending one or more up-arrows (^) to the end of the numeric format field. The exponent always uses four positions when it is printed. You can use from one to four up-arrows to specify the exponent.

VALUE	FIELD	RESULT
100	##^^^^	10E_01
-7751.21	##.##^	-7.75E_03
.001234	###^^^^	123E-05
0	##^^^	0E_00

Commas are not printed in a numeric field with an exponent. If commas occur in the field, they are treated as pound signs. In the first example below, the numeric field has five positions for the digits. This requires that the number be rounded to five significant digits. Blank characters are placed in any leading field positions that are unused.

VALUE	FIELD	RESULT
123456	#,###^^	12346E_01
234	#,###^^^^	23400E-02

Instead of the blanks, an asterisk (*) can be used as a fill character by placing two asterisks at the beginning of a numeric field, as shown below.

VALUE	FIELD	RESULT
754	**###	**754
-21	**###	**-21
12345	**###	12345

The two asterisks, like pound signs, are counted as two numeric positions. The asterisks are printed in the place of blanks only if blanks normally fill the field.

You cannot use an asterisk fill in fields with an exponent format. A single asterisk is treated as a print character and not as part of a numeric field.

A dollar sign ($) can be printed to the left of the first digit in a numeric field. This allows you to float a dollar sign; you specify this by placing two dollar signs at the beginning of a numeric field.

VALUE	FIELD	RESULT
10.10	$$###.##	__$10.10
1000.00	$$###.##	$1000.00
1000.00	$$#,##.##	$1,000.00
10000.00	$$#,##.##	10,000.00

Blanks fill the field when a floating dollar sign is a part of the numeric field. As with the asterisk fill, the two dollar signs are counted as two numeric positions. The last example above shows that the dollar sign prints only if a position is available.

Floating dollar signs cannot be used in fields with an exponent format. Also, if the numeric expression output into the field is negative, the minus sign (-) is printed in place of the dollar sign.

VALUE	FIELD	RESULT
-10	$$##.##	_-10.00
10	$$##.##	_$10.00

Note: a single dollar sign is treated as a print character and not as part of a numeric field.

Normally, a negative number has the sign floated to the left of the first digit in the number being printed. By placing a minus sign as the first or last character of a Using string, the minus sign can be placed in a fixed position in the field.

VALUE	FIELD	RESULT
-123.456	###.###-	123.456-
-123.456	-###.###	-123.456
-12.345	-###.###	-_12.345
0.3456	##.###-	__.346_
100.0001	-###.##	_100.00

Note: if the sign of the expression is positive, a blank is printed in place of a sign.

10.3 String Fields

There are three types of string fields: single character, variable length, and fixed length. A single character field is specified by an exclamation mark (!). The field prints the first character of a string expression. The following example prints the letter A.

```
PRINT USING "!"; "ABC"
```

Successive exclamation marks print the first letter of successive string expressions. In other words, each exclamation mark is a separate string field.

```
PRINT USING "!! !"; "XY","UV","PQ"
```

The output from the preceding statement is:

```
*
XU_P<NL>
```

This is the same notation used in Section 8. The asterisk (*) marks column 1 and the <NL> indicates that a new line starts.

A single ampersand (&) represents a variable length string field. The ampersand causes the entire string to print without editing.

```
PRINT USING "&"; "THIS IS A STRING"
```

The statement above prints:

```
*
THIS IS A STRING<NL>
```

The next example uses both variable length and single character string fields.

```
PRINT USING "& !. &";"Jim", "Allen","Smith"
```

The preceding statement prints:

```
*
Jim A. Smith
```

The third type of string field is the fixed length field. This field is delimited by slashes (/ /). The size of the field is the number of spaces or characters between the slashes plus two. Each slash is one position in the fixed field and each character between the slashes is also counted in the size of the field.

The string field in the following example consists of three spaces and the two slashes. Thus, the field has a total length of five characters. The left five characters of the string expression are printed.

```
PRINT USING "/   /"; "HI THERE"
```

The outputs from this statement is:

```
*
HI TH<NL>
```

You can place any characters between the slashes. The compiler ignores these characters but you can use them to document the use or size of the field. The following examples demonstrate this:

```
PRINT USING "/ NAME /"; NAME$

PRINT USING "/...5...9/";A$ + B$
```

If the string expression evaluates to a string shorter than a fixed length field, the expression is left justified in the field. Blanks are inserted to fill the field on the right.

```
PRINT USING "/...5...9/";"XYZ"
```

The preceding statement outputs:

```
*
XYZ_______<NL>
```

Both string and numeric fields can be mixed in a Using string.

```
PRINT USING "#.# XYZ &"; 7.2, "ABC"
```

The output from this statement is shown below:

```
*
7.2 XYZ ABC<NL>
```

The characters XYZ and the space before and after them are literal characters. They appear in the output just as they are in the Using string.

A Using string is reused if the compiler reaches the end of the Using string and there are still more expressions from the expression list to be printed. The Using string is reused by wrapping around to the beginning of the string.

```
PRINT USING "!"; "AX","BX","CX"
```

The output from the preceding statement follows:

```
*
ABC<NL>
```

The Using string is reused three times to allow each expression to be printed. In the following example, each field in the Using string is used once and then the first field is used a second time.

```
PRINT USING "## X &"; 5,"HI",6
```

The output from the preceding statement follows:

```
*
_5_X_HI_6_X_<NL>
```

After the three fields are output, a trailing "_X_" is printed. As each expression is printed, including the last expression, any literal characters following the field in the Using string are output. As soon as a string or numeric field is encountered, no more characters are printed. Also, if during execution you reach the end of the Using string, the Using string is not reused just to print literal characters.

```
PRINT USING "THIS IS A NUMBER ## TO PRINT"; 99
```

The output from this statement is:

```
THIS IS A NUMBER 99 TO PRINT<NL>
```

It is possible for characters in a string or numeric field to be treated as literal characters.

```
PRINT USING "&## X &";29
```

The output from this statement is:

```
*
&29_X_<NL>
```

The expression is numeric. Thus, every character in the Using string is treated as a literal character until a numeric field is found. In this case, the ampersand is printed as a literal character. After the last expression has been printed (in this

example there is only one expression), all characters in the Using string are printed as literal characters until the next string or numeric field is found. This results in the "_X_" being printed but not the second ampersand.

The following PRINT USING statements are invalid:

```
PRINT USING "###"; A$     No string field but string
                          expression.

PRINT USING "/  "; B$     Closing slash is missing.

PRINT USING "##" X+Y      Semicolon is missing.
```

10.4 Escape Characters

The backslash (\) serves as an escape character to force the next character to be a literal character. This allows characters such as pound signs (#) and ampersands (&) to be treated as literal characters.

```
PRINT USING "\###"; 10
```

The output from this statement is:

```
*
#10<NL>
```

The backslash causes the first pound sign to be treated as a literal character. An execution error occurs if the backslash is the last character in a Using string.

A backslash can be printed as a literal character by placing two backslashes in the Using string.

```
PRINT USING "\\#";3
```

The preceding statement outputs:

```
*
\3<NL>
```

10.5 Print Using to Files

The PRINT USING statement can also write formatted data to files. The same Using strings, explained throughout this section, can be used with file PRINT statements.

The following statement outputs one record to the selected file. The record is terminated with new line characters.

```
PRINT USING "&"; #1; A$
```

When the record is written to the file, quotation marks are not placed around string data and fields are not delimited by commas. The following statement shows how to output formulated data to a file using random access.

```
PRINT USING A$+B$; #F1%,REC%; X,Y,Z
```

If output is sent to a fixed file, the record is padded by blanks to ensure that it is the proper length.

End of Section

Section 11
Compiler Operation

This section describes how to use the CB-80 compiler to compile source programs and explains the workspace requirements of the compiler. Section 11.2 describes the toggles that modify compiler operation.

11.1 Compiling a Program

The following command starts the CB-80 compiler:

```
CB80 TEST
```

This command compiles TEST, generates a relocatable object file, and lists the program on the console. The listing provides a line number, the relative address of the code generated by the line, and the actual source line. TEST is the name of the source program that has a default filetype of BAS.

You can override the default filetype of BAS by typing a complete file specification.

```
CB80 TEST.PRl
```

The command above compiles the program TEST.PRl. The source file cannot have a filetype of REL.

The CB-80 compiler includes three overlays:

```
CB80.OVl
CB80.OV2
CB80.OV3
```

All of the overlays must be on the same logical device as the executable module: CB80.COM. When using CP/M or MP/M, the module CB80.COM and all the overlays must be on the logged-in disk. The source file can be on any logical disk device. For example:

```
CB80 D:TEST
```

compiles the program TEST.BAS from drive D:.

The compiler creates work files with a filetype of TMP on the same device as the source file unless a drive is specified by a compiler toggle described in Section 11.2. CB-80 uses the following temporary files:

```
PA.TMP
```

```
QCODE.TMP

DATA.TMP
```

If any files with these names exist on the work file disk when CB-80 starts, they are deleted. After the compilation is complete, CB-80 deletes any temporary files that are created.

In addition, CB-80 creates a file with the same name as the source file and filetype REL on the same device as the source file. If the source program contains errors, CB-80 does not create a relocatable REL file.

The size of the TMP files varies from program to program but the amount of temporary space required is approximately the same amount as the source files being compiled. The REL file is also about the same size as the source file.

On systems with limited disk space, you might have to break the program into modules and compile each module separately.

11.2 Command Line Directives

The command line that invokes the compiler can pass information to the compiler by using command line directives. The directives are alphabetic characters enclosed in square brackets ([]).

```
CB80 TEST [B]
```

The command above compiles TEST.BAS with the B toggle in effect. The source filename automatically terminates when the compiler encounters a left square bracket. The toggles can be either lower- or upper-case letters. The following commands have an identical effect as the one above:

```
CB80 TEST[B]
CB80 TEST[b]
```

In all cases, the source file specification is TEST.BAS.

If the source file cannot be located, an error message appears, and CB-80 returns control to the operating system. The same message appears if a %INCLUDE directive cannot find a source file.

A message appears and compilation terminates when other file system or memory space errors occur. Appendix A lists these messages.

CB-80 supports the following toggles:

Table 11-1. Toggles

Toggle	Action
B	Suppress listing of the source file on the console.
C	Change the default INCLUDE file disk.
I	Interlist the generated code with the source file.
L	Set the page length for printed listings.
N	Generate code for line numbers.
O	Suppress the generation of the object (REL) file.
P	List the source file on the printer.
R	Change the disk that the REL file is written to.
S	Include symbol name information in the REL file.
T	List the symbol table following the source listing.
U	Generate errors for undeclared variables.
W	Set the page width for printed listings.
X	Change the disk used for the work files.

The B toggle suppresses all listing. Only the statistical data concerning the size of code and data areas lists on the console. If CB-80 detects errors, the error and the source line containing the error are listed.

The B toggle overrides other toggles that result in compiler output. The B toggle starts the program with a %NOLIST compiler directive. The %LIST directive overrides the B toggle.

```
CB80 TEST [B]
```

If an %INCLUDE directive specifies a filename with no disk specified, the file is included from the same drive as the source file. The C toggle can override this assignment. The C toggle

changes the default logical drive for INCLUDE files. For example the following command gets INCLUDE files from drive D:

```
CB80 TEST [c(d)]
```

The required drive must be enclosed in parentheses. If the file specification in the %INCLUDE directive specifies a drive, then the C toggle has no effect.

This toggle allows program development to be independent of your particular configuration of the hardware.

The I toggle interlists the generated code with the source statements. The generated code uses standard 8080 mnemonics.

The L toggle changes the page length. The new length must follow the L and be enclosed in parentheses. The length can be any unsigned integer constant.

```
CB80 TEST [L(40)]
```

Initially the page length is set to 66.

The N toggle generates code that saves the current line number for each physical line in the source program. This allows the ERRL function to return the line number when an error occurs.

The O toggle suppresses the generation of the relocatable object (REL) file. This somewhat reduces the time to compile a program. The REL file is not created if the compiler detects errors.

The P toggle includes the listing to the printer. Each page has a heading with the page number and the source filename. A form-feed character is printed prior to printing the first page.

The R toggle selects a drive on which to place the REL file.

The S toggle includes information on all program variables and line numbers in the relocatable object (REL) file. The link editor can use this information to create a "SYM" file. The SYM file can be used with Digital Research's symbolic debugging program, SID™, to aid in debugging a program.

The T toggle lists the symbol table following the source file listing.

The U toggle generates an error if a variable name does not appear in an INTEGER, REAL, or STRING declaration. This toggle locates misspelled identifiers and improves documentation of a program by forcing all variables to be declared.

The W toggle changes the width of output to the printer. The width is initially set to 80 columns. The new width must follow the W and be enclosed in parentheses. The width can be any unsigned

integer constant.

```
CB80 TEST [W(72)]
```

The X toggle selects a drive for work files. If there is no X toggle specified, the work files are placed on the same drive as the source file. The required drive must be enclosed in parentheses. It can be either an upper-case or lower-case letter.

```
CB80 TEST [X(B)]
```

You can specify multiple toggles in the command line. For example, the following command line interlists the generated code with the source statements, lists the source file at the printer, sets the page width to 72 columns, and sets the page length to 40 lines.

```
CB80 TEST [IPW(72)L(140)]
```

Toggles are processed left to right. If you repeat a toggle with a conflicting parameter, the last toggle encountered prevails.

End of Section

Section 12
LK-80

LK-80 is a linkage editor that combines relocatable object (REL) modules into an executable file and optional overlay files. LK-80 is designed for use with the CB-80 compiler. When used with a language such as CB-80, LK-80 produces a composite program by combining the language's default library with the REL modules.

LK-80 can link any program that occupies less than 64K bytes of memory unless the length of symbols exhausts the space reserved for the symbol table. It can also link modules created by a relocatable assembler such as RMAC.

This section describes version 1 of LK-80 that operates with Digital Research's CP/M or MP/M operating systems. The CB-80 Licensing Guide explains how to use CBCK to verify that your copy of LK-80 is correct and has not been altered due to disk copy or hardware or software failure.

12.1 Operation of LK-80

The general form of an LK-80 command line is shown below.

```
LK80 [<fn>=]<fn.ft>{,<fn.ft>} { ([<fn>=]<fn.ft>{,<fn.ft>}) }
```

The brackets ([]) denote optional portions of the command. The braces ({}) indicate that the enclosed section can be repeated zero or more times. The symbol "fn" indicates a filename without a filetype. The symbol "fn.ft" represents a filename with an optional filetype and optional command toggles.

The remainder of this section details each option of the LK-80 command line.

12.2 Linking Modules

The following command starts LK-80:

```
LK80 TEST
```

The command above links the REL module "TEST.REL" producing an executable file "TEST.COM". In addition, a symbol location (SYM) file "TEST.SYM" is produced. The SYM file can be used with Digital Research's symbolic debugging program SID™. (SID is sold separately from CB-80.)

Section 12
LK-80

LK-80 is a linkage editor that combines relocatable object (REL) modules into an executable file and optional overlay files. LK-80 is designed for use with the CB-80 compiler. When used with a language such as CB-80, LK-80 produces a composite program by combining the language's default library with the REL modules.

LK-80 can link any program that occupies less than 64K bytes of memory unless the length of symbols exhausts the space reserved for the symbol table. It can also link modules created by a relocatable assembler such as RMAC.

This section describes version 1 of LK-80 that operates with Digital Research's CP/M or MP/M operating systems. The CB-80 Licensing Guide explains how to use CBCK to verify that your copy of LK-80 is correct and has not been altered due to disk copy or hardware or software failure.

12.1 Operation of LK-80

The general form of an LK-80 command line is shown below.

```
LK80 [<fn>=]<fn.ft>{,<fn.ft>} { ([<fn>=]<fn.ft>{,<fn.ft>}) }
```

The brackets ([]) denote optional portions of the command. The braces ({}) indicate that the enclosed section can be repeated zero or more times. The symbol "fn" indicates a filename without a filetype. The symbol "fn.ft" represents a filename with an optional filetype and optional command toggles.

The remainder of this section details each option of the LK-80 command line.

12.2 Linking Modules

The following command starts LK-80:

```
LK80 TEST
```

The command above links the REL module "TEST.REL" producing an executable file "TEST.COM". In addition, a symbol location (SYM) file "TEST.SYM" is produced. The SYM file can be used with Digital Research's symbolic debugging program SID™. (SID is sold separately from CB-80.)

When linking CB-80 programs, LK-80 automatically searches the default disk for the CB-80 run-time library "CB80.IRL". Any library modules required by the program being linked are combined into the executable module produced by LK-80. The combination of one or more REL files with a language library forms a composite program.

LK-80 prints information on the display about the module being linked. The next example shows the results of linking a simple program. The CB-80 program below, "TEST.BAS", can be compiled to produce a REL file "TEST.REL":

```
PRINT "THIS IS A TEST PROGRAM"
PRINT "IT IS USED TO DEMO LK-80"
STOP
```

Use the following command to link the module "TEST.REL":

```
LK80 TEST
```

The information that LK-80 prints on the display is shown below:

```
A>LK80
-------------------------------------------------
LK-80 Version 1.3 Serial No. 000-1234 Copyright (c)
1981 Digital Research, Inc.  All rights reserved
-------------------------------------------------

code size:      1173 (0100-1273)
common size:    0000
data size:      0168 (1280-13E7)
symbol table space remaining:   0A4C
```

The amount of memory allocated to code, common data, and local data is shown next. In this example, there is no common data. All the values are hexadecimal numbers.

The amount of symbol table space remaining provides an indication of the number of additional symbols that can be added to the modules being linked without running out of symbol table space.

Normally LK-80 produces a COM file with the same name as the REL file. For example, linking the example above results in an executable file "TEST.COM" on the default drive. You can override this naming convention.

```
LK80 PAYROLL=B:PAY
```

The command above links the module "PAY.REL" from drive B but creates an executable file "PAYROLL.COM" on the default drive.

The following command produces the same executable file as the previous example but the file "PAYROLL.COM" is placed on the B drive instead of the default drive.

```
LK80 B:PAYROLL=B:PAY
```

The names of the modules being linked can have filetypes as shown below.

```
LK80 A.REL,B.C
```

LK-80 assumes that the file is a REL file unless the filetype is IRL. This means that, in the preceding example, the module "B.C" is read assuming it is a relocatable object module. The filetype "C" is not ignored, but the file contents are treated as a REL file. If this file is not a relocatable object file, LK-80 will most likely abort with the error message:

```
"LK80 FAILURE 4."
```

12.3 Linking Multiple REL Files

Multiple REL modules can be combined into one executable file by listing a group of REL modules separated by commas.

The following command links the four REL modules, "A.REL", "B.REL", "C.REL", and "D.REL", to an executable file "A.COM". The first name in the list becomes the name of the COM file.

```
LK80 A,B,C,D
```

LK-80 can link as many as 60 REL files at one time. However, the total length of the command line cannot exceed 128 characters. Thus it might be necessary to rename some REL files to short names when a large number of files are being linked. Alternatively, the command line can be put into a file as discussed below.

The modules are linked in the order they appear in the LK-80 command. If no drive reference is specified, the files are read from the default drive. However, REL files can be linked from any drive.

```
LK80 AP,B:APMENUE,A:APSCN
```

When multiple modules are linked together, the executable filename can be specified in the command line. In the example below, the modules "TEST.REL" and "RTN.REL" are linked together forming an executable module "MYPROG.COM".

```
LK80 MYPROG=TEST,RTN
```

LK-80 can read the command line from a file on the disk. This feature is included because the 128-byte limit on the length of the console is exceeded if many modules are included in the same link.

There is no limit to the length of a command line if it is taken from a disk file.

To have LK-80 read the command line from a disk file, use the following command format:

```
LK80 $ <fn.ft>
```

The dollar sign must be the first non-blank character following the letters "LK80." Furthermore, at least one space must separate the "$" from the file specification (fn.ft) of the file that contains the link command. The command file, which can have any name or type you want, can be created with an editor.

Carriage returns, line-feeds, and tab characters can be placed anywhere in the command file so you can easily read long commands. They are ignored by LK-80 (not treated as delimiters).

In addition, the backslash character "\" causes LK-80 to ignore all subsequent characters up to and including the next line-feed. You can use the backslash to include comments and document large links.

To summarize the command file features of LK-80, suppose that a file named "command.lnk" has been created and has the following contents:

```
\\\  LINK command for LIST.COM
\\\  Last modified on 14 December 1981
\\
<tab> LIST = A <tab> \ Root Module                          <cr,lf>
<tab> (UPD)    <tab> \ First Overlay (Update file)          <cr,lf>
<tab> (A1,B1)  <tab> \ Second Overlay, Currently largest    <cr,lf>
```

Executing the command:

```
LK80 $ command.lnk
```

has the same effect as the command:

```
LK80 LIST = A(UPD)(A1,B1)
```

12.4 Producing Overlays

LK-80 produces overlay files that a CB-80 CHAIN statement can load and execute.

LK-80 produces overlay (OVL) files that preserve variables in COMMON including any dynamically created data such as arrays or strings. To place a REL module in an overlay, enclose the name of the REL file in parentheses.

```
LK80 A(B)
```

When the preceding command is executed, LK-80 produces two files:

```
A.COM
B.OVL
```

The COM file is the root. The file "B.OVL" is an overlay file that can be loaded only by a CHAIN statement contained in the root "A.COM".

Chaining to an overlay differs from the conventional concept of loading overlays. When the root chains to an overlay, the overlay replaces the root. Likewise when the overlay chains to another overlay or back to the root, the new overlay replaces the currently executing overlay.

CB-80 ensures that all library routines are contained in the root. Chaining preserves the libraries the overlay files use. This reduces the size of overlays and decreases the time required to load an overlay file.

An overlay file returns to the root that loaded it by chaining back to the COM file. The overlay can load another OVL if the second overlay was also linked with the same root. The following example produces a root "A.COM" and two overlays "B.COM" and "C.COM".

```
LK80 A(B)(C)
```

LK-80 can create up to 60 overlays. However the total number of REL modules linked cannot exceed 60. A particular overlay can contain multiple REL files.

```
LK80 A(B)(C,D,E)(F)(G)
```

The name of each overlay and the name of the root can be specified in the command line.

```
LK80 A=ROOT(B=OV1)(C=OV2)
```

The command line above produces a root "A.COM" and two overlays: "B.OVL" and "C.OVL".

12.5 LK-80 Toggles

To pass information to LK-80, place toggles between square brackets.

```
LK80 TEST[Q]
```

The command above passes the Q toggle to LK-80. The Q toggle causes LK-80 to place symbols beginning with a question mark into the SYM

file. The Q toggle adds about 100 symbols to the SYM file. If the Q toggle is not specified, the SYM file contains only the symbols defined in the programs being linked. Language developers use a symbol beginning with a question mark for library names.

12.6 LK-80 Error Messages

LK-80 prints a phrase on the display describing the error when an error is detected. Control then returns to CP/M if the error is a fatal error.

The following table describes the error messages that can occur.

Table 12-1. LK-80 Error Messages

Message	Meaning
Unresolved external: <symbol name>	The symbol name is defined as an external symbol but is never defined as a public symbol.
Out of Directory Space	LK-80 ran out of directory space while writing the root or overlay file.
Disk Full	LK-80 ran out of disk space while writing the root or overlay file.
Multiple Definition: <symbol name>	The symbol name is defined twice.
Too many overlays	More than 60 overlays were specified in the command line.
Too many modules	More than 60 modules were specified in the command line.

Table 12-1. (continued)

Message	Meaning
Symbol table overflow	There is not sufficient memory for the symbol table.
Cannot open source file	A source file specified in the command line cannot be opened.

12.7 Linking With Assembly Language

LK-80 links modules produced by Digital Research's RMAC Relocating Macro Assembler with REL files created by CB-80. The same commands explained at the beginning of this section apply.

An assembly language module that is linked with CB-80 must not contain any initialized data because of the run-time environment CB-80 requires. Any data that must have initial values can be placed in the code segment.

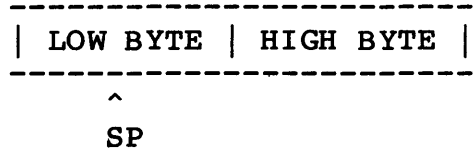
Note: that using assembly language routines makes a program machine dependent.

12.8 Passing Parameters

CB-80 passes all parameters on the 8080 hardware stack. The last entry on the stack contains the return address. Parameters are stored below the return address. When a routine is called, the first parameter is placed on the stack first. Each remaining parameter from left to right is then placed on the stack.

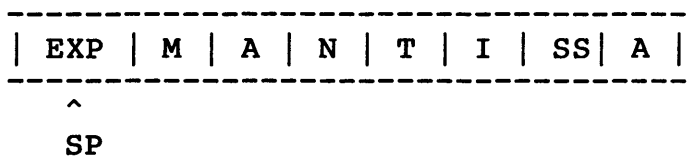
12.8.1 Integer Parameters

Integers are passed on the hardware stack as sixteen bit signed integers. The integers are stored with the low-order byte in the lower memory address.



12.8.2 Real Parameters

Real numbers are passed on the hardware stack as eight byte floating point decimal numbers.



Each of the seven mantissa bytes contain two binary coded decimal digits. The left four bits of each byte in the mantissa contain the most significant digit in that byte. The mantissa is normalized so that the most significant digit is always non-zero.

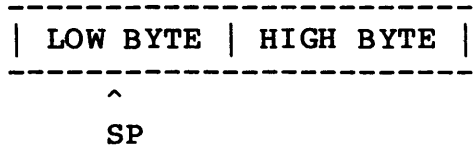
The left most bit of the exponent is the sign of the mantissa. If the bit is a one, the mantissa is negative, and if it is a zero, the mantissa is positive.

The remaining seven bits of the exponent represent the power of ten multiplier to be applied to the mantissa. The actual multiplier used is determined by subtracting 64 from the seven low-order bits of the exponent byte.

A number with a value of zero is represented by setting the exponent byte to 0. The mantissa is ignored. All eight bits of the exponent must be zero for the value to be zero.

12.8.3 String Parameters

Strings are passed by placing a pointer to the actual string on the hardware stack. The pointer is an unsigned sixteen bit integer.



If the value of the pointer is zero, the string is a null string. Otherwise the pointer is the address of the string. The first two bytes of the string contain an allocation bit and a fifteen bit string length. The left most bit of the first byte is the allocation bit.

If the allocation bit is a one, the string must be returned to the CB-80 pool of available storage prior to returning from the assembly language routine, and after all references to characters within the string have occurred. The ?RELS library routine returns

string space to CB-80.

If the 8080 registers H and L contain the pointer to the string passed as the string parameter, the following assembly language statements release a string with its allocation bit set.

```
MOV  A,H        ;IF PTR = 0 THEN
ORA  L          ;   NO RELEASE
RZ
MOV  A,M        ;GET HIGH BYTE OF LNG
ORA  A          ;IS ALLOC BIT = 1?
RP              ;IF NOT NO RELEASE
CALL ?RELS      ;RELEASE THE STRING
RET
```

If the allocation bit is a zero, the characters in the string should not be changed since the calling program still has access to the string. If the allocation bit is 0, the string cannot be released.

12.9 Returning Values to CB-80

An assembly language routine can return integer, real, or string values to CB-80. Prior to returning to CB-80, all parameters passed on the stack must have been removed and the stack pointer adjusted accordingly.

An integer number is returned in registers H and L.

Real numbers are returned by placing a pointer in registers H and L to an eight byte data area containing the real number to be returned.

The returned number must be stored in the format described above. The H and L registers contain the address of the exponent byte.

Strings are returned by placing a pointer to the string in registers H and L. The string must have been allocated by the CB-80 dynamic storage management routines.

The allocation bit of the returned string should be set to one. This ensures that the space is reclaimed when it is no longer required.

12.10 Dynamic Storage Allocation Routines

The CB-80 run-time library provides four routines that allow you to allocate and release memory and to determine the amount of space that is available for allocation.

The ?GETS routine allocates space. The number of bytes of memory required is placed in registers H and L. 32,762 bytes is the maximum amount of space that can be allocated.

?GETS returns a pointer in registers H and L to a contiguous block of memory. There is no restriction on what can be placed in the allocated memory, but the adjacent space at either end of the area cannot be modified. If there is insufficient space, an "OM" error occurs.

The ?RELS routine releases previously allocated memory. The address of the space being released is placed in registers H and L. ?RELS does not return a value.

The ?MFRE routine returns the size of the largest contiguous space that can be currently allocated using the ?GETS routine. The value returned is an unsigned integer; it is placed in registers H and L.

The ?IFRE routine returns the total amount of dynamic space that is currently unallocated. The returned value is an unsigned integer and it is placed in registers H and L.

12.11 Arithmetic Routines

The CB-80 run-time library provides routines for signed integer multiplication and division. The ?IMUL routine multiplies the signed integer in registers D and E by the signed integer in registers H and L. The result is placed in registers H and L.

The ?IDIV routine divides the signed integer in registers D and E by the signed integer in H and L. The result is placed in registers H and L.

End of Section

Appendix A
CB-80 Reserved Words

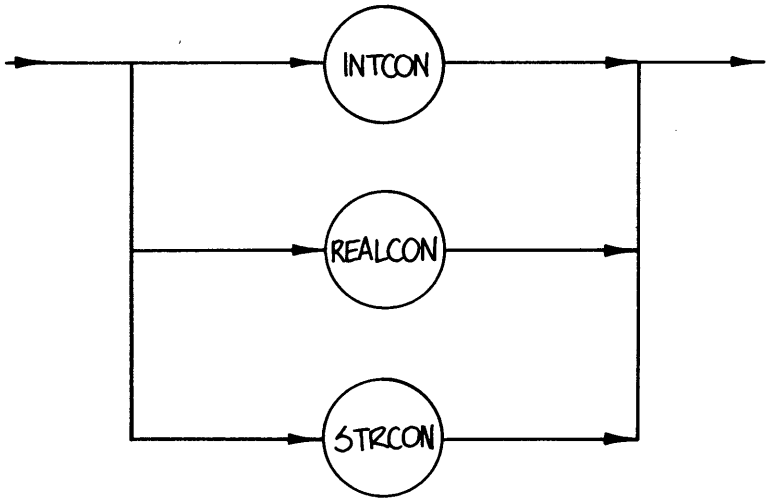
ABS	AND	AS	ASC	ATN
ATTACHP	BUFF	CALL	CHAIN	CHR$
CLOSE	COMMAND$	COMMON	CONCHAR%	CONSOLE
CONSTAT%	COS	CREATE	DATA	DEF
DELETE	DETACH	DIM	ELSE	END
ERR	ERRL	ERROR	EQ	EXP
EXTERNAL	FEND	FLOAT	FOR	FRE
GE	GET	GO	GOSUB	GOTO
GT	IF	INITIALIZE	INKEY	INP
INPUT	INT	INT%	INTEGER	LE
LEFT$	LEN	LET	LINE	LOCK
LOCKED	LOG	LPRINTER	LT	MATCH
MFRE	MID$	MOD	NE	NEXT
NOT	ON	OPEN	OR	OUT
PEEK	POKE	POS	PRINT	PUBLIC
PUT	RANDOMIZE	READ	READONLY	REAL
RECL	RECS	REM	REMARK	RENAME
RESTORE	RETURN	RIGHT$	RND	SADD
SGN	SIN	SIZE	SQR	STEP
STOP	STR$	STRING	SUB	TAB
TAN	THEN	TO	UCASE$	UNLOCK
UNLOCKED	USING	VAL	VARPTR	WEND
WHILE	WIDTH	XOR	%CHAIN	%EJECT
%INCLUDE	%LIST	%NOLIST	%PAGE	

End of Appendix

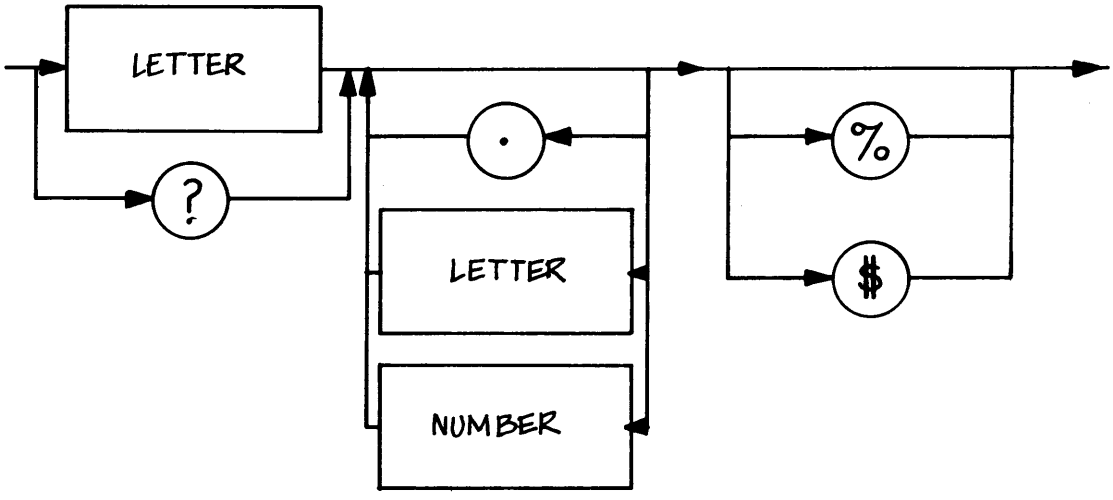
Appendix B
Collected Syntax Diagrams

This appendix contains the syntax diagrams that describe the complete syntax of CB-80.

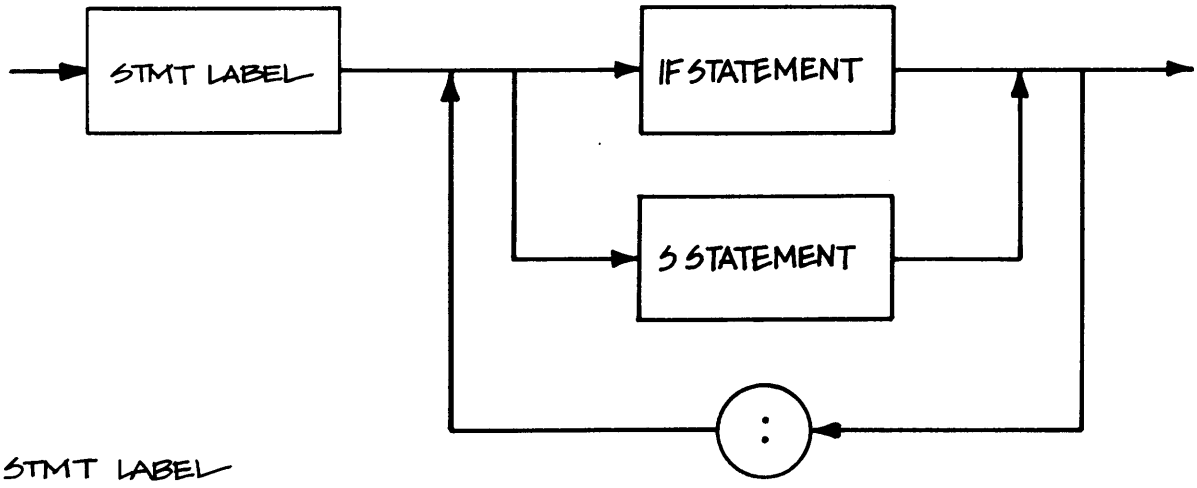
CONSTANT



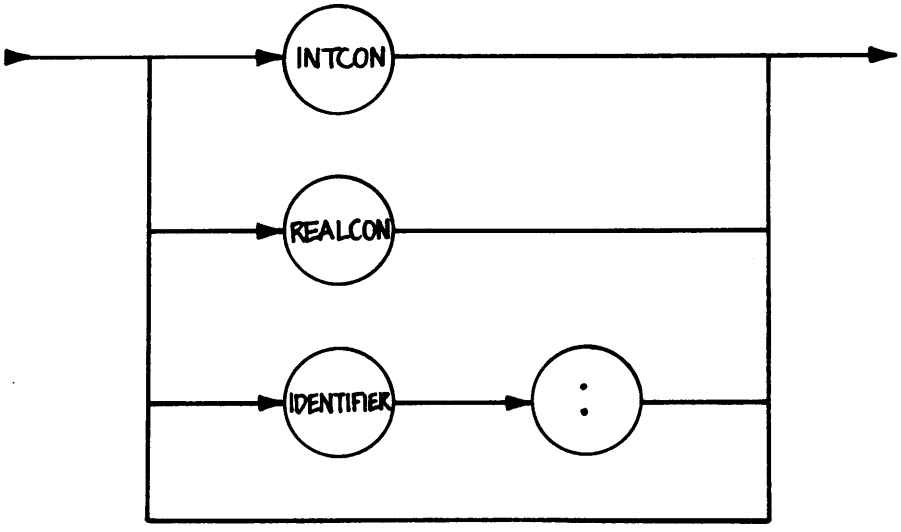
ID



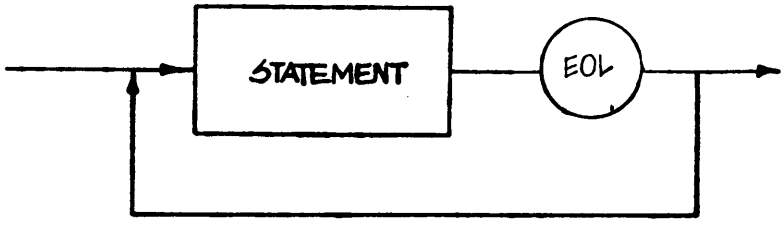
STATEMENT



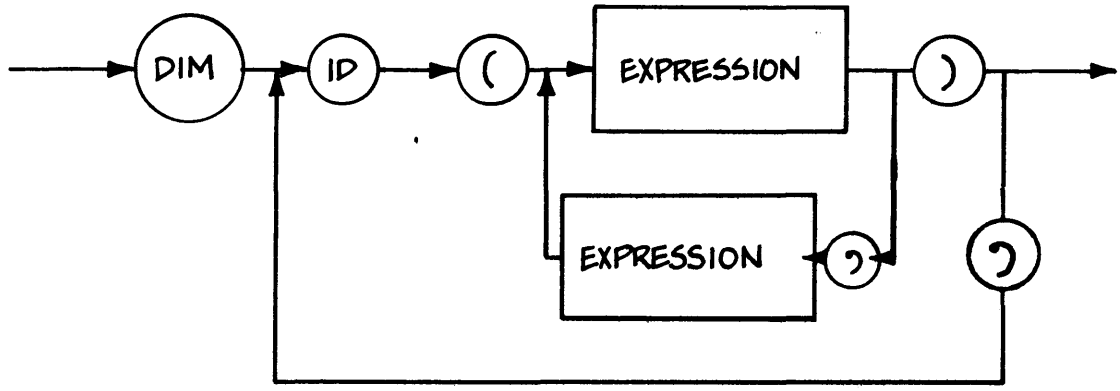
STMT LABEL



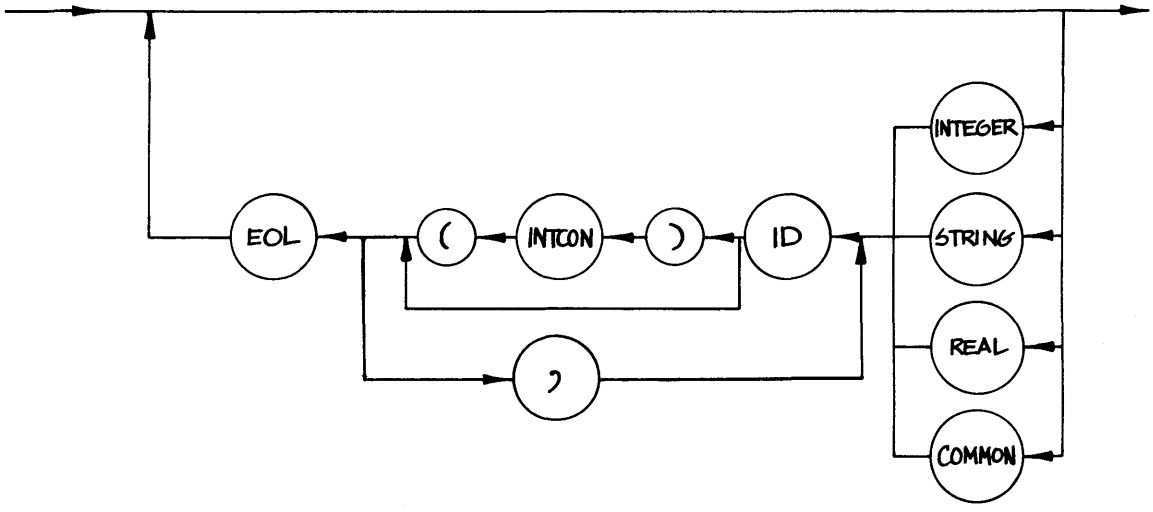
STMT GROUP



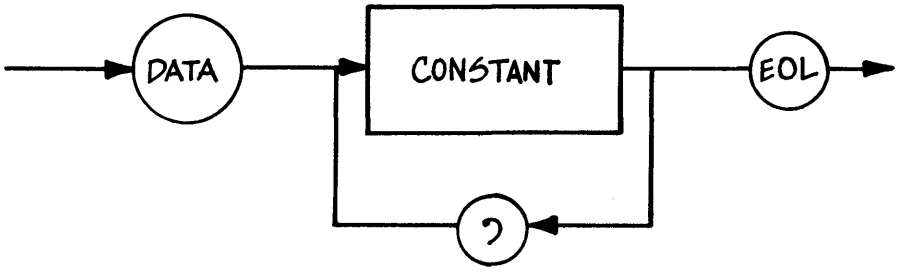
DIM



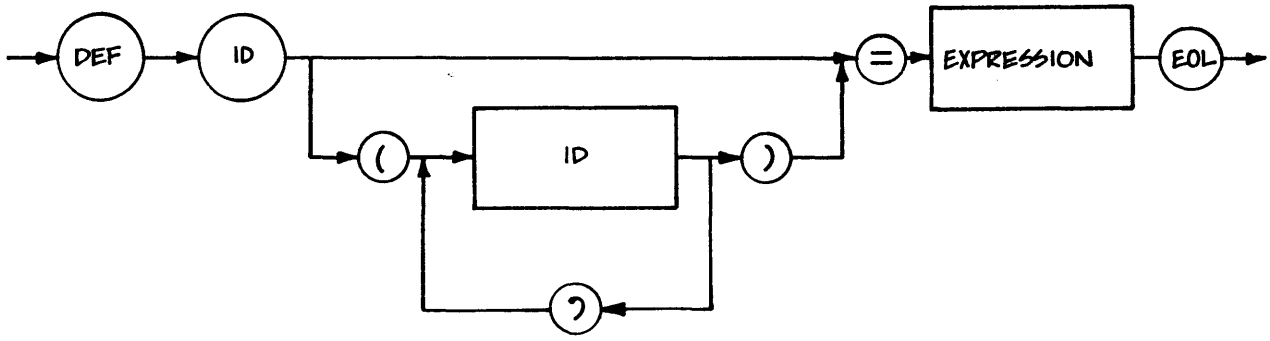
DEC GROUP



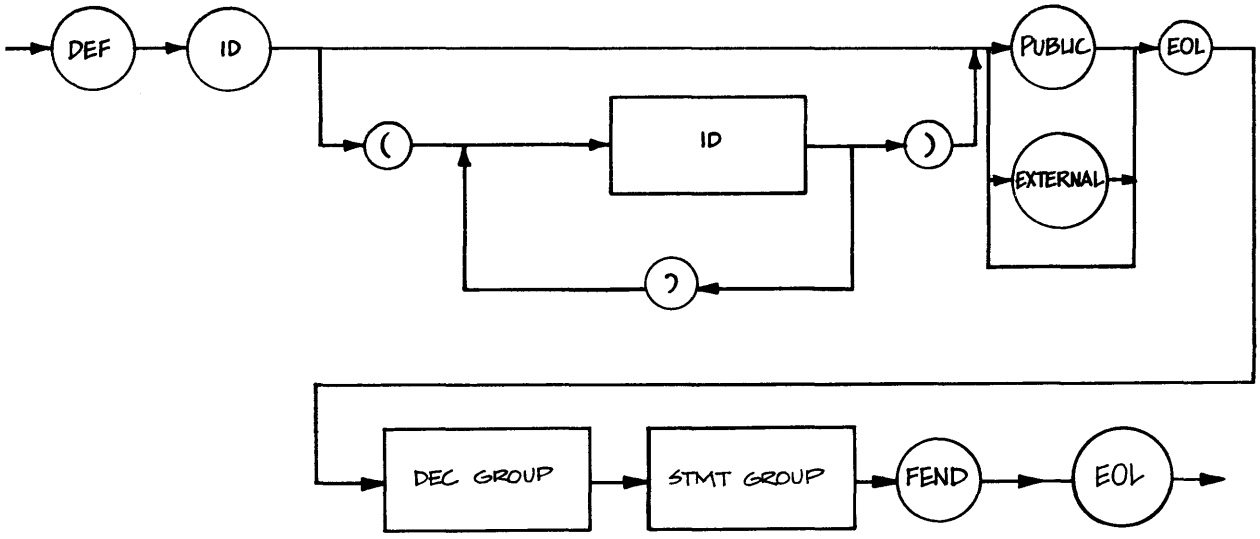
DATA



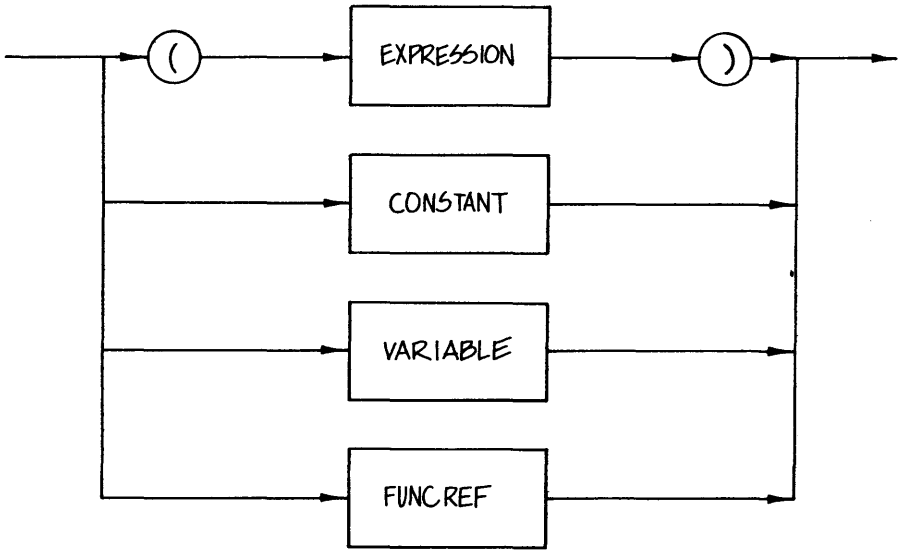
SINGLE LINE FUNCTION



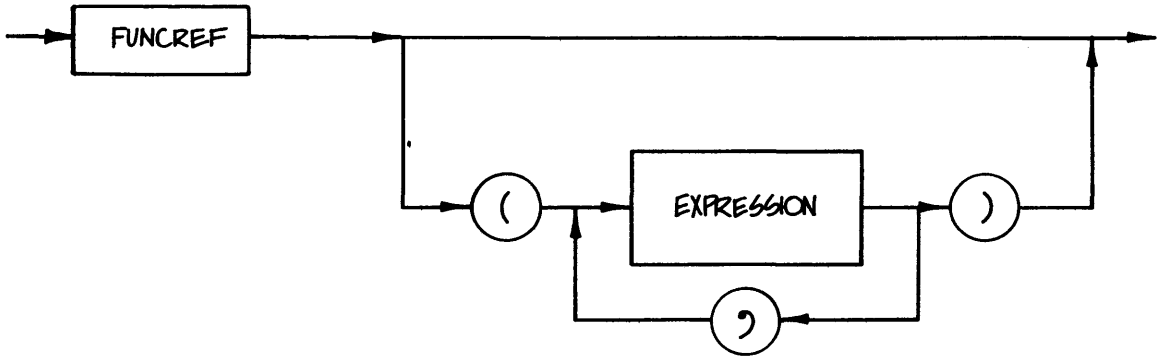
MULTIPLE LINE FUNCTION



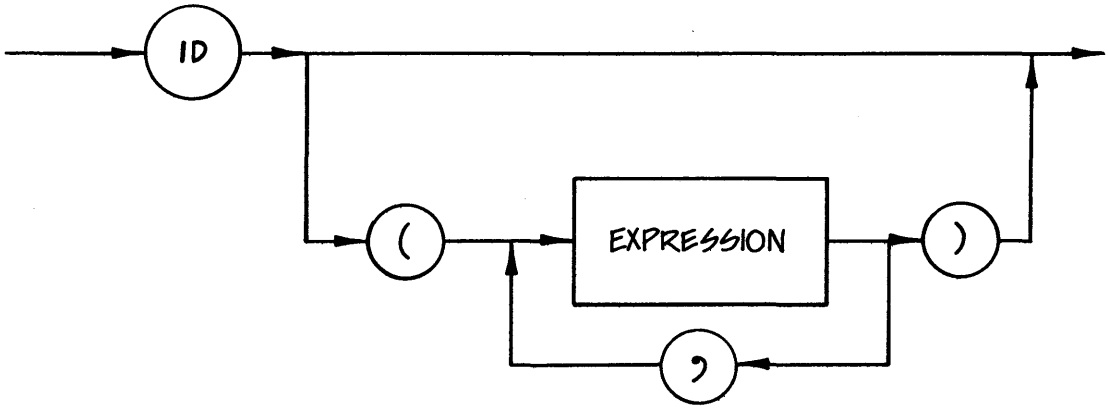
OPERAND



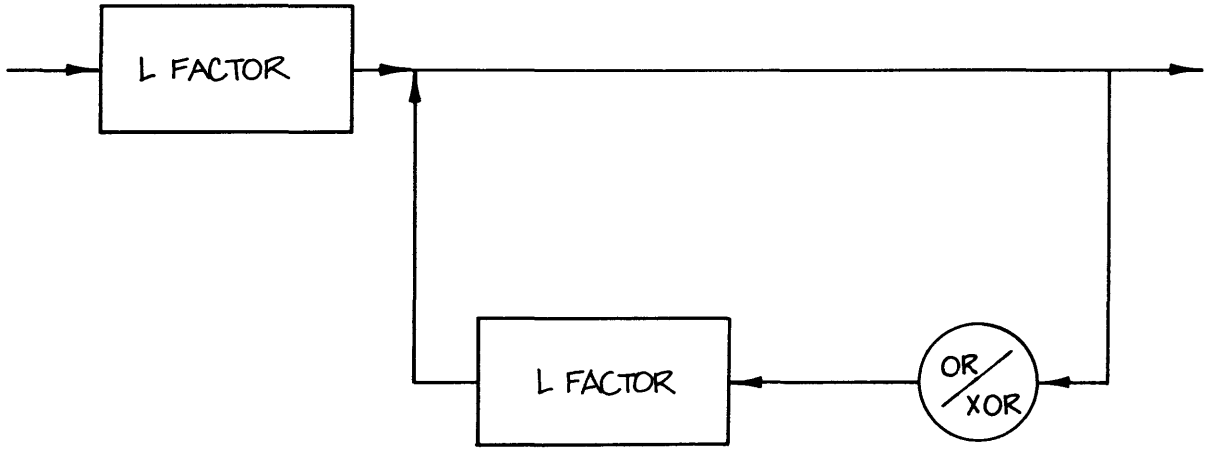
FUNCREF



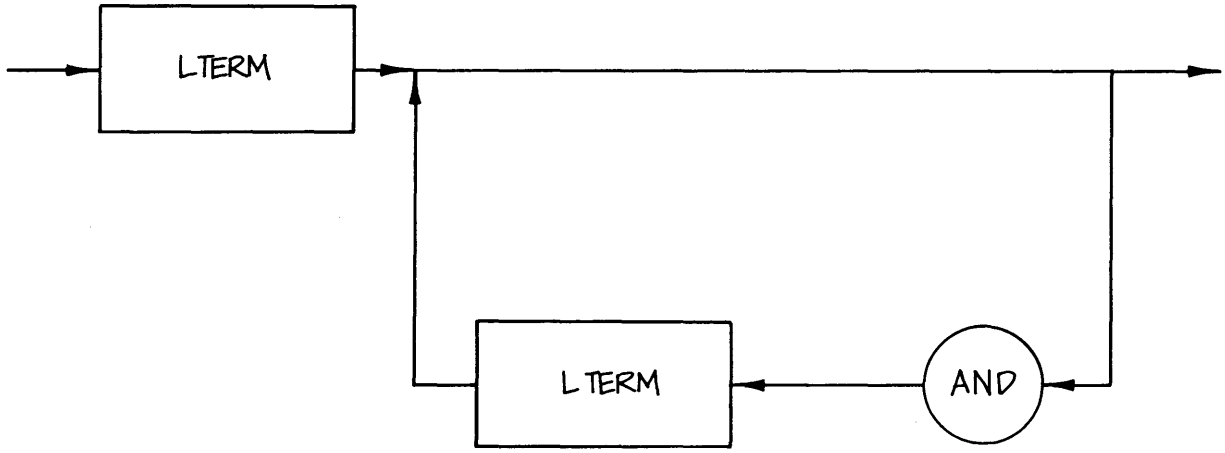
VARIABLE



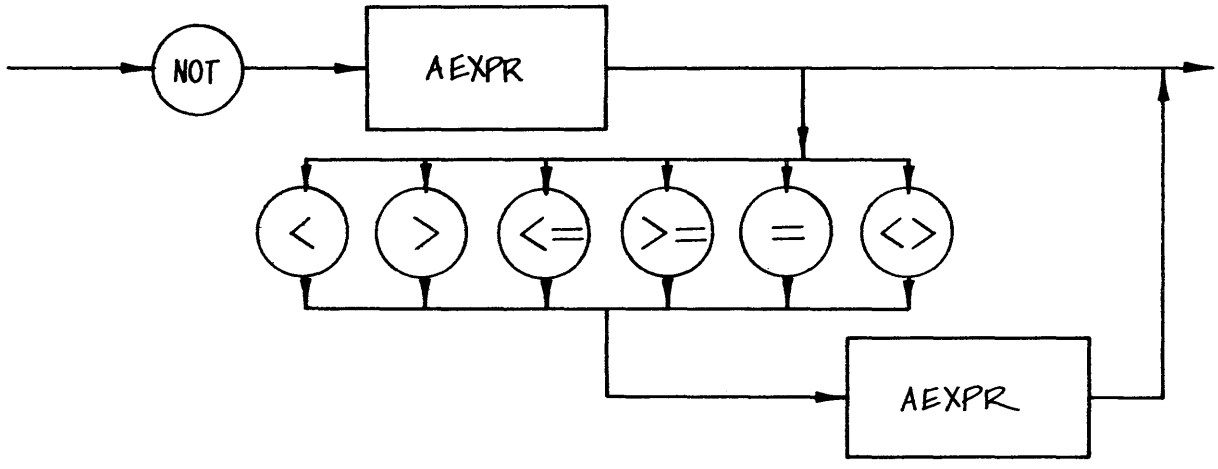
EXPRESSION



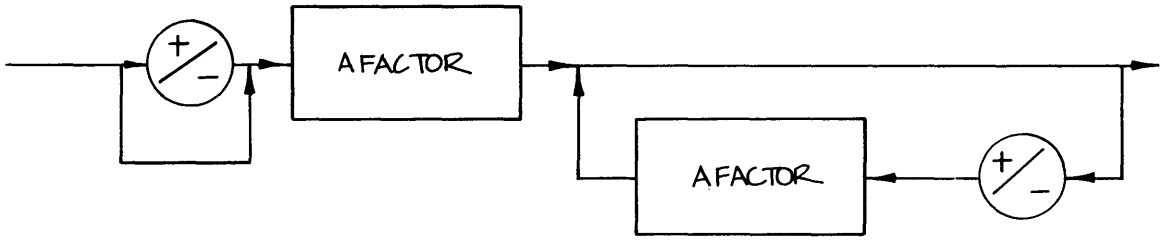
L FACTOR



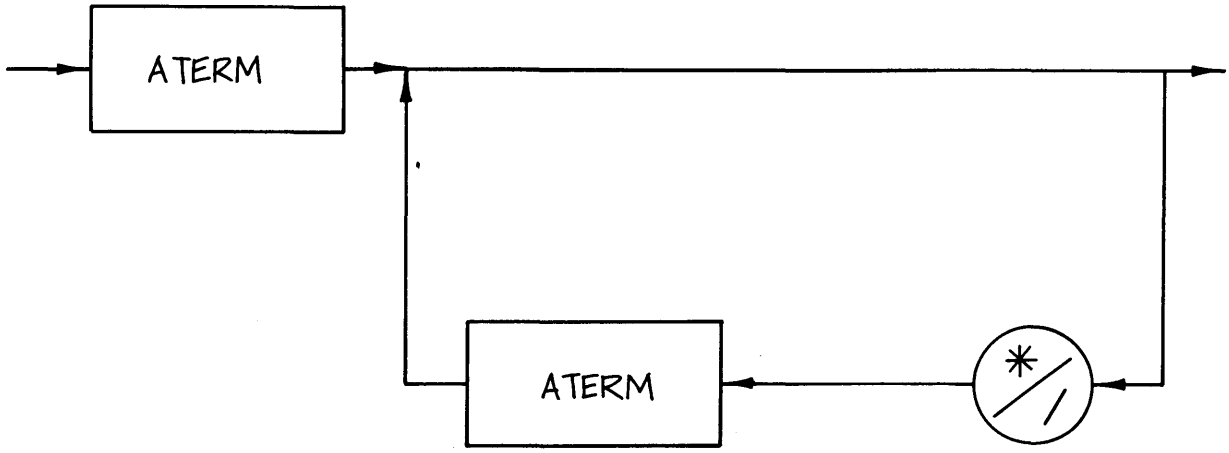
L TERM



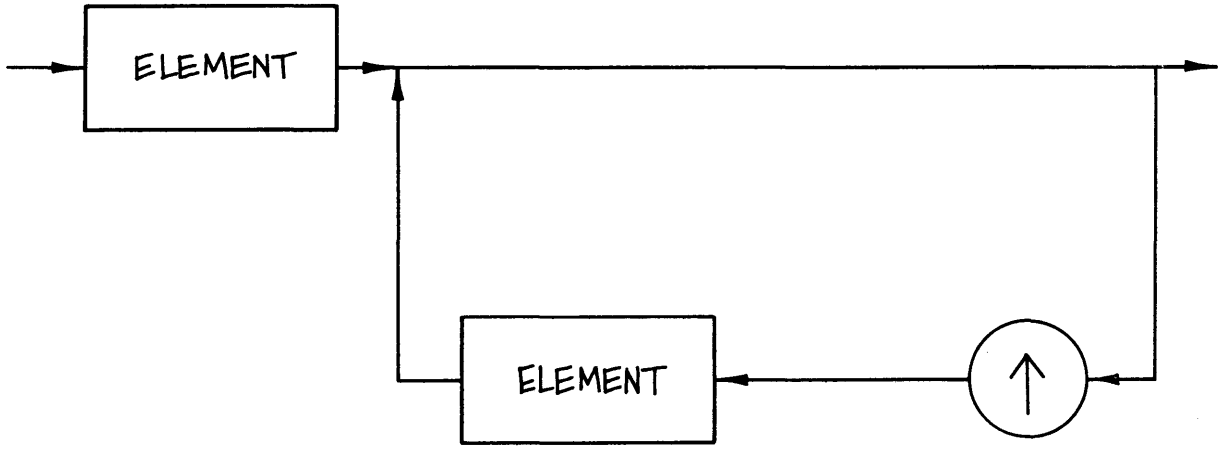
AEXPR



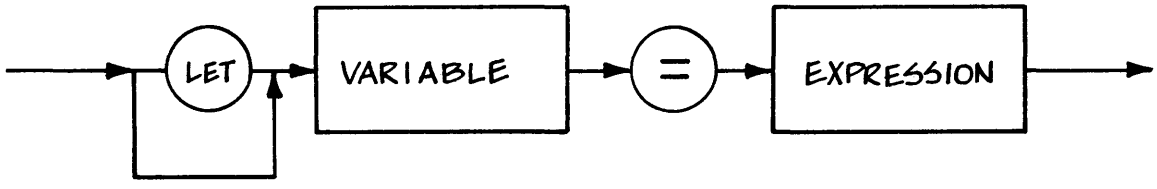
A FACTOR



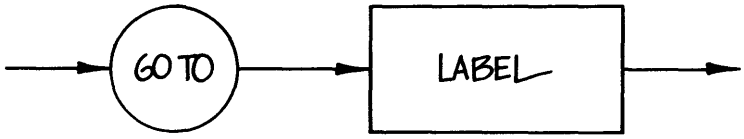
ATERM



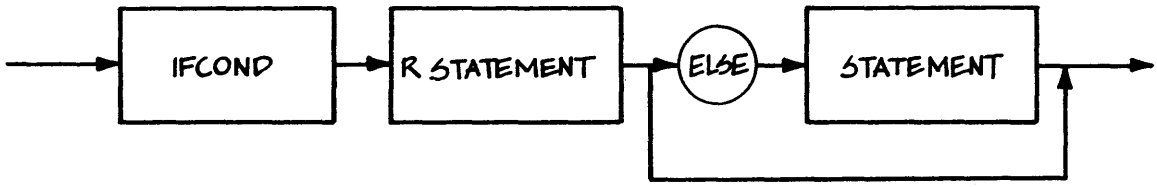
ASSIGNMENT STATEMENT



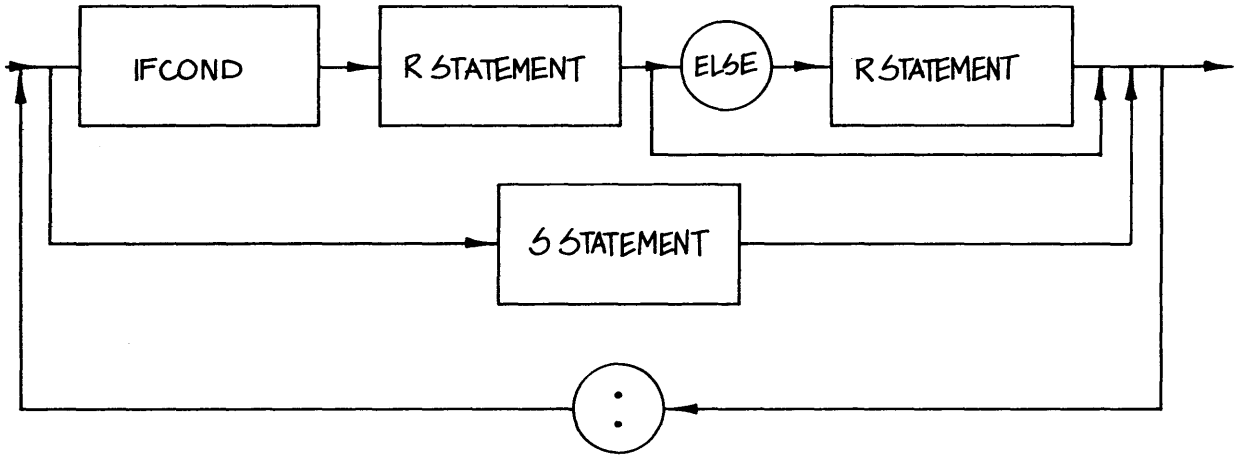
GO TO



IF STATEMENT

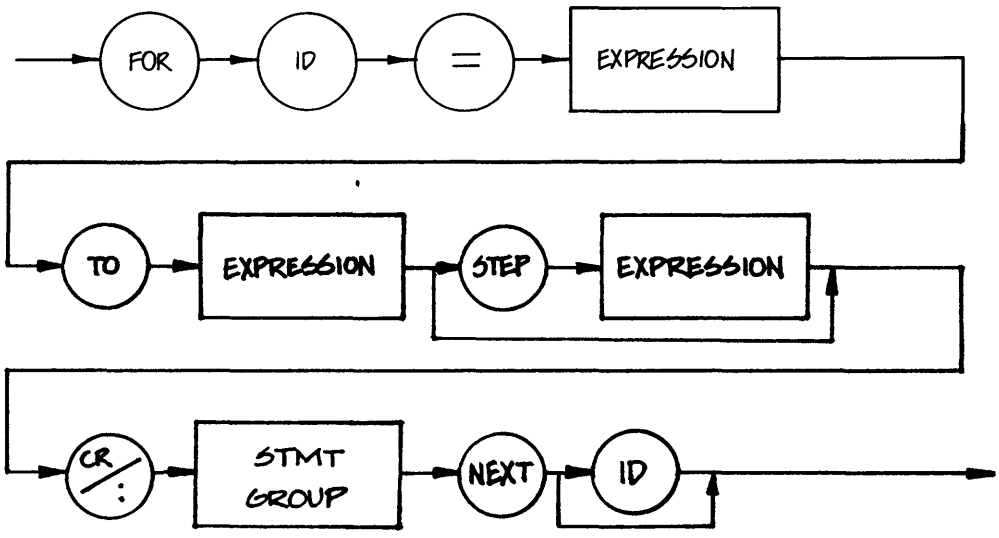


R STATEMENT

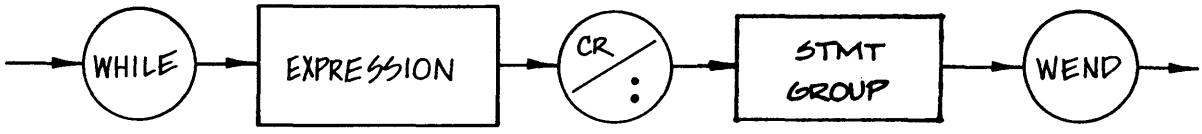


IFCOND

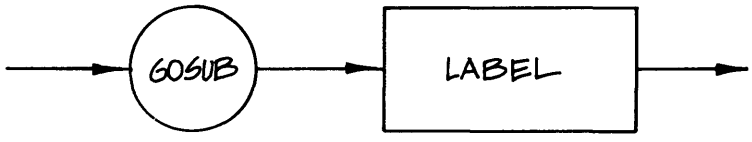
FOR LOOP



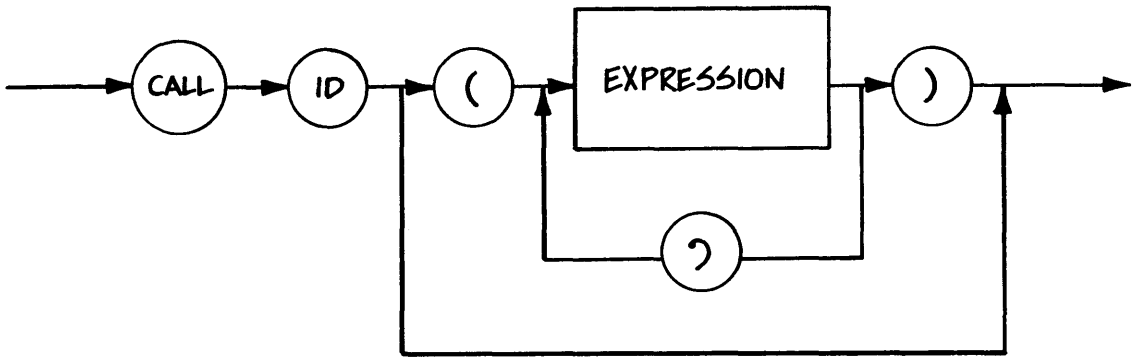
WHILE LOOP



GO SUB

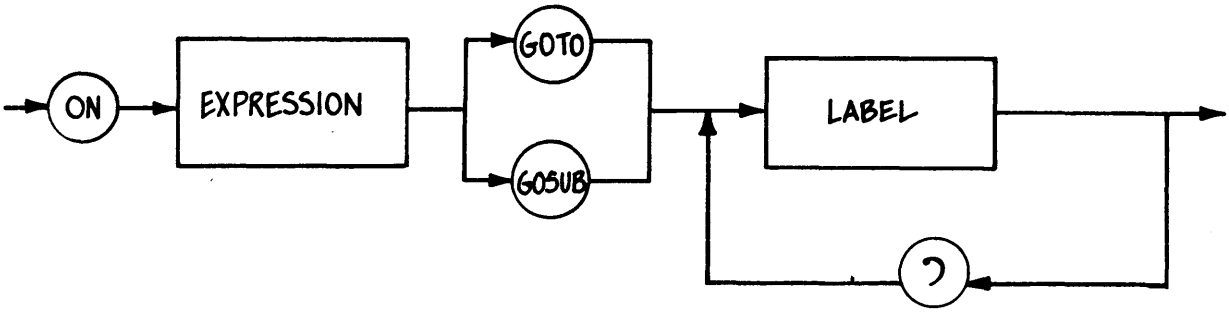


CALL

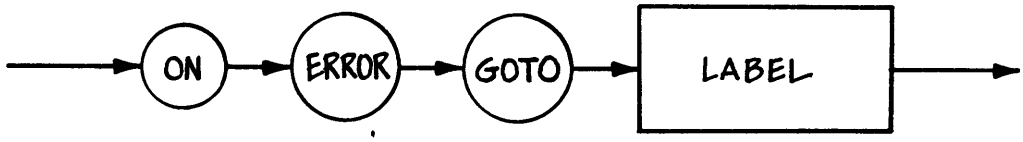


RETURN

ON

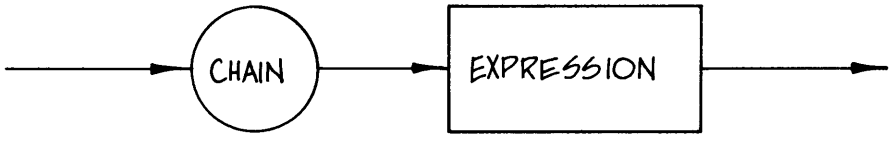


ON ERROR

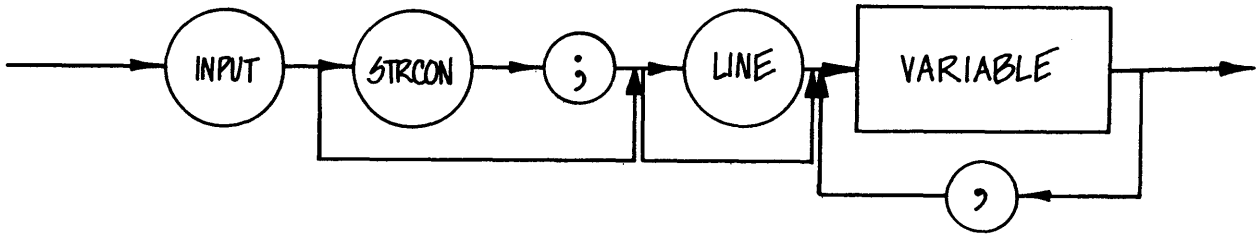


STOP

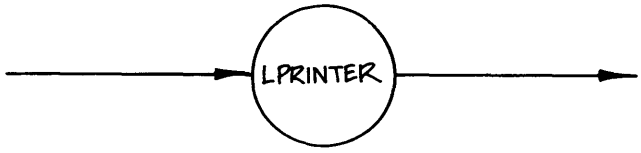
CHAIN



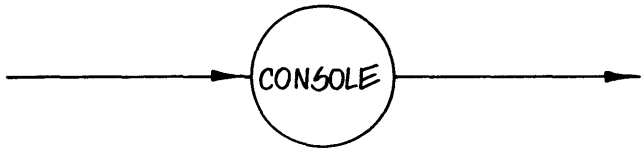
INPUT



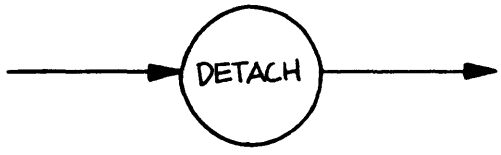
LPRINTER



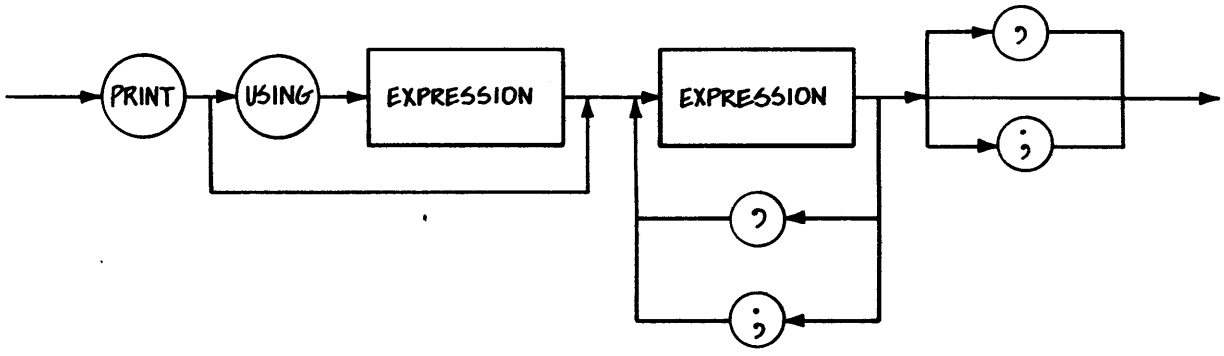
CONSOLE



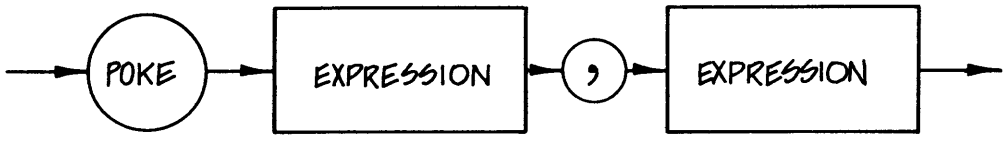
DETACH



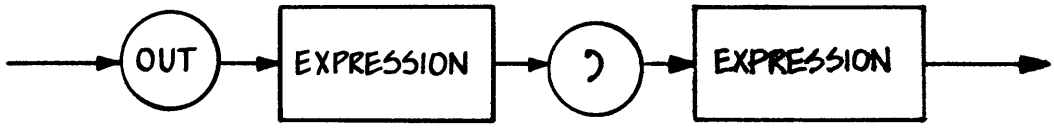
PRINT



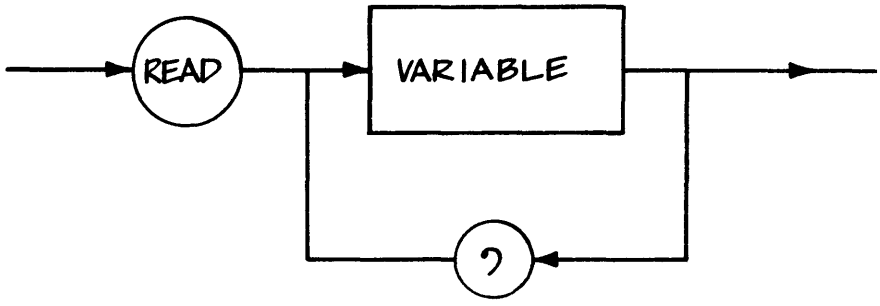
POKE



OUT



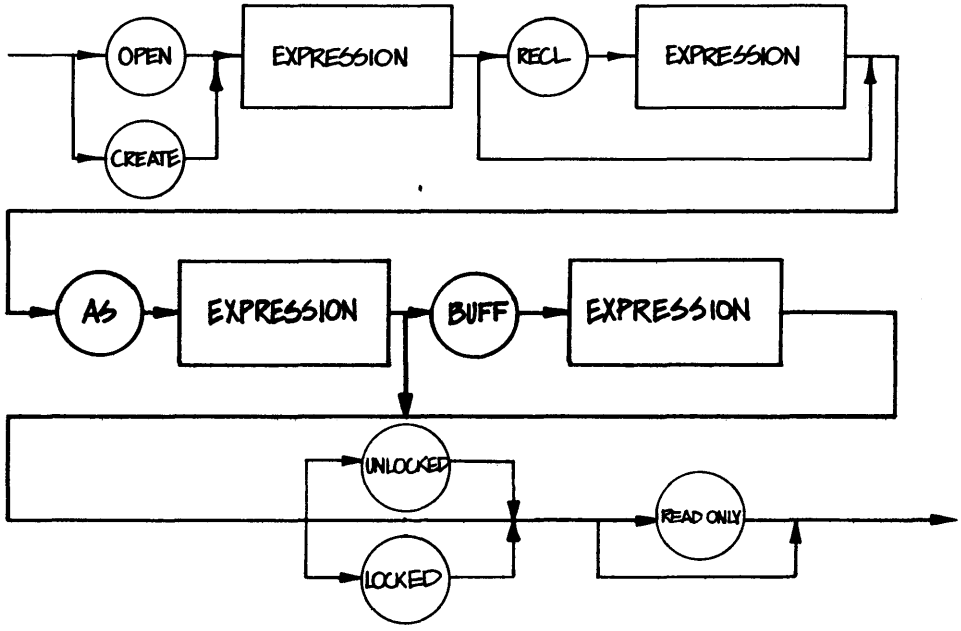
READ



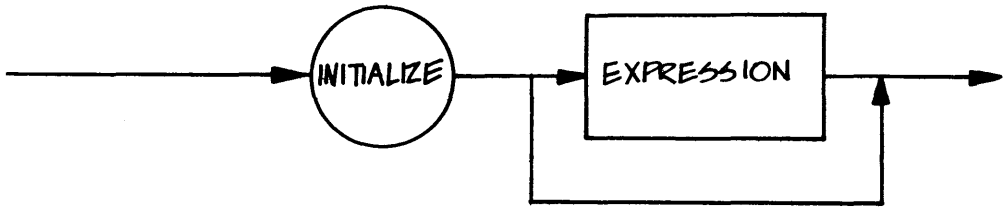
RESTORE

RANDOMIZE

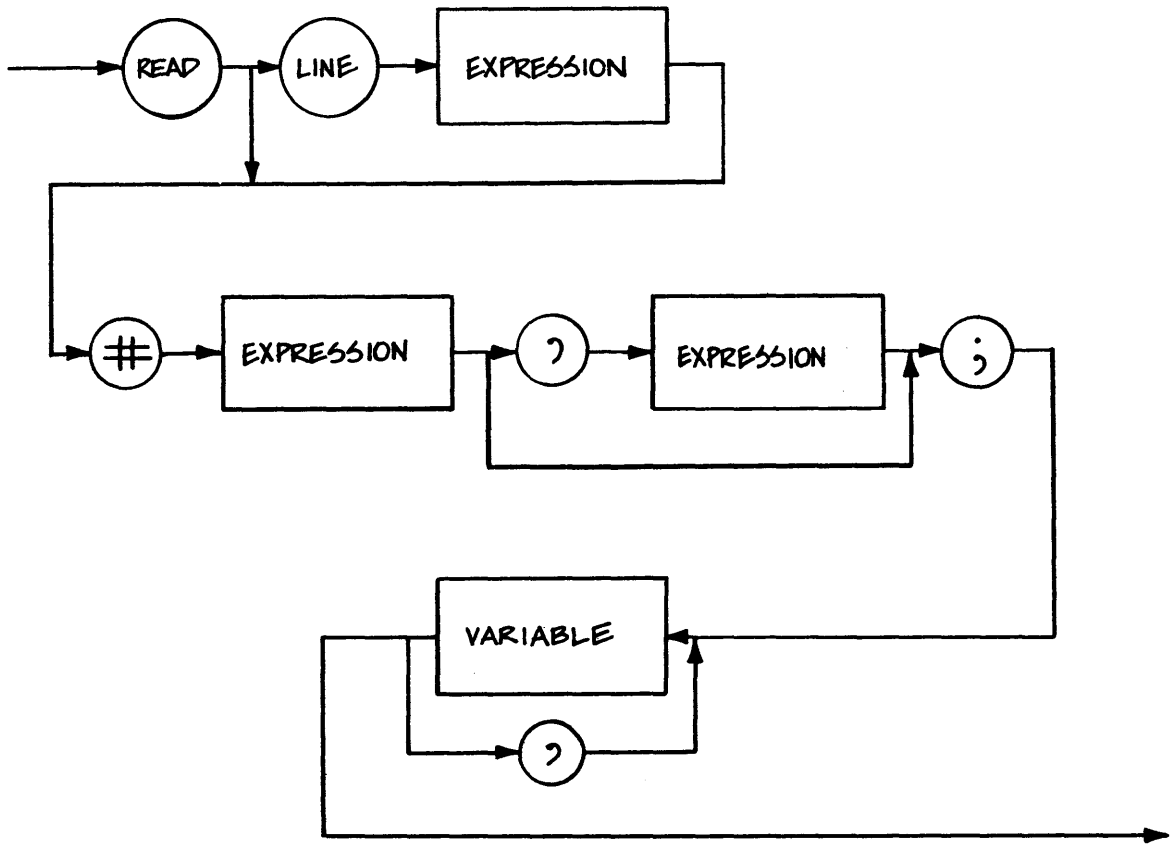
FACCESS



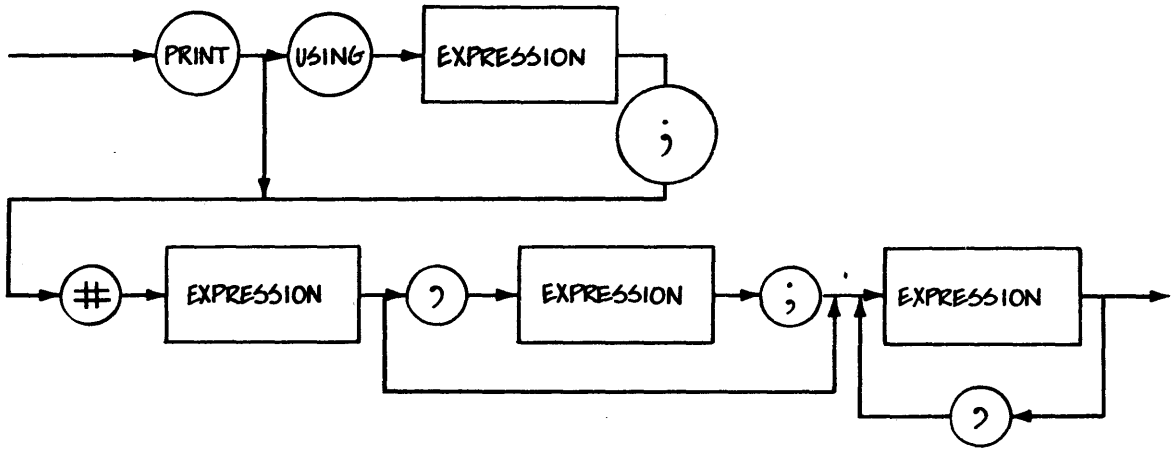
INIT



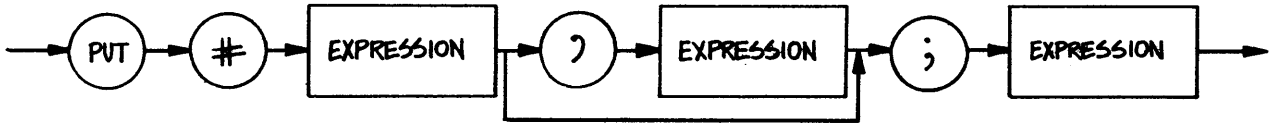
READ FILE



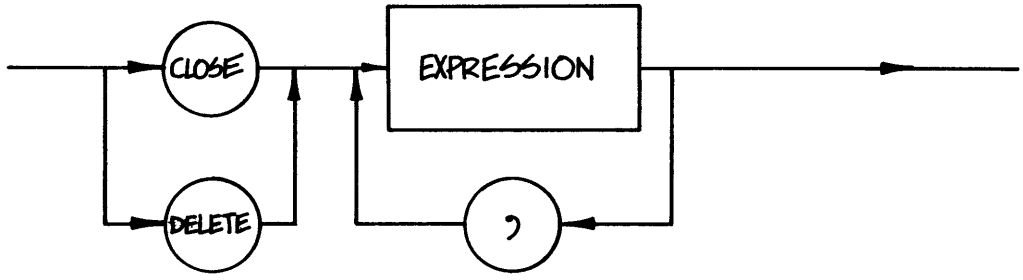
PRINT FILE



PUT



F TERM



IF END

End of Appendix

Appendix C
Compiler Error Messages

The compiler prints the following messages when a file system error or memory space error occurs. In each case, control returns to the operating system.

Table C-1. File System and Memory Space Errors

Error	Meaning
COULD NOT OPEN FILE: <filename>	The filename following the message cannot be located in the file system directory.
INCLUDES NESTED TO DEEP: <filename>	The filename following the message occurred in an %INCLUDE directive that exceeds the allowed nesting of %INCLUDE directives.
SYMBOL TABLE OVERFLOW	The available memory for symbol table space has been exceeded. Break the program into modules or use shorter symbol names.
INVALID FILE NAME: <filename>	The filename is not valid for your operating system.
DISK READ ERROR	The operating system reports a disk read error.
CREATE ERROR: <filename>	The file cannot be created. Normally this means there is no directory space on the disk.

Table C-1. (continued)

Error	Meaning
DISK FULL	The operating system reports that no additional space is available to write temporary or output files.
INVALID COMMAND LINE	The CB-80 command line is incorrect. This message also appears if you did not specify a source file.
CLOSE OR DELETE ERROR	The operating system reports that it cannot close a file. This occurs if diskettes are switched during compilation.

If the compiler detects an internal failure, the following error message appears:

FATAL COMPILER ERROR XXX

where XXX is a three digit number. Please advise Digital Research of the error and the circumstances under which it occurs.

The following error messages indicate a fatal compiler error occurred during compilation of a program. Compilation continues after the error is recorded.

Table C-2. **Compilation Error Messages**

Error	Meaning
1	An invalid character was detected in the source program. The character was ignored.
2	Invalid string constant. The string is too long or contains a carriage return.
3	Invalid numeric constant. An integer constant of zero is assumed.
4	Undefined compiler directive. This source line is ignored.
5	The %INCLUDE directive is missing a filename. This source line is ignored.

Table C-2. (continued)

Error	Meaning
6	Statements found after an END.
7	Not used.
8	A variable was used without being defined and the U toggle was used during compilation.
9	The DEF statement is not terminated by a carriage return. A carriage return was inserted.
10	A right parenthesis is missing from the parameter list. A right parenthesis was inserted.
11	A comma was expected in the parameter list. A comma was inserted.
12	An identifier was expected in the parameter list.
13	The same name is used twice in a parameter list.
14	A DEF statement occurred within a multiple line function. Multiple line functions cannot be nested. The statement was ignored.
15	A variable was expected.
16	The function name was missing following the keyword DEF. The DEF statement was ignored.
17	A function name was used previously. The DEF statement is ignored.
18	A FEND statement was expected. A FEND was inserted.
19	There are too many parameters in a multiple line function.
20	Inconsistent identifier usage. An identifier cannot be used as both a label and a variable.
21	Additional data exists in the source file following an END statement. This is the logical end of the program.

Table C-2. (continued)

Error	Meaning
22	Data statements must begin on a new line. The remainder of this statement was treated as a remark.
23	A reserved word appears in a declaration list. The reserved word was ignored.
24	A function name appears in a declaration within a multiple line function other than the multiple line function that defines this function name.
25	A function call was encountered with the incorrect number of parameters.
26	A left parenthesis was expected. A left parenthesis was inserted.
27	Invalid mixed mode. The type of the expression is not permitted.
28	Unary operator cannot be used with this operand.
29	Function call has improper type of parameter.
30	Invalid symbol follows a variable, constant, or function reference.
31	This symbol cannot occur at this location in an expression. The symbol is ignored.
32	Operator is missing. Multiplication operator inserted.
33	Invalid symbol encountered in an expression. The symbol is ignored.
34	A right parenthesis was expected. A right parenthesis inserted.
35	A subscripted variable is used with the incorrect number of subscripts.
36	An identifier is used as a simple variable with previous usage as a subscripted variable.
37	An identifier is used as a subscripted variable with previous usage as an unsubscripted variable.

Table C-2. (continued)

Error	Meaning
38	A string expression is used as a subscript in an array reference.
39	A constant was expected.
40	Invalid symbol found in declaration list. The symbol is skipped.
41	A carriage return was expected in a declaration statement. A carriage return was inserted.
42	Comma is missing in declaration list. A comma inserted.
43	A common declaration cannot occur in a multiple line function. The statement is ignored.
44	An identifier appears in a declaration twice in the main program or within the same multiple line function.
45	The number of dimensions specified for an array exceeds the maximum number allowed. A value of one was used. This might generate additional errors in the program.
46	Right parenthesis missing in the dimension specification within a declaration. A right parenthesis was inserted.
47	The same identifier is placed in COMMON twice.
48	An invalid subscripted variable reference was encountered in a declaration statement. An integer constant is required. A value of 1 was used.
49	An invalid symbol following a declaration or the symbol in the first statement in the program is invalid. The symbol is ignored.
50	An invalid symbol was encountered at the beginning of a statement or following a label.
51	An equal sign was expected in assignment. An equal sign was inserted.

Table C-2. (continued)

Error	Meaning
52	A name used as a label was previously used at this level as either a label or variable.
53	Unexpected symbol following a simple statement. The symbol was ignored.
54	A statement was not terminated with a carriage return. Text was ignored until the next carriage return.
55	A function name was used in the left part of an assignment statement outside of a multiple line function. Only when the function is being compiled can its name appear on the left of an assignment statement.
56	A predefined function name was used as the left part of an assignment statement.
57	In an IF statement, a THEN was expected. A THEN was inserted.
58	A WEND statement was expected. A WEND was inserted.
59	A carriage return or colon was expected at the end of a WHILE loop header.
60	In a FOR loop header the index is missing. The compiler skipped to end of this statement.
61	In a FOR loop header, a TO was expected. A TO was inserted.
62	An equal sign was missing in a FOR loop header assignment. An equal sign was inserted.
63	Expected carriage return or colon at end of FOR loop header.
64	A NEXT statement was expected. A NEXT was inserted.
65	Not used.
66	The variable that follows NEXT does not match the FOR loop index.

Table C-2. (continued)

Error	Meaning
67	A NEXT statement was encountered without a corresponding FOR loop header.
68	A WEND statement was encountered without a corresponding WHILE loop header.
69	A FEND statement was encountered without a corresponding DEF statement. This error indicates that the end of the source program was detected while within a multiple line function.
70	The PRINT USING string is not of type string.
71	A delimiter is missing in a PRINT statement. A semicolon was inserted.
72	A semicolon was expected in an INPUT prompt. A semicolon was inserted.
73	A delimiter is missing in an INPUT statement. A comma was inserted.
74	A semicolon was expected following a file reference. A semicolon was inserted.
75	The prompt in an INPUT statement was not of type string.
76	In an INPUT LINE statement, the variable following the keyword LINE was not a string variable.
77	In an INPUT statement a comma was expected between variables. A comma was inserted.
78	The keyword AS was missing in an OPEN or CREATE statement. AS was inserted.
79	The filename in an OPEN or CREATE statement was not a string expression.
80	A delimiter is missing in a READ statement. A comma was inserted.
81	In a GOTO, GOSUB or ON statement, a label was expected. This token can be an identifier previously used as a variable.

Table C-2. (continued)

Error	Meaning
82	The label is a GOTO statement is not defined. If the label is used in a function, it must be defined in that function.
83	A delimiter is missing in a file READ statement. A comma was inserted.
84	In a READ LINE statement, the variable following the keyword LINE is not a string variable.
85	The label in an IF END statement is not defined.
86	A pound sign (#) was expected in an IF END statement. A pound sign was inserted.
87	A THEN was expected in an IF END statement. A THEN was inserted.
88	In a PRINT statement, the semicolon is missing following a using string. A semicolon was inserted.
89	In an ON statement, a GOTO or GOSUB was expected. A GOTO was assumed.
90	The index of a FOR loop header is of type string. The index must be an integer or real value.
91	The expression following the keyword TO in a FOR loop header is of type string. The expression must be an integer or real value.
92	The expression following the keyword STEP in a FOR loop header is of type string. The expression must be an integer or real value.
93	A variable in a DIM statement has been defined previously as other than a subscripted variable.
94	An identifier was expected as an array name in a DIM statement. The entire statement was ignored.
95	A left parenthesis was expected in a DIM statement. A left parenthesis was inserted.

Table C-2. (continued)

Error	Meaning
96	A right parenthesis was expected in a DIM statement. A right parenthesis was inserted.
97	The maximum number of dimensions allowed with a subscripted variable was exceeded.
98	A comma was expected in a POKE statement. A comma was inserted.
99	The index of a FOR loop header was not a simple variable.
100	In a CALL statement, a multiple line function name was expected.
101	A file PRINT statement was terminated with a comma or semicolon.
102	A DIM statement is missing for this subscripted variable.
103	Expected a comma in the label list associated with an ON GOTO or ON GOSUB statement. A comma was inserted.
104	Expected a GOTO in an ON ERROR statement. A GOTO was inserted.
105	Expected a comma in a PUT statement. A comma was inserted.
106	The expression in an IF statement was of type string. An integer or real expression is required.
107	The expression in a WHILE loop header was of type string. An integer or real expression is required.
108	In an OPEN or CREATE statement, the filename was missing.
109	In an OPEN or CREATE statement, the expression following the reserved word AS was missing.
110	A multiple line function called itself.
111	A semicolon separates expressions in a file PRINT statement. A comma is substituted for the semicolon.

Table C-2. (continued)

Error	Meaning
112	A file PRINT statement does not have an expression list.
113	A TAB function is used in a file PRINT statement expression list.
114	Not used.
115	A GO not followed by a TO or SUB. GOTO is assumed.
116	An OPEN or CREATE statement specifies both UNLOCKED and LOCKED access control.
117	A CREATE statement uses the READ ONLY access control.

End of Appendix

Appendix D
Execution Error Messages

The following warning message might be printed during execution of a CB-80 program:

```
IMPROPER INPUT - REENTER
```

This message occurs when the fields you enter from the console do not match the fields specified in the INPUT statement. Following this message, you must reenter all values required by the input statement.

Execution errors cause a two-letter code to be printed. The following table contains valid CB-80 error codes.

If an error occurs with a code consisting of an asterisk followed by a letter such as *R, a CB-80 library has failed. Please notify Digital Research of the circumstances under which the error occurred.

Table D-1. CB-80 Error Codes

Code	Error
AC	The argument in an ASC function is a null string.
BN	The value following the BUFF option in an OPEN or CREATE statement is less than 1 or greater than 128.
CE	The file being closed cannot be found in the directory. This occurs if the file has been changed by the RENAME function.
CM	The file specified in a CHAIN statement cannot be found in the selected directory. If no filetype is present, the compiler assumes a type of OVL.
CT	The filetype of the file specified in a CHAIN statement is other than COM or OVL.
CU	A close statement specifies a file identification number that is not active.
CX	OVERLAY DOESN'T LOAD WITH ROOT
DF	An OPEN or CREATE statement uses a file identification number that is already used.

Table D-1. (continued)

Code	Error
DU	A DELETE statement specifies a file identification number that is not active.
DW	The operating system reports that there is no disk or directory space available for the file being written to and no IF END statement is in effect for the file identification number.
DZ	Division by zero was attempted.
EF	An attempt is made to read past the end of file and no IF END statement is in effect for the file identification number.
ER	An attempt is made to write a record of length greater than the maximum record size specified in the OPEN or CREATE statement for this file.
FR	An attempt is made to rename a file to a filename that already exists.
FU	An attempt was made to access a file that was not open.
IF	A filename in an OPEN or CREATE statement or with the RENAME function is invalid for your operating system.
IR	A record number of zero is specified in a READ or PRINT statement.
LN	The argument in the LOG function is zero or negative.
ME	The operating system reports an error during an attempt to create or extend a file. Normally, this means the disk directory is full.
MP	The third parameter in a MATCH function is zero or negative.
NE	A negative value is specified for the operand to the left of the power operator.
NF	A file identification is less than 1 or greater than the maximum number allowed. See Appendix E.

Table D-1. (continued)

Code	Error
NN	An attempt to print a numeric expression with a PRINT USING statement fails because there is not a numeric field in the USING string.
NS	An attempt to print a string expression with a PRINT USING statement fails because there is not a string field in the USING string.
OD	A READ statement is executed but there are no DATA statements in the program, or all data items in all the DATA statements have already been read.
OE	An attempt is made to OPEN a file that does not exist and for which no IF END statement is in effect.
OF	An overflow occurs during a real arithmetic calculation.
OM	The program runs out of dynamically allocated memory during execution.
RB	Random access is attempted to a file activated with the BUFF option specifying more than one buffer.
RE	An attempt is made to read past the end of a record in a fixed file.
RU	A random read or print is attempted to a stream file.
SL	A concatenation operation results in a string greater than the maximum allowed string length.
SQ	A attempt is made to calculate the square root of a negative number.
SS	The second parameter of a MID$ function is zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ is negative.
TL	A tab statement contains a parameter less than 1.
UN	A PRINT USING statement is executed with a null edit string, or an escape character (\) is the last character in an edit string.

Table D-1. (continued)

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

End of Appendix

Appendix E
Implementation Dependent Values

The following implementation dependent values apply to CB-80 version 1 for use with CP/M version 2 and MP/M-80™ versions 1 and 2:

Table E-1. Implementation Dependent Values

Parameter	Value	Minimum
Initial page width for compiler output	80	-
Initial page length for compiler output	66	-
Maximum number of errors maintained	95	-
Maximum nesting of include	6	4
Maximum number of formal parameters	15	15
Maximum number of subscripts in an array	15	15
Maximum unique identifier length	50	31
Maximum number of characters in string constant	255	255
Maximum length of Global and External names	6	6
Maximum nesting of FOR loops	13	-
Maximum nesting of WHILE loops	39	-
Number of files that can be open at one time	20	12
File buffer size in bytes	128	-

The minimum values are the minimum that are used in any CB-80 implementation.

The following extensions exist in CB-80 version 1.3 to provide compatibility with CBASIC version 2. Note that future versions of CB-80 might not support these extensions.

- The LPRINTER statement accepts a WIDTH option to be consistent with CBASIC. The width is ignored.
- Integer and real data is initialized to 0; strings are initialized to null strings.
- The INPUT prompt string can be any expression; the first operand must be a string constant.
- A file OPEN or CREATE statement accepts a RECS field for compatability with CBASIC. The expression is ignored.
- You can use the reserved words LT, GT, GE, LE, EQ, and NE in place of the relational operators <, >, <=, >=, =, and <>.
- CB-80 supports the following form of an IF statement,

  IF <expression> THEN <label>

  but the <label> must be a numeric label.

End of Appendix

Appendix F
Glossary

address: Location in memory.

ambiguous file specification: File specification that contains either of the Digital Research wildcard characters, ? or *, in the filename or filetype or both. When you replace characters in a file specification with these wildcard characters, you create an ambiguous filespec and can reference more than one file in a single command line.

applications program: Program that needs an operating system to provide an environment in which to execute. Typical applications programs are business accounting packages, word processing, and mailing list programs.

argument: Variable or expression value that is passed to a procedure or function and substituted for the dummy argument in the function. Same as "actual argument" or "calling argument". Used interchangeably with "parameter".

array: Data type that is itself a collection of individual data items of the same data type. Term used to describe a form of storing and accessing data in memory, visualized as matrices. The number of extents of an array is the number of dimensions of the array. A one dimensional array is essentially a list.

ASCII: Acronym for American Standard Code for Information Interchange. ASCII is a standard code for representation of the numbers, letters, and symbols that appear on most keyboards.

assembler: Language translator that translates assembly language statements into machine code.

assignment statement: Statement that assigns the value of an expression on the right side of an equal sign to the variable name on the left side of the equal sign.

back-up: Copy of a file or disk made for safe keeping, or the creation of the file or disk.

binary: Base two numbering system containing the two symbols zero and one.

bit: Common contraction for "binary digit". "Switch" in memory that can be set to on (1) or off (0). Eight bits grouped together comprise a byte.

buffer: Area of memory that temporarily stores data during the transfer of information.

byte: Unit of memory or disk storage containing eight bits.

call: Transfer of control to a computer program subroutine.

chain: Transfer of control from the currently executing program to another named program without returning to the system prompt or invoking the run-time monitor.

code: Sequence of statements of a given language that make up a program.

command: Instruction or request for the operating system or a system program to perform a particular action. Generally, a Digital Research command line consists of a command keyword, a command tail usually specifying a file to be processed, and a carriage return.

common: Variables used by a main program and all programs executed through a chain statement.

compiler: Language translator that translates the text of a high level language into machine code.

compiler directive: Reserved words that modify the action of the compiler.

compiler error: Error detected by the compiler during compilation; usually caused by improper formation of language statement.

compiler toggle: "Switch" to modify the output of the compiler.

concatenate: Join one string to another or one file to another.

concatenation operator: Symbol peculiar to a given language that instructs the compiler to combine two unique data items into one.

console: Primary input/output device. The console consists of a listing device such as a screen and a keyboard through which the user communicates with the operating system or the applications program.

constant: String or numeric value that does not change throughout program execution.

control character: Nonprinting character combination that sends a simple command to the operating system or applications program. To enter a Control character, press the Control (CTRL) key on your terminal and strike the character key specified.

control statement: Language statement that transfers control or directs the order of execution of instructions by the processor.

cursor: One-character symbol that can appear anywhere on the video

screen. The cursor indicates the position where the next keystroke at the console will have an effect.

data: Information; numbers, figures, names and so forth.

data base: Large collection of information, usually covering various aspects of related subject matter.

data file: Nonexecutable file of similar information that generally requires a command file to process it.

data structure: Mechanism, including both storage layout and access rules, by which information can be stored and retrieved within a computer system. Data structures can reside in memory or on secondary storage. System tables such as symbol tables, matrices of numerical data, and data files are examples of data structures.

data type: Class or use of the data; for example, integer, real or string.

debug: Remove errors from a program.

default: Values, parameters or options a given command assumes if not otherwise specified.

delimiter: Special characters or punctuation that separate different items in a command line or language statement.

dimension: Refers to the number of extents of an array. A one dimensional array is essentially a list of the elements of the array. A two dimensional array can be visualized as a matrix of rows and columns of storage space for the elements of the array. A three dimensional array can be thought of as a geometric solid having volume, and so forth.

directory: Portion of a disk that contains entries for each file on the disk. In response to the DIR command, CP/M and MP/M systems display the file specifications stored in the directory.

disk, diskette: Magnetic media used to store information. Programs and data are recorded on the disk in the same way that music is recorded on a cassette tape. The term "diskette" refers to smaller capacity removable floppy diskettes. The term "disk" can refer to a diskette, a removable cartridge disk, or a fixed hard disk.

disk drive: Peripheral device that reads and writes on hard or floppy disks. CP/M and MP/M systems assign a letter to each drive under their control.

drive specification: Alpha character A-P followed by a colon that indicates the CP/M or MP/M drive reference for the default or specified drive.

dummy argument: Argument used in the definition of a command or language statement (especially a function) that holds a place that

will later contain a usable "actual" or "calling" argument that is passed to the function by a calling statement. Same as "formal argument."

editor: Utility program that creates and modifies text files. An editor can be used to create documents or code for computer programs.

element: Individual data item in an array.

executable: Ready to run on the processor. Executable code is a series of instructions that can be carried out on the processor. For example, the computer cannot "execute" names and addresses, but it can execute a program that prints names and addresses on mailing labels.

execute a program: Start a program running. When the program is executing, a process is executing a sequence of instructions.

FCB: File Control Block. Structure used for accessing files on disk. Contains the drive, filename, filetype and other information describing a file to be accessed or created on the disk.

field: Portion of a record; length and type are defined by the programmer. One or more fields comprise a record.

file: Collection of related records containing characters, instructions or data; usually stored on a disk under a unique file specification.

filename: Name assigned to a file. The filename can include 1-8 alpha, numeric and/or some special characters. The filename should tell something about the file.

filetype: Extension to a filename. A filetype is optional, can contain from 0 to 3 alpha, numeric and/or some special characters. The filetype must be separated from the filename by a period. Certain programs require that files to be processed have specific filetypes.

file access: Refers to methods of entering a file to retrieve the information stored in the file.

file specification: Unique file identifier. A Digital Research file specification includes an optional drive specification followed by a colon, a primary filename of 1-8 characters, and an optional period and filetype of 0-3 characters. Some Digital Research operating systems allow an optional semicolon and password of 1-8 characters following the filename or filetype. All alpha and numeric characters and some special characters are allowed in Digital Research file specifications.

fixed: Type of file organization used when data is to be accessed randomly - not in sequential order. Refers generally to the nonvarying lengths of the records composing the file.

floating point: Value expressed in decimal notation that can include exponential notation; a real number.

floppy disk: Flexible magnetic disk used to store information. Floppy disks are manufactured in 5 1/4 and 8 inch diameters.

flowchart: Graphic diagram that uses special symbols to indicate the input, output and flow of control of part or all of a program.

flow of control: Order of the execution of statements within a program.

format: System utility that writes a known pattern of information on a disk so a given hardware configuration can properly support reading and writing on that disk.

formatted printing: Output specifically designed in a certain pattern and achieved through particular coded language statements.

fragmentation: Division of storage area in a way that causes areas to be wasted.

function: Subroutine to which you can pass values and which returns a value. Useful when the same code is required repeatedly, as the program can call the function at any time.

global: Relevant throughout an entire program.

hex file: ASCII-printable representation of a code or data file in hexadecimal notation.

hexadecimal notation: Notation for the base 16 number system using the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F to represent the sixteen digits. Machine code is often converted to hexadecimal notation because it can be more easily understood.

high bound: Upper limit of one dimension of an array.

high level language: Set of special words and punctuation that allows a programmer to code software without being concerned with internal memory management.

identifier: String of characters used to name elements of a program, such as variable names, reserved words, and user-defined function names. Commonly used synonymously with "variable name".

include: Call an external file into the code sequence of a program at the point where the include statement is executed.

initialize: Set a disk system or one or more variables to initial values.

I/O: Abbreviation for input/output.

input: Data entered to an executing program, usually from an operator typing at the terminal or by the program reading data from a disk.

instruction: Set of characters that defines an operation.

integer: Positive or negative nonexponential whole number that does not contain a decimal point.

interface: Object that allows two independent systems to communicate with each other, as an interface between the hardware and software in a microcomputer.

intermediate code: Code generated by the syntactical and semantic analyzer portions of a compiler.

interpreter: Computer program that translates and executes each source language statement before translating and executing the next one.

ISAM: Abbreviation for Indexed Sequential Access Method.

key: Particular field of a record on which the processing is performed.

keyword: Reserved word with special meaning for statements or commands.

kilobyte: 1024 bytes denoted as 1K. 32 kilobytes equal 32K. 1024 kilobtyes equal one megabyte, or over one million bytes.

linker: System software module that connects previously assembled or compiled programs or program modules into a unit that can be loaded into memory and executed.

linked list: Data structure in which each element contains a pointer to its predecessor or successor (singly linked list) or both (double linked list).

list device: Device such as a printer onto which data can be listed or printed.

listing: Output file created by the compiler that lists the statements in the source program, the line numbers it has assigned to them, and possibly other optional information.

literal data: Verbatim translation of characters in the code, such as in screen prompts, report titles and column headings.

load: To move code from storage into memory for execution.

local variable: Relevant only within a specific portion of a program, such as within a function.

logged-in: Made known to the operating system, in reference to

drives. A drive is logged-in when it is selected by the user or an executing process.

logical: Representation of something such as a console, memory or disk drive that might or might not be the same in its actual physical form. For example, a hard disk can occupy one physical drive, and yet you can divide the available storage on it to appear to the user as if there were several different drives. These apparent drives are the logical drives.

logical device: Reference to an I/O device by the name or number assigned to the physical device.

logical operator: NOT, AND, OR, and XOR.

lower bound: Lower limit of one dimension of an array.

machine code: Output of an assembler or compiler to be executed directly on the target processor.

machine language: Instructions directly executable by the processor.

memory: Storage area within and/or attached to a computer system.

microprocessor: Silicon chip that is the Central Processing Unit (CPU) of the microcomputer system.

mixed mode: Combination of integer and real or numeric and string values in an expression. Mixed string and numeric operations are generally not allowed in high level languages.

mnemonic operator: Alphabetical symbol for algebraic operator: LT, LE, GT, GE, NE, and EQ.

module: Section of software having well-defined input and output that can be tested independently of other software.

multiple line function: Function composed of a function definition statement and one or more additional statements.

numeric constant: Real or integer quantity that does not vary within the program.

numeric variable: Real or integer identifier to which varying numeric quantities can be assigned during program execution.

null string: A string that contains no character; essentially an empty string.

object code: Output of an assembler or compiler that executes on the target processor.

open: System service that informs the operating system of the manner in which a given resource, usually a disk file, is intended

to be used.

operating system: Collection of programs that supervises the execution of other programs and the management of computer resources. An operating system provides an orderly input/output environment between the computer and its peripheral devices, enabling user programs to execute safely.

operation: Execution of a piece of code.

option: One of a set of parameters that can be part of a command or language statement. Options are used to modify the output of an executing process.

output: Data that the processor sends to the console, printer, disk, or other storage media.

parameter: Value supplied to a command or language statement that provides additional information for the command or statement. Used interchangeably with "argument." An actual parameter is a value that is substituted for a dummy or formal argument in a given procedure or function when it is invoked.

peripheral device: Devices external to the CPU. For example, terminals, printers, and disk drives are common peripheral devices that are not part of the processor, but are used in conjunction with it.

pointer: Data item whose value is the address of a location in memory.

primitive: Most basic or fundamental unit of data such as a single digit or letter.

process: Program that is actually executing, as opposed to being in a static state of storage on disk.

program: Series of specially coded instructions that performs specific tasks when executed on a computer.

prompt: Any characters displayed on the input terminal to help the user decide what the next appropriate action is. A system prompt is a special prompt displayed by the operating system, indicating to the user that it is ready to accept input.

random access: Method of entering a file at any record number, not necessarily the first record in the file.

random access file: File structure in which data can be accessed in a random manner, irrespective of its position in the file.

random number: Number selected at random from a set of numbers.

real number: Numeric value specified with a decimal point; same as "floating point notation".

record: One or more fields usually containing associated information in numerical or textual form. A file is composed of one or more records and generally stored on disk.

record number: Position of a specific record in a fixed-length file, relative to record number 1. A key by which a specific record in a fixed file is accessed randomly.

recursive: Code that calls itself.

relational operator: Comparison operator. The following set of operators expressed in algebraic or mnemonic symbols: LT, LE, NE, EQ, GT, GE, EQ. A relational operator states a relationship between two expressions.

reserved word: Keyword that has a special meaning to a given language or operating system.

return value: Value returned by a function.

row-major order: Order of assignment of values to array elements in which the first item of the subscript list indicates the number of "rows" in the array.

run a program: Start a program executing. When a program is running, the microprocessor chip is executing a series of instructions.

run-time error: Error occurring during program execution.

run-time monitor: Program that directly executes the coded instructions generated by a compiler/interpreter.

sequential access: Type of file structure in which data can only be accessed serially, one record at a time. Data can be added only to the end of the file and cannot be deleted. An example of a sequential access media is magnetic tape.

source program: Text file that is an input file for a processing program, such as an editor, text formatter, assembler or compiler.

statement: Defined way of coding an instruction or data definition using specific keywords in a specific format.

storage: Place for keeping data temporarily in memory or permanently on disk.

stream organization: Type of file organization used when data is to be accessed sequentially. Can contain variable length records.

string constant: Literal data, as in a screen prompt, column heading, or title of a report.

string variable: Identifier of type string to which varying strings

can be assigned during program execution.

subroutine: Section of code that performs a specific task, is logically separate from the rest of the program, and can be prewritten. A subroutine is invoked by another statement and returns to the place of invocation after executing. Subroutines are useful when the same sequence of code is used more than once in a program.

subscript: Integer expression that specifies the position of an element in an array.

subscript list: Numeric value appended to a variable name that indicates the number of elements in each dimension in the array of that name. Each dimension must have a value in the subscript list indicating the number of elements for which to allocate storage space.

syntax: Rules for structuring statements for an operating system or programming language.

toggle: "Switch" enabled by a special code in the command line that modifies the output of the executing program.

trace: Option used for run-time debugging. The trace option generally lists each line of code as it executes to enable the programmer to note where a problem occurs.

upward-compatible: Term meaning that a program created for the previously released operating system or compiler runs under a later release of the same software program.

user-defined function: Set of statements created and given a function name by the user. The function performs a specific task and is called into action by referencing the function by name.

utility: Tool. Program or module that facilitates certain operations, such as copying, erasing and editing files, or controlling the cursor positioning on the video screen from within a program. Utilities are created for the convenience of programmers and applications operators.

value: Quantity expressed by an integer or real number.

variable: Name to which the program can assign a numerical value or string.

variable length: Usually refers to records, where each record in a file is not necessarily the same length as another.

variable name: Same as variable.

wildcard characters: Special characters, ? and *, that can be included in a Digital Research filename and/or filetype to identify more than one file in a single file specification.

End of Appendix

Index