Hyperion™ Programmer Guide

620005-000-02

BYTEC PROGRAM LICENSE AGREEMENT

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS SOFTWARE PACKAGE. OPENING THIS SOFTWARE PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

Bytec Management Corporation provides this program and licenses its use in the United States and Canada. You assume responsibility for the selection of the program to achieve your intended results obtained from the program.

LICENSE

You may:

a. use the program on ***a single*** Bytec microcomputer;

b. copy the program into any machine readable or printed form for backup or modification purposes in support of your use of the program on the single Bytec microcomputer (Certain programs, however, may include mechanisms to limit or inhibit copying. They are marked "copy protected".);

c. modify the program and/or merge it into another program for your use on the single Bytec microcomputer. (Any portion of this program merged into another program will continue to be subject to the terms and conditions of this Agreement.); and,

d. transfer the program and license to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred; this includes all modifications and portions of the program contained or merged into other programs.

You must reproduce and include the copyright notice on any copy, modification or portion merged into another program.

YOU MAY NOT USE, COPY, MODIFY, OR TRANSFER THE PROGRAM, OR ANY COPY, MODIFICATION OR MERGED PORTION, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE.

IF YOU TRANSFER POSSESSION OF ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM TO ANOTHER PARTY, YOUR LICENSE IS AUTOMATICALLY TERMINATED.

TERM

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in the Agreement or if you fail to comply with any term or condition of the Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form.

LIMITED WARRANTY

THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT BYTEC MANAGEMENT CORPORATION OR AN AUTHORIZED BYTEC COMPUTER DEALER) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME STATES OR PROVINCES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE OR PROVINCE TO PROVINCE.

Bytec Management Corporation does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, Bytec Management Corporation warrants the diskette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your bill of sale.

LIMITATIONS OF REMEDIES

Bytec Management Corporation's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) not meeting Bytec Management Corporation's "Limited Warranty" and which is returned to Bytec Management Corporation or an Authorized Bytec Computer Dealer with a copy of your bill of sale, or

2. if Bytec Management Corporation or the dealer is unable to deliver a replacement diskette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL BYTEC MANAGEMENT CORPORATION BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF BYTEC MANAGEMENT CORPORATION OR AN AUTHORIZED BYTEC COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES OR PROVINCES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY OF INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the Province of Ontario, Canada.

Should you have any questions concerning this Agreement, you may contact Bytec Management Corporation by writing to Customer Support Division, Bytec Management Corporation, 8 Colonnade Road, Ottawa, Ontario, Canada, K2E 7M6.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERCEDES ANY PROPOSAL, OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Important Notice:

* Dynalogic Info-Tech Corporation is committed to providing Hyperion users with an optimal balance between IBM(tm) PC compatibility and enhanced capability.

  The GW-BASIC(tm) provided with all Hyperions produced during the first quarter of calendar year 1983 does not meet our criteria for either compatibility or enhanced capability. Microsoft(tm) Corporation, producers of the GW-BASIC, were unable to meet stated delivery deadlines for the IBM(tm) PC compatible version.

* We will therefore gladly upgrade the BASIC supplied with these early Hyperions, at no charge to the owner. You will qualify for this free upgrade if:

  1. This notice was present in your Hyperion Programmer Guide;

     -- AND --

  2. Your Hyperion Programmer Diskette contains GWBASIC.COM, instead of the compatible BASICA.COM;

     -- AND --

  3. You have returned your Hyperion Warranty Certificate, completed by the authorized Dynalogic Dealer from whom you purchased your Hyperion.

* Most programs written in the interim GW-BASIC will run without change both on the IBM(tm) PC and on Hyperions using the eventual BASICA version.

  The contents of your current Hyperion Programmer Guide will also be replaced when the BASIC version is replaced by Dynalogic.

  For those programmers who will be writing BASIC code before the upgrade, we have enclosed a list of differences between the documented and actual actions of the interim version.

* Naturally, Dynalogic regrets any inconvenience this may cause Hyperion users. We will be delivering the upgrade package as soon as Microsoft(tm) make their compatible BASIC available.

List of differences

1. Starting BASIC:

For the interim version, the command GWBASIC must be used from DOS to initiate BASIC, instead of the command BASICA.

2. Statements and Functions:

Statement /function	As Documented	Current GWBASIC
LINE	Optional attibute	Attribute not optional
LINE	Attribute out of range automatically changed to default	Attribute out of range results in error message
PSET	Optional attibute	Attribute not optional
PSET	Attribute out of range automatically changed to default	Attribute out of range results in error message
PRESET	Optional attibute	Attribute not optional
PRESET	Attribute out of range automatically changed to default	Attribute out of range results in error message
CIRCLE	Optional color	Color not optional
CIRCLE	Color out of range automatically changed to default	Color out of range results in error message
KEY OFF		Syntax error incorrectly reported if keys on
SCREEN	Screen change to mode 0 retains current width	Screen change to mode 0 sets width to 80
CHR$(n)		Outputs a blank space for any n < 32
ON COM COM ON OPEN "COM1:..."		Not available in current GWBASIC version
BLOAD & BSAVE	Add extension .BAS to filename before disk search or save	Use exact filename as specified

3. Keyboard Input (INKEY$) Differences:

It should be possible to detect the presence of an extended key scan code by testing for LEN(INKEY$)=2. In the current version, that test fails.

As well, the possible extended key scan codes in the current version are limited to those listed below. This list should be used instead of the list given in Appendix G.

Extended Key Scan Codes:

Code	Meaning	Code	Meaning
11	Home	12	Ctrl + Home
30	Up Arrow		
24	Pg Up		
29	Left Arrow	2	Ctrl + Left Arrow
28	Right Arrow	6	Ctrl + Right Arrow
14	End	5	Ctrl + End
31	Down Arrow		
25	Pg Dn	26	Ctrl + Pg Dn
18	Ins		
127	Del		
9	Tab	1	Shift + Tab

4. Comressed Save Incompatibility:

The current GW-BASIC can not correctly read IBM BASIC programs from a diskette if they were saved in the compressed (non-ASCII) form.

To transfer an IBM BASIC program to the current GW-BASIC on the Hyperion, SAVE the program from the IBM BASIC using the ",A" option of the SAVE command. This SAVEs the program in its original pure text form. This ASCII file can then be LOADed into the current Hyperion GW-BASIC.

5. Error Codes:

The current GW-BASIC reports certain error conditions with different error code numbers than the IBM PC version. This will affect only those programs that test for specific error code conditions.

HYPERION PROGRAMMER GUIDE

Published by: Dynalogic Info-Tech Corporation
06 February 1983
Version 02

Federal Communication Commission Compliance

The Hyperion is subject to Federal Communication Commission (FCC) rules. The certification process is underway. The Hyperion will comply with the appropriate FCC rules prior to final delivery to buyers or centers of distribution. The Hyperion will also have Canadian Standards Association (CSA) approval, and will conform with the Government of Canada telecommunications interconnect requirements (CS-03).

The Hyperion generates and uses radio frequency energy and, if not installed and used according to the manufacturer's instructions, may cause interference to radio and television reception. The Hyperion has been type tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause interference to radio or television reception: riorient the receiving antenna; relocate the computer; plug the computer into a different branch outlet; consult an experienced radio/television technician; or consult the booklet, How to Identify and Resolve Radio-TV Interference Problems, prepared by the FCC and available from the U.S. Government Printing Office, Washington, D.C. 20402, Stock No. 004-000-00345-4.

Trademarks

Hyperion is a trademark of Dynalogic Info-Tech Corporation.
MS-DOS is a trademark of Microsoft Corporation.
Microsoft is a trademark of Microsoft Corporation.
MULTIPLAN is a trademark of Microsoft Corporation.
IBM is a trademark of International Business Machines Corporation.

Disclaimer

The information in this manual has been carefully prepared and checked for completeness and accuracy. There is, however, always the possibility of omission or error. In such an event, Dynalogic Info-Tech Corporation cannot assume liability for any damages resulting from the use of this manual.

TABLE OF CONTENTS

PART I - BASIC INTERPRETER

SECTION 1 - GENERAL INFORMATION ABOUT BASIC

SECTION 2 - THE BASIC FULL SCREEN EDITOR

SECTION 3 - BASIC COMMANDS

SECTION 4 - BASIC STATEMENTS & VARIABLES

SECTION 5 - BASIC FUNCTIONS

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 1 - MACRO ASSEMBLER

Section 2 - LINK

Section 3 - DEBUG

Section 4 - EDLIN

Section 5 - CREF

Section 6 - EXE2BIN

PART III - APPENDICES

PART I - BASIC INTERPRETER

SECTION 1 - GENERAL INFORMATION ABOUT BASIC

1.1 STARTING BASIC

Start DOS, as described in the Hyperion User Guide. Remove the Hyperion User Diskette from drive A, and insert the Hyperion Programmer Diskette. Enter the command: "BASICA", and press the Return key. BASIC will display the version, the release number, and the number of free bytes.

1.2 MODES OF OPERATION

After BASIC is initialized, it types the prompt "Ok". "Ok" means BASIC is at command level, that is it is ready to accept commands. At this point, BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the **direct mode,** BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The **indirect mode** is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format. Square brackets indicate optional entries, angle brackets indicate data entered by the programmer.

nnnnn BASIC statement[:BASIC statement...] <Rtn>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of automatic return feature. It allows you to continue typing a logical line on the next physical line without entering a <Rtn>. When you reach the end of the physical line, the Hyperion automatically returns the cursor to the left-most position of the next line. Entering <Rtn> is the signal to BASIC that the end of a logical line has been reached.

1.4 LINE NUMBERS

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.5 CHARACTER SET

The BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The **alphabetic characters** in BASIC are the upper case and lower case letters of the alphabet.

The **numeric characters** in BASIC are the digits 0 through 9.

The following **special characters** and **keys** are recognized by BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent

#	Octothorpe
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period, or decimal point
'	Apostrophe
;	Semi-colon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash, or integer division symbol
@	At-sign
_	Underline
<RubOut>	Deletes last character typed
<Esc>	Escapes Edit Mode subcommands (See Section 2.2)
<Tab>	Moves print position to next tab stop (Tab stops are every eight columns.)
<Rtn>	Terminates input of a line

1.6 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A **string constant** is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. **Integer Constants**

   Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. **Fixed Point Constants**

   Positive or negative real numbers, i.e., numbers that contain decimal points.

3. **Floating Point Constants**

   Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38. Examples:

   ```
   235.988E-7 = .0000235988
   2359E6 = 2359000000
   ```

   (Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. **Hex Constants**

   Hexadecimal numbers with the prefix &H. Examples:

   ```
   &H76 = Decimal 118
   &H32F = Decimal 815
   ```

5. **Octal Constants**

Octal numbers with the prefix &O or &. Examples:

```
&O347 = Decimal 231
&1234 = Decimal 668
```

1.6.1 Single and Double Precision Form for Numeric Constants.

Numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A **single precision constant** is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A **double precision constant** is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#).

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.7 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.7.1 Variable Names and Declaration Characters

BASIC variable names may be any length, however, only the first 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is allowed. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word, although BASIC will allow embedded reserved words. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string of text. **String variable names** are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

% Integer variable

! Single precision variable

Double precision variable

The default type for a numeric variable name is single precision.

Examples of BASIC variable names follow:

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single precision value

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in section 5.

1.7.2 Array Variables

An **array** is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An **array variable name** has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

1.7.3 Space Requirements

VARIABLES:	BYTES
INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8

ARRAYS:	BYTES
INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.8 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e. that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision:
   Example:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428471428571
```

   The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

   The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators (see Section 5) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.
Example:

```
10 A = 2.04
20 B# = A
30 PRINT A; B#
RUN
 2.04  2.039999961853027
```

1.9 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

1.9.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

OPERATOR	OPERATION	SAMPLE EXPRESSION
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y Y/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
$\frac{X-Y}{Z}$	X-Y/Z
XY/Z	X*Y/Z
X+Y/Z	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(-Y)	X*(-Y)

Two consecutive operators must be separated by parentheses.

1.9.1.1 Integer Division and Modulus Arithmetic

Two additional operators are available in BASIC: Integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.
Example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.9.1.2 Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.9.1.3 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Section 5.)

Operator	Relation Test	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 5.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X) <0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.9.2 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be combined to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 10
```

Logical Operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16=16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14=14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010=1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.9.3 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's intrinsic functions are described in Section 5.

BASIC also allows "user defined" functions that are written by the programmer. See DEF FN, Section 5.

1.9.4 String Operations

Strings of test may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78" WHERE B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.10 ERROR MESSAGES

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of BASIC error codes and error messages, see Appendix C.

PART I - BASIC INTERPRETER

SECTION 2 - THE BASIC FULL SCREEN EDITOR

2.1 The BASIC Full Screen Editor

The time saving benefit of the Full Screen Editor during program development cannot be over emphasized. To that end, it is suggested that a sample program be entered and each edit command practiced until it becomes second nature.

In the following discussion of edit commands, the term "cursor" refers to the "blinking" line appearing just to the right of the last character typed. This marks the next position at which a character is to be inserted or deleted.

The dynamic nature of editing anywhere on the screen makes it difficult to provide clear examples of command usage in printed text; therefore, the best way of getting the "feel" for the editing process is to try editing a few lines while studying the edit commands that follow.

2.1.1 Inputting the BASIC Program

Any line of text typed while BASIC is in Direct Mode will be processed by the Full Screen Editor. BASIC is always in Direct Mode after the prompt OK and until a RUN command is given.

Any line of text typed that begins with a numeric character (digit) is considered a <u>Program statement</u> and will be processed in one of four ways:

1. A new line is added to the program. This occurs if the line number is legal (range is 0 thru 65529) and at least one non-blank character follows the line number in the line.

2. An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. This line is replaced with the text of the newly entered line.

3. An existing line is deleted. This occurs if the line number matches the line number of an existing line and the entered line contains ONLY a line number.

4. An error is produced.

   a) If an attempt is made to delete a non-existent line, an "Undefined line number" error message is displayed.

b) If program memory is exhausted, and a line is added to the program, the error; "Out of Memory" is displayed and the line is not added.

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon (:).

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the automatic return feature. This feature automatically returns the cursor to the left margin of the next line when you approach the 80th column of a physical line. When <Rtn> is finally entered, the entire logical line is passed to BASIC for storage in the program.

Occasionally, BASIC may return to Direct Mode with the cursor positioned on a line containing a message issued by BASIC such as "OK". When this happens the line is automatically erased. This is provided as a courtesy to the programmer. If the line were not erased and the programmer typed <Rtn>, the message would be given to BASIC and a "Syntax Error" would surely result. BASIC messages are terminated by HEX 'FF' to distinguish them from user text.

2.1.2 Altering Lines with the Editor

Editing existing lines on the Screen is achieved by moving the cursor on the screen to the place requiring change and then performing one of the following four functions:

1. Overtyping characters already there.

2. Deleting characters to the left of the cursor.

3. Deleting characters to the right of the cursor.

4. Inserting characters at the cursor while pushing characters following the cursor to the right.

5. Adding, or appending, characters to the end of the current logical line.

The Full Screen Editor recognizes 9 special or numeric key-pad keys, the back-space key, the ESC key, plus the CTRL key for moving the cursor to a location on the screen, inserting characters, or deleting characters. The keys and their ASCII decimal values are:

Home	Moves the cursor to the upper left hand corner of the screen. (ASCII decimal value - 11)
Ctrl+Home	Clears the screen and positions the cursor in the upper left hand corner of the screen. (ASCII decimal value - 12)
↑	Moves the cursor up one line. (ASCII decimal value - 30)
↓	Moves the cursor one position down. (ASCII decimal value - 31)
←	Moves the cursor one position left. When the cursor is advanced beyond the left of the screen, it will be moved to the right side of the screen on the preceeding line. (ASCII decimal value - 29)
→	Moves the cursor one position right. When the cursor is advanced beyond the right of the screen, it will be moved to the left side of the screen on the next line down. (ASCII decimal value - 28)
Ctrl+→	Moves the cursor right to the next word. The next word is defined as the next character to the right of the cursor in the set [A...Z] or [0...9]. (ASCII decimal value - 14)
Ctrl+←	Moves the cursor left to the previous word. The next word is defined as the next character to the left of the cursor in the set [A..Z] or [0..9]. (ASCII decimal value - 02)
End	Moves the cursor to the end of the Logical Line. Characters typed from this position are appended to the line. (ASCII decimal value -)
Ctrl+End	Depressing the CTRL and END key erases to the End of Logical Line from the current cursor position. All physical Full Screens are erased until the terminating carriage return is found. (ASCII decimal value - 05)

Ins — Toggles Insert Mode. If Insert Mode is off, turns it on. If on, then turns it off. (ASCII decimal value - 18)

Insert Mode is indicated by the blinking cursor blotting the lower half of the character position. In Graphic Modes, the normal cursor covers the whole character position. When Insert Mode is active only the lower half of the character position is blotted.

When in Insert Mode, characters following the cursor are moved to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding is observed. That is, as characters advance off the right side of the screen they are inserted from the left on subsequent lines.

When out of Insert Mode, characters typed will replace existing characters on the line.

Tab — When out of Insert Mode, depressing the TAB key moves the cursor over characters until the next tab stop is reached. Tab stops occur every 8 character positions. (09)

When in Insert Mode, depressing the TAB key causes blanks to be inserted from the current cursor position to the next tab stop. Line folding is observed as above.

Del — Deletes one character immediately to the right of the cursor for each depression. All characters to the right of the one deleted are then moved one position left to fill in the one deleted. If a logical line extends beyond one physical line, characters on subsequent lines are moved left one position to fill in the previous space, and the character in the 1st column of each subsequent line is moved up to the end of the preceeding line. (ASCII decimal value - 128)

Rub Out — Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of the cursor are moved left one position. Subsequent characters and lines within the current logical line are moved up as with the DEL key. (ASCII decimal value - 08)

Esc When typed anywhere in the line causes the entire logical line to be erased. (ASCII decimal value - 21)

Ctrl+Brk Returns to Direct Mode, without saving any changes that were made to the current line being edited. (ASCII decimal value - 03)

Other Control Characters may be used in BASIC:

Ctrl+NumLock Pauses, suspending program execution. Pressing any key resumes program execution.

Ctrl+G Sounds the speaker in the Hyperion.

Ctrl+H Deletes the last character typed (i.e. RubOut).

2.1.3 Changing a BASIC Program

Modifying existing programs is achieved by displaying program lines on the screen with the LIST statement. List the range of lines to be edited, (See the LIST statement, section 5). Position the cursor at the line to be edited, modify the line using the keys described in "Altering Lines with the Editor". Type <Rtn> to store the modified line in the program.

NOTE: A program line is not actually modified within the BASIC program until <Rtn> is entered. Therefore, when several lines need alteration, it is sometimes easier to move around the screen making corrections to several lines at once, and then, go back to the first line changed and enter <Rtn> at the beginning of each line and by doing so store the modified line in the program.

Note that it is not necessary to move the cursor to the end of the logical line before pressing <Rtn>. The Full Screen Editor remembers where each logical line ends and transfers the whole line even if the carriage return is typed at the beginning of the line.

To truncate a line at the current cursor position, enter

CTRL+END followed by <Rtn>.

SYNTAX ERRORS

When a Syntax Error is encountered during program execution, BASIC automatically enters EDIT at the line that caused the error. For example:

```
10 A = 2$12
RUN
?Syntax Error in 10
10 A = 2$12
_
```

The Full Screen Editor has displayed the line in error and positioned the cursor under the digit 1. The user moves the cursor right to the dollar sign ($) and changes it to an up-arrow (^), followed by a carriage return. The corrected line is now stored back in the program.

In this example, storing the line back in the program causes all variables to be lost. Had the programmer wanted to examine the contents of some variable before making the change, BREAK would be typed to return to Direct Mode. The variables would be preserved since no program line was changed, and after the programmer was satisfied, the line could be edited and the program re-run.

2.1.4 The EDIT Statement

With the Full Screen Editor, the EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. The line may then be modified using the keys described in "Altering Lines with the Editor".

PART I - BASIC INTERPRETER

SECTION 3 - BASIC COMMANDS

3.1 AUTO - Command

Format: AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Ctrl+Brk. The line in which Ctrl+Brk is typed is not saved. After Ctrl+Brk is typed, BASIC returns to command level.

Example:

AUTO 100,50	Generates line numbers 100, 150, 200 ...
AUTO	Generates line numbers 10, 20, 30, 40 ...

3.2 CLEAR - Command

Format: CLEAR [,[<expression1>][,<expression2>]]

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC-80.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: Hyperion BASIC allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Example:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

3.3 CONT - Command

Format: CONT

Purpose: To continue program execution after a Ctrl+Brk has been entered, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjuction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. Execution cannot be CONTinued if a direct mode error has occurred during the break.

Example: See example in STOP, Section I.4.

3.4 DELETE - Command

Format: DELETE[<line number>][-<line number>]

Purpose: To delete program lines.

Remarks: BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

A period "." may be used instead of a line number to indicate the current line.

Example:

DELETE 40	Delete line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40.

3.5 EDIT - Command

Format: EDIT <line number>

Purpose: To edit a specified line.

Remarks: With the Full Screen Editor, the EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. The line may then be modified using the keys described in "Altering Lines with the Editor".

<line number> is the program line number of a line existing in the program. If there is no such line, an "Undefined Line number" error message is displayed.

The "." always gets the last line referenced by an EDIT statement, LIST command, or Error message. Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter EDIT at the current line. (The line number symbol "." always refers to the current line).

3.6 FILES - Command

Format: FILES [filespec]

Purpose: This command displays the names of files on a specified diskette. It is similar to the 'DIR' command in DOS.

Remarks: [filespec] is a string expression for the file specification. If [filespec] is not entered, all files on the source drive will be listed.

If [filespec] is included, all files matching the filename are listed. FILES allows the DOS 'wildcard' feature to be used: '?' may be substituted for any single character, or '*' may be used as a substitute for a string of characters.

If a drive is included in the filename, the files which match the [filespec] on that drive are listed. Otherwise, the source drive is the default drive.

Example: FILES

This displays all files on the default source drive.

FILES "*.COM"

This displays all files with the extension '.COM' on the default source drive.

Files "B:*.*"

This displays all files on drive B.

FILES "TEXT??.COM"

This displays all files on the default source drive whose filenames begin with TEXT followed by two or less other characters, and an extension of '.COM'.

3.7 KILL - Command

Format: KILL <filename>

Purpose: To delete a file from a disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

See also Appendix B.

3.8 LIST - Command

Format 1: LIST [<line number>]

Format 2: LIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Ctrl+Brk.) If <line number> is included, BASIC will list only the specified line.

Format 2: This format allows the following options.

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

Example: Format 1:

LIST	Lists the program currently in memory.

Format 2:

LIST 150-	Lists all lines form 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

3.9 LLIST - Command

Format: LLIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of BASIC.

Example: See the examples for LIST, Format 2.

3.10 LOAD - Command

Format: LOAD <file spec> [,R]

Purpose: To load a BASIC program into memory from disk and to optionally run the program.

Remarks: <file spec> is a valid string expression containing the device and file name. The device must be 4 characters in length. The file name may be 1 to 8 characters in length.

When ',R' is specified, the program will begin execution from the first statement after loading.

LOAD closed all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R: option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Rules:

1. If the device identifier is omitted and the filename is less than 1 character or greater than 8 characters in length, a "Bad File Name" error is issued and the load is aborted.

2. If the ,R option is omitted, BASIC returns to Direct Mode after the program is loaded. If the ,R option is specified, the program is executed after loading.

3. RUN <file spec> is equivalent to LOAD <file spec>,R

Example:

```
LOAD"MENU"          'Load program MENU, do not run it.
LOAD"INVENT",R      'Load and run the program INVENT.
RUN"INVENT"         'Same as LOAD"INVENT",R
LOAD"CAS1:"         'Load the next Program encountered.
```

3.11 MERGE - Command

Format: MERGE <filename>

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. If the filename is less than 1 character or greater than 8 characters in length, a "Bad File Name" error is issued and the MERGE is aborted.

If the program being merged was not saved in ASCII with a ,A option, a "Bad File Mode" error is issued. The program in memory remains unchanged.

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example: MERGE "SUBRTN"

3.12 NAME - Command

Format: NAME <old filename> AS <new filename>

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:

```
Ok
NAME "AACT" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

3.13 NEW - Command

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

3.14 RENUM - Command

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines number 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Example:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

3.15 RUN - Command

Format: RUN <filename>[,R]

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also Appendix B.

3.16 SAVE - Command

Format: SAVE <filename>[,A ¦ ,P]

Purpose: To save a program file on disk.

Remarks: <filename> is quoted string that conforms to DOS requirements for filenames. If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Example:

```
SAVE"COM2",A
SAVE"PROG",P
```

See also Appendix B.

3.17 SYSTEM - Command

Format: SYSTEM

Purpose: Exits BASIC and returns to DOS.

Remarks: SYSTEM closes all files before it returns to DOS. Your BASIC program is not saved.

3.18 TRON/TROFF - Commands

Format: TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
OK
```

PART I - BASIC INTERPRETER

SECTION 4 - BASIC STATEMENTS & VARIABLES

4.1 BEEP - Statement

Format: BEEP

Purpose: The BEEP statement sounds the speaker at 800 Hz for 1/4 seconds.

Example: 2430 IF X < 20 THEN BEEP 'X is out of range.

4.2 BLOAD - Statement

Format: BLOAD <file spec> [,<offset>]

Purpose: The BLOAD statement allows a file to be loaded anywhere in user memory.

Remarks: <file spec> Is a valid string expression containing the device and file name. The device must be 4 characters in length. The file name may be 1 to 8 characters in length.

<offset> Is a valid numeric expression returning an unsigned Integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement at which loading is to start.

Action: If the BLOAD statement is entered in direct mode, the file names will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message "Skipped." for the files not matching the named file, and "Found." when the named file is found. Types of files and their letter are:

- .B For Binary Basic Programs.
- .P For Protected Binary Basic Programs.
- .A For Ascii Basic Programs.
- .M For Memory Image files.
- .D For Data Files.

To see what files are on a cassette tape, enter; LOAD"FOO" or some other name that is known not to be on the tape. All file names will then be displayed.

Note that Ctrl+Brk may be typed at any time during BLOAD or LOAD. between files or after a time-out period, Basic will exit the search and return to Direct Mode. Previous memory contents remain unchanged.

If the BLOAD command is executed in a Basic program, the file names skipped and found are not displayed on the Screen.

Rules:

1. If device is omitted, the source drive is assumed.

2. If the device identifier is omitted and the filename is less than 1 character or greater than 8 characters in length, a "Bad File Name" error is issued and the load is aborted.

3. If the device identifier is specified and the filename is omitted, the NEXT Memory Image file encountered is loaded.

4. If offset is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location it was saved from.

5. If offset is specified, a DEF SEG statement should be executed before the BLOAD. When offset is given, BASIC assumes the user wants to BLOAD at an address other than the one saved. The last known DEF SEG address will be used.

6. CAUTION: BLOAD does not perform an address range check. That is, it is possible to BLOAD anywhere in memory. The user must not BLOAD over BASIC's stack, BASIC Program or BASIC's variable area.

Example:

```
10 'Load a machine language program at 60:F000
20 DEF SEG          'Restore Segment to BASIC's DS.
30 BLOAD"PROG1",&HF000    'Load PROG1 into the DS.
```

```
10 'load the screen buffer from disk.
20 DEF SEG= &HB800  'Point segment at screen buffer.
30 BLOAD"PICTURE",0 'Load file PICTURE into screen.
```

Note that the DEF SEG statement in 20 and the offset of 0 in 30 is wise. This guarantees that the correct address is used.

The BSAVE example in the next section illustrates how PICTURE was saved.

4.3 BSAVE - Statement

Format: BSAVE <file spec>,<offset>,<length>

Remarks: <file spec> is a valid string expression containing the device and file name. The device must be 4 characters in length. The file name may be 1 to 8 characters in length.

<offset> is a valid numeric expression returning an unsigned Integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG to start saving from.

<length> is a valid numeric expression returning an unsigned Integer in the range 1 to 65535. This is the length of the memory image to be saved.

Rules:

1. If device is omitted, the DOS default diskette drive is used.

2. If filename is less than 1 character or greater than 8 characters in length, a "Bad File Name" error is issued and the save is aborted.

3. If offset is omitted, a "Bad File Name" error is issued and the save is aborted. A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

4. If length is omitted, a "Bad File Name" error is issued and the save is aborted.

Example:

```
10 'Save the screen buffer on disk.
20 DEF SEG= &HB800          'Point segment at screen buffer.
30 BSAVE"PICTURE",0,16384   'Save screen buffer in file PICTURE
```

Note that the DEF SEG statement in 20 and the offset of 0 in 30 is wise. This guarantees that the correct address is used.

4.4 CALL - Statement

Format: CALL <variable name>[(<argument list>)]

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 1.5)

<variable name> contains the address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the variables or constants, separated by commas, that are passed to the external subroutine.

Invocation of the CALL statement causes the following to occur:

1. For each parameter in the argument list; the 2 byte offset into the DS of the parameter's location is pushed onto the stack.

2. The return address Code segment [CS], and offset are pushed onto the Stack.

3. Control is transferred to the user's routine via the segment address given in the last DEF SEG statement, and offset given in <variable name>.

The user's routine now has control. Parameters may be referenced by moving the Stack pointer [SP] to the Base Pointer [BP] and adding a positive offset to [BP].

Rules:

1. The CALLed routine may destroy any registers.

2. The CALLed program MUST know how many parameters were passed. Parameters are referenced via a positive offset being added to [BP]. (Assuming the called routine moved the current stack pointer into BP, ie: MOV BP,SP).

   That is, the location of p1 is at 8[BP], p2 is at 6[BP], p3 is at 4[BP], ...etc.

   The CALLed routine must do a RET <n> where <n> is the number of parameters in the argument list *2. This is necessary in order to adjust the stack to the point at the start of the calling sequence.

3. Values are returned to BASIC by including the variable name which will receive the result in the argument list.

4. If the argument is a string, the parameter's offset points to 3 bytes called the "String Descriptor. Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

   CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

   ```
   20 A$ = "BASIC"+""
   ```

   This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

5. Strings may be altered by user routines but the length MUST NOT be changed! BASIC cannot correctly erase strings if their lengths are modified by external routines.

Example:

```
100 DEF SEG=&H8000
110 FOO=0
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment to 8000 Hex. FOO is set to zero so that the call to FOO will execute the subroutine at location 8000H.

The following sequence of 8086 assembly language demonstrates access of the parameters passed and storing a return result in the variable 'C'.

```
MOVE  BP,SP       ;Get the current Stack posn in BP.
MOVE  BX,6[BP]    ;Get address of B$ in dope.
MOVE  CL,[BX]     ;Get length of B$ in CL.
MOVE  DX,1[BX]    ;Get addr of B$ text in DX.
      .
      .
      .
MOV  SI,8[BP]     ;Get address of 'A' in SI.
MOV  DI,4[BP]     ;Get pointer to 'C' in DI.
MOVS WORD         ;Store variable 'A' in 'C'.
RET  6            ;Restore Stack, return.
```

BEWARE!: the called program must know the variable type for numeric parameters passed. In the above example, the instruction; MOVS WORD will copy only 2 bytes. This is fine if variables A and C are Integer. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

4.5 CHAIN - Statement

Format: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVERLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVERLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for any use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

4.6 CIRCLE - Statement

Format: CIRCLE (x,y),r[,color[,start,end[,aspect]]]

Purpose: To draw an ellipse on the screen with center (x,y) and radius r

Remarks:

(x,y)	are the coordinates of the center of the ellipse. The coordinates may be given in either absolute or relative form.
r	is the radius (major axis) of the ellipse in points.
color	is a number which specifies the color of the ellipse, in the range 0 to 3. In medium resolution, color selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, number 3. In high resolution, a color of 0 indicates black, and the default of 1 indicates white.
start, end	are angles in radians and may range from -2*PI to 2*PI, where PI=3.141593.
aspect	is a numeric expression.

start and end specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise.

If the start or end angle is negative the ellipse will be connected to the center point with a line, and the angles will betreated as if they were positive (note that this is not the same as adding 2*PI). The start angle may be greater or less than the end angle.

The aspect ratio affects the ratio of the x-radius to the y-radius. For example, the IBM aspect ratio of 2/5 in high resolution indicates that the vertical axis of the screen is 2/5 as long as the horizontal axis. This ratio is not equal to 640/200 simply because pixels are higher than they are wide. The default aspect ratios for each of the four graphic modes are as follows:

MODE	Screen	Ratio
IBM medium resolution (320 x 200)	1	4/5
IBM high resolution (640 x 200)	2	2/5
Hyperion medium resolution (320 x 250)	101	1
Hyperion high resolution (640 x 250)	102	1/2

If the aspect ratio of 2/5 is used with IBM high resolution (screen=2), then the 2 vertical pixels have the same length as the 5 horizontal pixels. For example:

```
CIRCLE (100,120),20,1,0,6.28,2/5
```

produces a perfect circle with its centre at (100,120). Changing the aspect ratio from 2/5 to a number greater than 1 would produce an elipse with the radius measured in points in the vertical direction. When the aspect is less than one, the radius given is the x-radius, i.e. the radius is measured in horizontal pixels. If the aspect ratio is greater than one the radius is measured in vertical pixels.

```
CIRCLE (200,90),35,3,,,1/3
```

produces an elipse with a horizontal radius of 35 pixels.

4.7 CLOSE - Statement

Format: CLOSE[[#]<file number>[,]#]<file number...>]]

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Appendix B.

4.8 CLS - Statement

Format: CLS

Purpose: The CLS statement erases the current active screen page. (See the Screen statement.)

Rules:

1. If the Screen is in Alpha Mode, the active page is cleared to the currently selected Background Color. (See the COLOR statement).

2. If the Screen is in Graphics medium or Hi-res mode, the entire Screen buffer is cleared to Black.

3. The Screen may be also be cleared by depressing the Ctrl+L or Ctrl+Home keys.

4. NOTE: The SCREEN and WIDTH statements will force a Screen clear if the resultant Screen mode created is different than the mode currently in force.

Example: 1 CLS 'Clears the screen.

4.9 COLOR - Statement (Alpha Mode)

Format: COLOR [foreground] [,[background]]

Purpose: The COLOR statement selects the Foreground, Background and Border screen display colors.

Remarks: If the Screen Board Imitation setting in the DOS MODE command is set to COLOR Board imitation:

Foreground color:	0	-black
	1-15	-white

Background color:	0	-black
	1-7	-white

If the Screen Board Imitation is set to Monochrome Board imitation with IBM alpha mode (SCREEN 0):

Foreground Color	Font
0,2-7,32,34-39,64,66-71	White on Black
1,33,65	Underlined
8,10-15,40,42-47,72,74-79	Intensified
9,41,73	Underlined, Intensified
16,18-23,48,50-55,80,82-87	Blinking
17,49,81	Blinking, Underlined
24,26-31,56,58-63,88,90-95	Blinking, Intensified
25,57,89	Blinking, Underlined, Intensified

If the Screen Board Imitation is set to Monochrome Board imitation with Hyperion alpha mode (SCREEN 100):

Foreground Text Color	Font
0,2-7	White on Black
1	Underlined
8,10-15	Intensified
9	Underlined, Intensified
16,18-23	Blinking
17	Blinking, Underlined
24,26-31	Blinking, Intensified
25	Blinking, Underlined, Intensified
Add 32 to the above numbers	Superscripted
Add 64 to the above numbers	Subscripted

4.10 COLOR - Statement (Graphic Mode)

Format: COLOR [background]

where: background = 0 (black), 1-7 (dark grey), 8-14 (light grey), 15 (white).

Other parameters are ignored and do not return errors.

Purpose: The COLOR statement is used in medium resolution graphics to set the background color.

Remarks: The foreground color is the last used foreground color or an explicit setting in PSET, PRESET, LINE, CIRCLE, PAINT, or DRAW, and ranges 0 for black, 1 for dark grey, 2 for light grey, and 3 for white.

In graphics, the COLOR statement has meaning for medium resolution only (SCREEN 1 or SCREEN 101). Attempts to use COLOR in high resolution (SCREEN 2 or SCREEN 102) will result in an "Illegal function call" error.

Any values entered outside the range o to 255 will result in an "Illegal Function call" error. Previous values will be retained.

4.11 COM(n) - Statement

Format: COM(n) ON
COM(n) OFF
COM(n) STOP

Purpose: Enables or disables trapping of communcations activity to the specified communications adapter.

Remarks: n is the number of the communications adapter (1 of 2)

A COM(n)ON statement must be executed to allow trapping by the ON COM(n) statement. After COM(n)ON, if a non-zero line number is specified in the ON COM(n) statement, BASIC checks to see if any characters have come in to the communications adapter every time a new statement is executed.

If COM(n) is OFF, no trapping takes place and any communication activity is not remembered even if it does take place.

If a COM(n)STOP statement has been executed, no trapping can take place. However, any communications activity that does take place is remembered so that an immediate trap occurs when COM(n)ON is executed.

4.12 COMMON - Statement

Format: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
       .
       .
       .
```

4.13 CSRLIN - Variable

Format: x = CSRLIN

Function: The CSRLIN function returns the current line (or row) position of the cursor.

Rules: x is a numeric variable receiving the value returned. The value returned will be in the range 1 to 24.

x = POS(0) will return the column location of the cursor. A value in the range 1 to 40 or 1 to 80 depending upon the current WIDTH.

Example:

```
10 Y = CSRLIN  'Record current line.
20 X = POS(0)  'Record current column.
30 LOCATE 24,1 :PRINT "HELLO" 'Print HELLO on last line.
40 LOCATE Y,X  'Restore position to old line, column.
```

4.14 DATA - Statement

Format: DATA <list of constants>

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section I-4.)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

See Also: See examples in READ.

DATA statements may be reread from the beginning by use of the RESTORE statement.

4.15 DATE$ - Variable and Statement

Format: As a variable:

```
x$ = DATE$
```

As a statement:

```
DATE$ = x$
```

Purpose: Sets or retrieves the date.

Remarks: For the variable (v$ = DATE$):

As 10-character string of the form mm-dd-yyyy is returned. Here, mm represents two digits for the month, dd is the day of the month (also 2 digits), and yyyy is the year. The date may have been set by DOS prior to entering BASIC.

For the statement (DATE$ = X$):

x$ is a string expression which is used to set the current date. You may enter x$ in any one of the following forms:

```
mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy
```

The year must be in the range 1980 to 2099. If you use only one digit for the month or day, a 0 (zero) is assumed in front of it. If you give only one digit for the year, a zero is appended to make it two digits. If you give only two digits fro the year, the year is assumed to be 19yy.

Example:

```
Ok
10 DATE$= "8/17/82"
20 PRINT DATE$
RUN
08-17-1982
Ok
```

In the example we set the date to August 17th, 1982. Notice how, when we read the date back using the DATE$ function, a zero was included in from of the month to make it two digits, and the year became 1982. Also, the month, day, and year are separated by hyphens even though we entered them as slashes.

Caution: Changing DATE$ within BASIC resets the Hyperion's internal clock. This should be avoided. See the DATE command in the Hyperion User Guide for more information.

4.16 DEF FN - Statement

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Purpose: To define and name a function that is written by the user.

Remarks: **<name>** must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If is does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
      .
      .
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
      .
      .
```

Line 410 defines the function FNAB. The function is called in line 420.

4.17 DEFINT/SNG/DBL/STR - Statements

Format: DEF<type> <range(s)>

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes all variables without declaration characters are single precision variables.

Examples:

10 DEFDBL L-P All variables beginning with the letters L, M, N, O, and P will be double precision variables.

10 DEFSTR A All variables beginning with the letter A will be string variables.

10 DEFINT I-N,W-Z All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

4.18 DEF SEG - Statement

Format: DEF SEG [=<address>]

Purpose: The DEF SEG statement assigns the current value to be used by a subsequent BLOAD, BSAVE, PEEK, POKE, CALL, or user defined function call.

Remarks: **<address>** is a valid numeric expression returning an unsigned Integer in the range 0 to 65535.

The address specified is saved for use as the segment required by the BLOAD, BSAVE, PEEK, POKE and CALL statements.

Rules:

1. Any value entered outside of this range will result in an "Illegal Function Call" Error. The previous value is retained.

2. If the address option is omitted, the segment to be used is set to Basic's Data Segment. This is the initial default value.

3. If the address option is given, it should be a value based upon a 16 byte boundary. For the BLOAD, BSAVE, PEEK, POKE, or CALL statements, the value is shifted left 4 bits to form the Code Segment address for the subsequent call instruction. BASIC does not perform additional checking to ensure that the resultant segment + offset value is valid.

4. NOTE: DEF and SEG MUST be separated by a space! Otherwise, Basic would interpret the statement; DEFSEG=100 to mean: "assign the value 100 to the variable DEFSEG".

Example:

```
10 DEF SEG=&HB800   'Set segment to Screen buffer.

20 DEF SEG          'Restore segment to BASIC's DS.
```

4.19 DEF USR - Statement

Format: DEF USR[<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine, which is later called by the USR function.

Remarks: **<digit>** may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
      .
      .
      .
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
      .
      .
      .
```

4.20 DIM - Statement

Format: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statment, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section ???).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)        '21 elements from 0 to 20
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
     .
     .
     .
```

4.21 DRAW - Statement

Format: DRAW <string>

Purpose: Draws an object as specified by <string>

Remarks: You use the DRAW statement to draw using a "graphics definition language". The language commands are contained in the string expression <string>. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of <string> and interprets single letter commands from the contents of the string. These commands are detailed below:

The following movement commands begin movement from the last point referenced. After each command, the last point referenced is the last point the command draws.

U n	Move up.
D n	Move down.
L n	Move left.
R n	Move right.
E n	Move diagonally up and right.
F n	Move diagonally down and right.
G n	Move diagonally down and left.
H n	Move diagonally up and left.

n in each of the preceding commands indicates the distance to move. The number of points moved is n times the scaling factor (set by the S command).

M x,y Move absolute or relative. If x has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

B Move, but do not plot any points.

N Move, and return to original position when finished.

A n Set angle n. n may be from 0 to 3 (0 is 0 degrees, 1 is 90, 2 is 180, 3 is 270).

C n Set color n. n may be from 0 to 3 in medium resolution, and from 0 to 1 in high resolution. In medium resolution, n selects the color from the current palette as defined by COLOR. Background is 0, default is foreground color number 3. In high resolution, 0 is black and 1 (default) indicates white.

S n
Sets scale factor. n may be from 1 to 255; n divided by four is the scale factor. The scale factor multiplied by the distances given with U, D, L, R, E, F, G, H, and M gives the actual distance moved. The default value is 4, so the scale factor is 1.

X variable
Executes substring, allowing a second string from within a string.

The aspect ratio of the screen determines the spacing of the horizontal, vertical and diagonal points. For example, the IBM standard aspect ratio of 4/5 in medium resolution, indicates that the vertical axis of the screen is 4/5 as long as the horizontal axis. This information can be used to determine how many vertical points are equal in length to how many horizontal points. The default aspect ratios from each of the four graphic modes are as follows:

Mode	Screen	Ratio
IBM medium res. (320 x 200)	1	4/5
IBM high res. (640x200)	2	2/5
Hyperion medium res. (320x250)	101	1
Hyperion high res. (640x250)	102	1/2

The aspect ratio of 4/5 indicates that 4 vertical pixels have the same length as 5 horizontal pixels.

For example, to draw a square box with 20 horizontal pixels, it would require 20 x (4/5) or 16 vertical pixels. That is:

```
DRAW "U16R2;0D16L20"
```

produced a square in IBM medium resolution (screen 1)

```
DRAW "U10R20D10L20"
```

produces a square in Hyperion high resolution (screen 102).

4.22 END - Statement

Format: END

Purpose: To terminate program execution, close all files, and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

4.23 ERASE - Statement

Format: ERASE <list of array variables>

Purpose: To eliminate arrays from a program

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Example:

```
300 DIM B(150),A(12)
      .
      .
      .
450 ERASE A,B
460 DIM B(99)
      .
      .
      .
```

4.24 ERR & ERL - Variables

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in the LET (assignment) statement. BASIC's error codes are listed in Appendix C.

The ERROR statement can be used to assign user-defined error codes to the ERR variable.

4.25 ERROR - Statement

Format: ERROR <integer expression>

Purpose:

1) To simulate the occurrence of a BASIC error; or
2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC, the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC's error codes. These are listed in Appendix C. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error mesage has been defined, BASIC responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15            (you type this line)
String too long     (BASIC types this line)
Ok
```

Example 2:

```
      .
      .
      .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
      .
      .
      .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
      .
      .
      .
```

4.26 FIELD - Statement

Format: FIELD[#]<file number>,<field width> AS <string variable>...

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

4.27 FOR...NEXT - Statements

Format:

```
FOR <variable>=x TO y [STEP x]
        .
        .
        .
NEXT [<variable>][,<variable>...]
```

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I=5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

4.28 GET - Statement (Files)

Format: GET [#]<file number>[,<record number>]

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Appendix B.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer. If a FIELD statement was used to assign variable names to the random buffer, these names should not be used in the subsequent INPUT statement.

4.29 GET - Statement (Graphics)

Format: GET(x1,y1)-(x2,y2),<arrayname>

Purpose: Reads points from an area of the screen.

Remarks: (x1,y1)(x2,y2)
are coordinates in either absolute or relative form.

<arrayname> is the name of the array you want to hold the information.

GET reads the colors of the points within the specified rectangle into the array. The specified rectangle has points (x1,y1) and (x2,y2) as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the B option.)

GET and PUT can be used for high speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations which move bits onto (PUT) and off of (GET) the screen.

PUT and GET are also used for random access files, but the syntax of the file-oriented statements is different.

The array is used simply as a place to hold the image and must be numeric; it may be any precision, however. The required size of the array, in ;bytes, is:

```
4+INT((x*<bitsperpixel+7)/8)*y
```

where x and y are the lengths of the horizontal and vertical sides of the rectangle, respectively. The value of <bitsperpixel> is 2 in medium resolution, and 1 in high resolution.

For example, suppose we want to use the GET statement to get a 10 by 12 image in medium resolution. The number of bytes required is 4+INT((10*2+7)/8)*12, or 40 bytes. The bytes per element of an array are:

* 2 for integer
* 4 for single-precision
* 8 for double-precision

Therefore, we could use any integer array with at least 20 elements.

The information from the screen is stored in the array as follows:

1. two bytes giving the x dimension in bits
2. two bytes giving the y dimension in bits
3. the data itself

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The x dimension is in element 0 of the array, and the y dimension is in element 2. Keep in mind, however, that integers are stored low byte first, then high byte; but the data is actually transferred high byte first, then low byte.

The data for each row of points in the rectangle is left justified on a byte boundary, so if there are less than a multilple of eight bits stored, the rest of the byte will be filled with zeros.

PUT and GET work significantly faster in medium resolution when x1 MOD 4 is equal to zero, and in high resolution when x1 MOD 8 is equal to zero. This is a special case where the rectangle boundaries fall on the byte boundaries.

4.30 GOSUB...RETURN - Statements

Format:

```
GOSUB <line number>
      .
      .
      .
RETURN
```

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs a program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE"
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

4.31 GOTO - Statement

Format: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: <line number> must exist, or an "Undefined line number" error will be returned. If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5         AREA = 78.5
R = 7         AREA = 153.86
R = 12        AREA = 452.16
?Out of data in 10
Ok
```

4.32 IF...THEN[...ELSE] and IF...GOTO

Format: IF <expression> THEN <statement(s) or <line number> [ELSE <statement(s) or <line number>]

Format: IF <expression> GOTO <line number> [ELSE <statement(s) or <line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. BASIC allows a command before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT
"LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C" ELSE PRINT
"A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

this test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2:

```
100 IF(I<20)AND(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go to either the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

4.33 INKEY$ - Variable

Format: INKEY$

Purpose: To read a single character from the keyboard.

Action: Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through to the program except for Ctrl+Brk, which terminates the program.

Example:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT$=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

Remarks: See Appendix A for a list of extended keyboard scan codes that can be read into a two-byte INKEY$ variable.

4.34 INPUT - Statement

Format: INPUT[;][<"prompt string">;]<list of variables>

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the termnial.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Example:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?  5    (The 5 was typed in by the user
          in response to the question mark.)
 5 SQUARED IS 25
Ok
```

```
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

4.35 INPUT# - Statement

Format: INPUT#<file number>,<variable list>

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The dtaa items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC is canning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Appendix B.

4.36 KEY - Statement

Format:

```
KEY <key number>,<string expression>
KEY LIST
KEY ON
KEY OFF
```

Purpose: The KEY statement allows Function keys to be designated "Soft Keys".

Remarks: Any one or all of the ten Special Function Keys may be assigned a 15 byte string which, when the Key is depressed, will be input to Basic.

Initially, the Soft Keys are assigned the following values:

F1 - LIST	F2 - RUN
F3 - LOAD	F4 - SAVE
F5 - CONT	F6 - ,"LPT1:"
F7 - TRON	F8 - TROFF
F9 - KEY	F10 - SCREEN 0,0,0

<key number> is the Key number. An Expression returning an unsigned Integer in the range 1 to 10.

<string expression> is the Key assignment text. Any valid string expression.

KEY ON This is the initial setting. Causes the Key values to be displayed on the 25th Line. When the Width is 40, 5 of the 10 Soft Keys are displayed. When the width is 80, all 10 are displayed. In either width, only the first 7 characters of each value are displayed.

KEY OFF Erases the Soft Key display from the 25th line.

KEY LIST Lists all 10 Soft Key values on the screen. All 15 characters of each value are displayed.

KEY <key number>,<string expression>
Assigns the string expression to the Soft Key specified (1 to 10).

Rules:

1. If the ~~value~~ returned for <key number> is not in the range 1 to 10, an "Illegal Function Call" Error is taken. The previous Key string assignment is retained.

2. The Key assignment string may be 1 to 15 characters in length. If the string is longer than 15 characters, the first 15 characters are assigned.

3. Assigning a null string (string of length 0) to a Soft Key disables the Function Key as a Soft Key.

4. When a Soft Key is assigned, the INKEY$ function returns one character of the soft key string per invocation. If the Soft Key is disabled, INKEY$ returns a string of length 2. The first character is binary zero, the second is the Key Scan Code.

Example:

50 KEY ON
Display the Soft Keys on the 25th Line.

200 KEY OFF
Erase Soft Key display.

10 KEY 1,"MENU"+CHR$(13)
Assigns the string 'MENU'<carriage return> to soft key 1. Such assignments might be used for rapid data entry. This example might be used in a program to select a menu display when entered by the user.

```
20 KEY 1,""
```

Would erase Soft Key 1.

The following routine initializes the first 5 soft keys:

```
1 KEY OFF 'Turn off key display during init.
10 DATA KEY1,KEY2,KEY3,KEY4,KEY5
20 FOR I=1 TO 5:READ SOFTKEYS$(I)
30 KEY I,SOFTKEYS$(I)
40 NEXT I
50 KEY ON 'now display new softkeys.
```

4.37 LET - Statement

Format: [LET] <variable>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
     .
     .
     .

    or

110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
     .
     .
     .
```

4.38 LINE - Statement

LINE is the most powerful of the graphics statement. It allows a group of pixels to be controlled with a single statement.

Format: LINE [(x1,y1)] -(x2,y2),attribute[,b[f]]

Remarks: The simplest form of line is:

```
LINE -(x2,y2),attribute
```

This will draw from the last point to the point (x2,y2) in the foreground attribute.

We can include a starting point also:

```
LINE (0,0)-(319,199),1   'draw diagonal line down screen
LINE (0,100)-(319,100),1 'draw bar across screen
```

We can append the attribute to draw the line in:

```
LINE (10,10)-(20,20),2   'draw in color 2!
```

```
10 CLS
20 LINE -(rnd*319,rnd*199),rnd*3
30 GO TO 20  'draw lines forever using random attribute
```

```
10 FOR x=0 TO 319
20 LINE (x,0)-(x,199),x AND 1
30 NEXT
```

(draw alternating pattern - line on line off)

The final argument to line is ",b" -- box or ",bf" filled box. The syntax indicates we can leave out the attribute argument and include the final argument as follows:

```
LINE (0,0)-(100,100),1,b   'draw box in foreground
```

or include it:

```
LINE (0,0)-(200,200),2,bf  'filled box attribute 2
```

The ",b" tells BASIC to draw a rectangle with the points (x1,y1) and (x2,y2) as opposite corners. This avoids giving the four LINE commands:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
```

```
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

which perform the equivalent function.

The ",bf" means draw the same rectangle as ",b" but also fill in the interior points with the selected attribute.

When out of range coordinates are given the line command the coordinate which is out of range is given the closest legal value. In other words, negative values become zero, y values greater than 199 become 199 and x values greater than 319 in medium res become 319 and greater than 639 in hi resolution become 639.

In the examples and syntax the coordinate form STEP(xoffset,yoffset) is not shown. However this form can be used wherever a coordinate is used. Note that all of the graphics statements and functions update the "more recent point used". In a line command if the relative form is used on the second coordinate it is relative to the first coordinate. The only other way "the most recently used" point is changed is that SCREEN and CLS initialize it to be the point in the middle of the screen (160,100) for medium and (320,100) for hi resolution.

The graphics commands have been fully optimized to take advantage of the 8088. They are significantly faster than other machines.

Last Example:

```
10 CLS
20 LINE -(rnd*639,rnd*199),rnd*2,bf
30 GO TO 20
```

4.39 LINE INPUT - Statement

Format: LINE INPUT[;]["<prompt>";]<stringvar>

Purpose: Reads an entire line (up to 254 characters) from the keyboard into a string variable, ignoring delimiters.

Remarks: "<prompt>" is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

<stringvar> is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the <Rtn> is assigned to <stringvar>. Trailing blanks are ignored.

If LINE INPUT is immediately followed by a semicolon, then pressing <Rtn> to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing <Ctrl+Brk>. BASIC returns to command level and displays Ok. You may then enter CONT to resume execution at the LINE INPUT.

Example: See example in "LINE INPUT# - Statement"

4.40 LINE INPUT# - Statement

Format: LINE INPUT#<file number>,<string variable>

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES   234,4   MEMPHIS
LINDA JONES   234,4   MEMPHIS
Ok
```

4.41 LOCATE - Statement

Format: LOCATE [row] [, [col] [, [cursor] [, [start] [,stop]]]]

Purpose: The LOCATE statement moves the Cursor to the specified position on the active Screen. Optional parameters turn the blinking cursor on and off and define the start and stop raster lines for the cursor.

Remarks

row — Is the Screen Line number. A numeric expression returning an unsigned Integer in the range 1 to 24.

col — Is the Screen Column number. A numeric expression returning an unsigned Integer in the range 1 to 40 or 1 to 80, depending upon Screen Width.

cursor — Is a boolean value indicating whether the cursor is visible or not. 0 for off, non-zero for on.

start/stop — Is the cursor starting and ending scan lines. If start = stop, cursor becomes invisible (stop-start <1). If stop - start = 1, cursor becomes underbar. If stop - start > 1, cursor becomes block.

stop — Is the cursor stop scan line. A numeric expression returning an unsigned Integer in the range 0 to 31.

Action: Moves the cursor to the specified position. subsequent PRINT statements begin placing characters at this location. Optionally may be used to turn the blinking cursor on or off, or change the size of the blinking cursor.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" Error. Previous values are retained.

2. Any parameter may be omitted. Omitted parameters assume the old value.

3. If the start scan line parameter is given and the stop scan line parameter is omitted, stop assumes the start value. This produces a single scan line cursor.

4. Cursor Blink is not selectable and always blinks 16 times a second.

5. The 25th line is reserved for Soft Key display and may not be written over, even if Soft Key display is Off.

Example:

10 LOCATE 1,1	Moves to the home position in the upper left hand corner.
20 LOCATE ,,1	Make the blinking cursor visible, position remains unchanged.
30 LOCATE 5,1,1,0,7	Move to Line 5, column 1, turn cursor on, cursor will cover entire character cell starting at scan line 0 and ending on scan line 7.

4.42 LPRINT and LPRINT USING - Statements

Format: LPRINT [<list of expressions>][;]
LPRINT USING <string exp>;<list of expressions>[;]

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer (the PRN device).

LPRINT assumes a 132-character-wide printer.

For a description of the <string exp> parameter of the LPRINT USING statement see the PRINT USING statement.

4.43 LSET and RSET - Statements

Format: LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra position.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions.

Example:

```
150 LSET A$=MKS$(AMT)
```

See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

4.44 ON COM(n) - Statement

Format: ON COM(n) GOSUB<line>

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

Remarks: n is the number of the ocmmunication adapter (1 of 2).

<line> is the line number of the beginning of the trap routine. Setting <line equal to 0 (zero) disables trapping of communications activity for the specified adapter.

A COM(n) ON statement must be executed to activate this statement for adapter n. After COM(n) ON, if a non-zero line number is specified in the ON COM(n) statement then every time the program starts a new statement, BSIC checks to see if any characters have come in to the specified communications adapter. If so, BASIC performs a GOSUB to the specified <line>.

If COM(n) OFF is executed, no trapping takes place for the adapter. Even if communications activity does take place, the event is not remembered.

If a COM(n) STOP statement is executed, no trapping takes place for the adapter. However, any characters being received are remembered so an immediate trap takes place when COM(n) ON is executed.

When the trap occurs an automatic COM(n) STOP is executed so recursive traps can never take place.

The RETURN from the trap routine automatically does a COM(n) ON unless an explicit COM(n) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Typically the communications trap routine reads an entire message from the communications line before returning back. It is not recommended that you use the communications trap for single character messages since at high baud reates the overhead of trapping and reading for each individual character may allow the interrupt buffer for communication to overflow.

You may use RETURN<line> if you wnat to go bak to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
      .
      .
      .
500 REM incoming characters.
      .
      .
      .
590 RETURN 300
```

This example sets up a trap routine for the first communications adapter at line 500.

4.45 ON ERROR GOTO - Statement

Format: ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, and "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occr within the error handling subroutine.

Example: 10 ON ERROR GOTO 1000

4.46 ON...GOSUB and ON...GOTO - Statements

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

4.47 ON KEY - Statement

Format: ON KEY(n)GOSUB<line>

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Remarks: n is a numeric expression in the range 1 to 14 indicating the key to be trapped, as follows:

1-10 function keys F1-F10
11 Cursor Up
12 Cursor Left
13 Cursor Right
14 Cursor Down

<line> is the line number of the beginning of the trapping routine for the specified key. Setting <line> equal to 0 disables trapping of the key.

A KEY(n)ON statement must be executed to activate this statement. After KEY(n)ON, if a non-zero line number is specified in the ON KEY(n) statement then every time the program starts a new statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified <line>.

If a KEY(n)OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(n)STOP statement is executed, no trapping takes place for the specified key. However, it the key is pressed the event is remembered, so an immediate trap takes place when KEY(n)ON is executed.

When the trap occurs an automatic KEY(n)STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a KEY(n)ON unless an explicit KEY(n)OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting form an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, COM(n), and KEY(n)).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You may use RETURN<line> if you want to go back to the BASIC program at a fixed line number. Use this non-local return with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(n)ON has no effect on whether the softkey values are displayed at the bottom of the screen.

Example: The following is an example of a trap routine for function key 5.

```
100 ON KEY(5) GOSUB 200
110 KEY(5) ON
    .
    .
    .
200 REM function key 5 pressed
    .
    .
    .
290 RETURN 140
```

4.48 OPEN - Statement

Format: OPEN [<dev>] <filename> [FOR <mode>] AS [#]<file number> [LEN=<lrecl>]

Purpose: To establish addressability between a physical device and an I/O buffer in the data pool.

Remarks: <dev> is optionally part of the filename string and may be one of the following:

A:	Drive A
B:	Drive B
C:	Ram disk
D:	Hard disk
PRN	Line Printer - Output Only.
CON	Screen - Output Only
KYBD:	Keyboard - Input Only
SCRN:	Screen - Output Only
LPT1:	Line Printer Output Only
COM1:	RS232 serial communications - Input, Output, or random only.

<filename> is a valid string literal or variable optionally containing a <dev>. If <dev> is omitted, disk A: is assumed. Refer to "DISK FILES" for naming conventions.

<mode> determines the initial positioning within the file and the action to be taken if the file does not exist. The valid modes and actions taken are:

INPUT — Position to the beginning of an existing file. A "File not found" error is given if the file does not exist.

OUTPUT — Position to the beginning of the file. If the file does not exist, one is created.

APPEND — Position to the end of the file. If the file does not exist, one is created.

DEFAULT — If the FOR <mode> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the Random I/O mode. That is, records may be read or written at will at any position with the file.

<file number> is an integer expression returning a number in the range 1 thru 15. The number is used to associate an I/O buffer with a disk file or device. This association exists until a CLOSE or CLOSE <file number> statement is executed.

<lrecl> is an integer expression in the range 2 to 32768. This value sets the record length to be used for random files (see the FIELD statement). If omitted, the record length defaults to 128 byte records.

Action: When a disk file is OPENed FOR APPEND, the position is initially at the end of the file and the record number is set to the last record of the file (LOB(x)/128). PRINT, WRITE, or PUT will then extend the file. The program may position elsewhere in the file with a GET statement. If this is done, the mode is changed to random and the position moves to the record indicated.

Once the position is moved from the end of the file, additional records may be appended to the file by exeucting a GET #x,LOF(x)/<lrecl> statement. This positions the file pointer at the end of the file in preparation for appending.

Rules:

1. Any values entered outside of the ranges given will result in an "Illegal Function Call" error. The file is not opened.

2. If the file is opened as INPUT, attempts to write to the file will result in a "Bad File Mode" error.

3. If the file is opened as OUTPUT, attempts to read the file will result in a "Bad File Mode" error.

4. At any one time, it is possible to have a particular disk filename OPEN under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, to use different file numbers for different modes of access. Each file number has a different buffer, so several records from the same file may be kept in memory for quick access.

   A file may NOT be opened FOR OUTPUT, however, on more than one file number at a time.

5. If the LEN=<lrecl> option is used, lrecl may not exceed the value set by the /S:<lrecl> switch option to the command.

Examples:

```
10 OPEN "A:MYDATA" FOR OUTPUT AS #1
10 OPEN "KYBD:" FOR INPUT AS #2
10 OPEN "B:INVENT.DAT" FOR APPEND AS #1
10 OPEN "C:QUICK" AS #1 'for random I/O on RAM disk
```

4.49 OPEN "COM1:" - Statement

Format: OPEN "COM1:<speed>,<parity>,<data>,<stop>" AS [#] <file number>

Purpose: Allocates a buffer for I/O in the same manner as OPEN for disk files.

Remarks:

COM1: Is the name of the Hyperion serial communications device.

speed Is a literal integer specifying the transmit/receive baud rate. Valid speeds are: 110, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200.

parity Is a one character literal specifying the Parity for Transmit and Receive as follows:

- O ODD, Odd Transmit/Receive Parity checking.
- E EVEN, Even Transmit/Receive Parity checking.
- N NONE, No Transmit Parity, No Receive Parity checking.

data Is a literal integer indicating the number of transmit/receive data bits. Valid values are: 7 or 8.

NOTE: 8 data bits with any parity is illegal.

stop Is a literal integer indicating the number of stop bits. Valid values are 1 or 2. If omitted then 110 bps transmits two stop bits, all others transmit one stop bit.

file number Is an integer expression returning a valid file number. The number is then associated with the file for as long as it is OPEN and is used to refer other COM I/O statements to the file.

DEFAULTS: Missing parameters invoke the following defaults:
speed - 300 bps, parity - EVEN, data - 7, stop -2 if 110 bps else 1.

NOTE: The COM1: device may be OPENed to only one file number at a time.

Any coding errors within the File Name String will result in a "Bad File Name" error. No indication as to which parameter is in error is given.

A "Device Timeout" error will occur if Data Set Ready (DSR) is not detected. Refer to hardware documentation for proper cabling instructions.

Example:

```
10 OPEN "COM1: " AS 1
```

File 1 is opened for communication with all defaults. Speed at 300 bps, Even Parity, 7 data bits, and one stop bit.

```
20 OPEN "COM1:2400 " AS #2
```

File 2 is opened for communication at 2400 bps. Parity and number of data bits are defaulted.

```
10 OPEN "COM1:1200,N,8" AS #1
```

File number 1 is opened for Asynchronous I/O at 1200 bits/second, no parity is to be produced or checked, and 8 bit bytes will be sent and received.

Communications I/O

Since the communication port is opened as a file, all Input/Output statements that are valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files. They are: INPUT#<file number>, LINE INPUT#<file number>, and the INPUT$ variable.

COM sequential output statements are the same as those for disk, and are: PRINT#<file number>, and PRINT#<file number> USING.

Refer to INPUT and PRINT sections for details of coding syntax and usage.

COM I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates above 2400 bps., it is necessary to suspend character transmission from the Hyperion long enough to "catch up". This can be done by sending XOFF (Ctrl+NumLock) to the host and XON (any key) when ready to resume.

BASIC provides three functions which help in determining when an "over-run" condition is eminent. These are given below, where x is the file number specified.

LOC(x) Returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the queue, LOC(x) returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 255 characters remain in the queue, LOC(x) returns the actual count.

LOF(x) Returns the amount of free space in the input queue. That is, /C:<size>-LOC(x). Use of LOF may be used to detect when the input queue is getting full. In practicality, LOC is adequate for this purpose.

EOF(x) If true (-1), indicates that the input queue is empty. Returns false (0) if any characters are waiting to be read.

4.50 OPTION BASE - Statement

Format: OPTION BASE n
where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

4.51 OUT - Statement

Format: OUT I,J
where I and J are integer expressions in the range 0 to 65535. I is a machine port number, and J is the data to be transmitted.

Purpose: To send a byte to a machine output port.

Remarks: OUT is the complementary statement to the INP function.

Example:

```
100 OUT 12345,225
```

In assembly language, this is equivalent to:

```
MOV  DX,12345
MOV  AL,225
OUT  DX,AL
```

4.52 PAINT - Statement

Format: PAINT(x,y)[,<paint>[,<boundary>]]

Purpose: Fills in an area on the screen with the selected color. Only used in graphics modes (Screen 1,2,101 or 102).

Remarks: (x,y) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form. This point will be used as a starting point.

<paint> is the color to be painted with, in the range 0 to 3. In medium resolution, this color is the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, <paint> equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

<boundary> is the color of the edges of the figure to be filled in, in the range 0 to 3 as described above.

The figure to be filled in is the figure with edges of <boundary> color. The figure is filled in with the color <paint>.

Since there are only two colors in high resolution it doesn't make sense for <paint> to be different from <boundary>. Since <boundary> is defaulted to equal <paint> we don't need the third parameter in high resolution mode.

In high resolution this means "blacking out" an area until black is hit, or "whiting out" an area until white is hit.

In medium resolution we can fill in with color 1 with a border of color 2.

The starting point of PAINT must be inside the figure to be painted. If the specified point already has the color <boundary> then PAINT will have no effect. If <paint> is omitted the foreground color is used (3 in medium resolution, 1 in high resolution). PAINT can paint any type of figure, but "jagged" edges on a figure will increase the amount of stack space required by PAINT. So if a lot of complex painting is being done you may want to use CLEAR at the beginning of the program to increase the stack space available.

The PAINT statement allows scenes to be displayed with very few statements. This can be a very useful capability.

Example:

```
5 SCREEN 1
10 LINE (0,0)-(100,150),2,B
20 PAINT (50,50),1,2
```

The PAINT statement in line 20 fills in the box drawn in line 10 with color 1.

4.53 PLAY - Command

Format: PLAY <string expression>

REMARKS: PLAY implements a concept similar to DRAW by embedding a string expression into the string data type.

The single character commands in PLAY are:

A-G [#,+,-] Play the note. A "#" or "+" afterwards means sharp, and "-" means flat.

L <n> Length - Sets the length of each note. L4 is a quarter note, L1 is a whole note, etc. n may range from 1 to 64

The length may also follow the note when it is desired to change the length only for the note. In this case, A16 is equivalent to L16A.

MF Music Foreground. Music (PLAY statement) and SOUND are to run in Foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. This is the initial default.

MB Music Background. Music (PLAY statement) and SOUND are to run in Background. That is, each note or sound is placed in a buffer allowing the BASIC program to continue execution while music plays in the background. Up to 32 notes (or rests) can be played in background at a time.

MN Music Normal. Each note will play 7/8ths of the time determined by L (length).

ML Music Legato. Each note will play the full period set by L (length).

MS Music Staccato. Each note will play 3/4ths of the time determined by L (length).

N <n>	Play note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. N=0 means rest.
O <n>	Octave - Sets the current octave. There are 7 octaves (0...6).
P <n>	Pause. P may range from 1 to 64.
T <n>	Tempo - Sets the number of L4's in a second. n may range from 212 to 255. Default is 120.
	Dot or Period. After each note causes the note to play 3/2 times the period determined by L (length) times T (tempo). Multiple dots may appear after a note. The period is scaled accordingly. (Example: A. 3/2, A.. 9/4, A... 27/8 etc.). Dots may appear after a pause (P) and scale the pause length as described above.
X <string>	Execute substring
	Because of the slow clock interrupt rate, some notes will not play at higher tempos; e.g. L64 at T255. Which note/temp0o combinations these are must be determined through experimentation.

4.54 POKE - Statement

Format: POKE I,J
where I and J are integer expression

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 1 to 255. I must be in the range 0 to 65536.

Data may be POKEd into memory locations above 32768 by supplying a negative number for I. The value of I is computed by subtracting 65536 from the desired address. For example, to POKE data into location 45000, I = 45000-65536, or -20536

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
10 POKE &H5A00,&HFF
```

4.55 PRINT - Statement

Format: PRINT [<list of expressions>][;]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or semicolon terminates the list of expression, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are always preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10^-7 is output as .0000001 and 10^-8 is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:
```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0             -25           3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:
```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUAREd IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to printed before the next prompt.

Example 3:
```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

4.56 PRINT USING - Statement

Format: PRINT USING <string exp>;<list of expressions>[;]

Purpose: To print strings or numbers using a specified format.

Remarks: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

<u>String Fields</u>

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
40 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20    5.30    66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

\+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus of minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7.01
68.95-   22.45    7.01-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4   *-0.9   765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential

(^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50
```

```
PRINT USING "####.##,";1234.5
1234.50,
```

^^^^ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E02
```

```
PRINT USING ".####^^^^-";888888
 .8889E+06
```

```
PRINT USING "+.##^^^^";123
+12E+03
```

_ The underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "_ _"in the format string.

Note: If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

4.57 PRINT# and PRINT# USING - Statements

Format: PRINT#<filenumber>,[USING<string exp>;]<list of exps>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described in PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

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

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

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

would write the following image to disk:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1,CHR#(34);A$:CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC","   93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Appendix B.

See also WRITE#, Section I.4.

4.58 PSET - Statement

Format: PSET (xcoordinate , ycoordinate) [, attribute]

Purpose: PSET sets a point, and defines its attribute.

Remarks: The first argument to PSET is the coordinate of the point to PSET. Coordinates always can come in one of two forms:

PSET (x offset, y offset) or
(absolute x, absolute y)

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced. Examples are:

PSET (10,10) absolute form
PSET (10,0) offset 10 in x and 0 in y
PSET (0,0) origin

Note that when BASIC scans coordinate values it will allow them to be beyond the edge of the screen, however values outside the integer range (-32768 to 32767) will cause an overflow error.

Note that (0,0) is always the upper left hand corner. It may seem strange to start numbering y at the top so the bottom left corner is (0,249) in both Hyperion high-resolution and medium resolution (screen 101 and 102), but this is standard.

PSET allows the attribute argument to be left off and it is defaulted to 3 in medium resolution and 1 in high resolution, since these are the foreground attributes for those modes.

Example:

```
10 FOR i=0 TO 100
20 PSET (i,i)
30 NEXT            'draw a diagonal line
40 FOR i=100 TO 0 STEP -1
50 PSET (i,i),0
60 NEXT            'clear the line
```

4.59 PRESET - Statement

PRESET has an identical syntax to PSET. The only difference is that if no third parameter is given for the backround color zero is selected. When a third argument is given, PRESET is identical to PSET.

Line 50 in the example above could be:

```
50 PRESET (i,i)
```

If an out of range coordinate is given to PSET or PRESET no action is taken nor is an error given. If an attribute greater than 4 is given this will result in illegal function call. Attribute value 2 will be treated like 0 in hi-resolution, and 3 will be treated like 1 for compatibility with medium resolution.

4.60 PUT - File Oriented Statement

Format: PUT [#]<file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Appendix B.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

4.61 PUT - Statement (Graphics)

Format: PUT(x,y),<array>[,<action>]

Purpose: Writes colors onto a specific area of the screen.

Remarks:

(x,y) are the coordinates of the top left corner of the image to be transferred.

<array> is the name of a numeric array containing the information to be transferred. See "GET - Statement (Graphics)" for more information on this array.

<action> is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it onto the screen. However it also provides the option of interacting with the data already on the screen by the use of the action.

PSET as an action simply stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except a negative image is produced. That is, a value of 0 in the array causes the corresponding point to have a color number 3, and vice versa; a value of 1 in the array causes the corresponding point to have a color number 2, and vice versa.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode which may be used for animation. XOR cuases the points on the screen to be inverted where a point exists in the array image. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background twice, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution modes, AND, XOR, and OR have the following effects on color:

AND

		array value			
		0	1	2	3
screen	0	0	0	0	0
	1	0	1	0	1
	2	0	0	2	2
	3	0	1	2	3

OR

		array value			
		0	1	2	3
screen	0	0	1	2	3
	1	1	1	3	3
	2	2	3	2	3
	3	3	3	3	3

XOR

		array value			
		0	1	2	3
screen	0	0	1	2	3
	1	1	0	3	2
	2	2	3	0	1
	3	3	2	1	0

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR).

2. Recalculate the new position of the object.

3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image

4. Go to step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between step 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after" images of the object. This way the extra area will effectively erase the old image. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an "Illegal function call" error occurs.

4.62 RANDOMIZE - Statement

Format: RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3 (user
types 3)
 .2226007   .5941419   .2414202   .2013798   5.361748E-02
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user
types 4) for new sequence>
 .628988   .765605   .5551561   .775797   .7834911
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (user
types 3)
 .2226007   .5941419   .2414202   .2013798   5.361748E-02
Ok
```

4.63 READ - Statement

Format: READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables. (See DATA.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ reads values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (See RESTORE.)

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE          ZIP
DENVER,         COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

4.64 REM - Statement

Format: REM <remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are shown exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of REM.

WARNING: Do not use this in a data statement as it would be considered legal data.

Example:

```
    .
    .
    .
120 FOR I=1 TO 20    'CALCULATE AVERAGE VELOCITY
130 SUM=SUM=V(I)
140 NEXT I
    .
    .
    .
```

4.65 RESTORE - Statement

Format: RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
   .
   .
   .
```

4.66 RESUME - Statement

Format: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0
Execution resumes at the statement which caused the error.

RESUME NEXT
Execution resumes at the statement immediately following the one which caused the error.

RESUME <line number>
Execution resumes at <line number>

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
   .
   .
   .
900 IF (ERR=230) AND (ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
   .
   .
   .
```

4.67 RETURN - Statement

Format: RETURN <line>

Purpose: To bring you back from a subroutine.

Remarks:

line is the line number of the program line you wish to return to.

Although you can use RETURN <line> to return from any subroutine, this enhancement was added to allow non-local returns from the event trapping routines. From one of these routines you will often want to go back to your program at a fixed line number, while still eliminating the GOSUB entry the trap created. Use non-local RETURN with care: any GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

4.68 SCREEN - Statement

Format: SCREEN [mode] [, [burst] [, [apage] [,vpage]]]

Purpose: The Screen statement sets the screen attributes.

Remarks:

mode — a valid numeric expression returning an unsigned Integer value 0, 1 or 2. Valid Modes are:

- 0 - Alpha mode at current width (40 or 80), and IBM attribute interpretation.
- 1 - 320x200 medium resolution Graphics mode.
- 2 - 640x200 high resolution Graphics mode.
- 100 - Alpha mode at current width (40 or 80) and Hyperion attribute interpretation.
- 101 - 320 x 250 medium resolution graphics mode.
- 102 - 640 x 250 high resolution graphics mode.

burst — ignored parameter. On other machines using this BASIC, a value of 0 forces color screens to black and white only. Non-zero values enable color images.

apage — Active page. Valid in alpha only. A numeric expression returning an unsigned Integer in the range 0 to 7 for width 40, or 0 to 3 for Width 80. Selects the page to be written to.

vpage — Visual Page. Valid in alpha only. Same values as apage above, selects which page is to be displayed on the screen. May be different than the active page.

If all parameters are legal, the new screen mode is stored, the screen is erased, Foreground color is set to white, background color is set to Black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is Alpha, and only <apage> and <vpage> are specified, the affect is that of changing display pages for viewing.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" Error. Previous values are retained.

2. Any parameter may be omitted. Omitted parameters assume the old value.

Example:

```
10 SCREEN 0,0,0,0    'Select Alpha mode,
                     'Active and visual page to 0.
20 SCREEN ,,1,2      'Mode unchanged,
                     'use active page 1, but
                      'display page 2.
30 SCREEN 102        'Switch to high res graphic mode.
40 SCREEN 101        'Switch to medium res graphics mode.
50 SCREEN ,0         'Medium res graphics, color off.
```

Note: If the screen 1 or 101 is currently selected (medium resolution graphics), width 80 forces screen 2 or 102, respectively.

If screen 2 or 102 is currently selected (high resolution graphics), width 40 forces screen 1 or 101, respectively.

4.69 SOUND - Statement

Format: SOUND freq,duration

Purpose: The SOUND statement generates sound through the speaker.

Remarks: freq is the desired frequency in Hertz. A valid numeric expression returning an unsigned Integer in the range 37 to 32767.

duration is the desired duration in Clock ticks. A valid numeric expression returning an unsigned Integer in the range 0 to 65535.

Clock ticks occur 18.2 times per second.

Rules:

1. If the duration is zero, any current SOUND statement that is running is turned off. If no SOUND statement is running, SOUND x,0 has no effect.

Example:

```
2500 SOUND RND*1000+37,2 'Creates random sounds.
```

4.70 STOP - Statement

Format: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 1.3)

Example:

```
10 INPUT A,B,A
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

4.71 SWAP - Statement

Format: SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type of variable may be SWAPed (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

4.72 TIME$ - Variable and Statement

Format: As a statement:

TIME$ = <string expression) To set the current time.

As a variable:

<string expression> = TIME$ To get the current time.

Purpose: The TIME$ statement may be used to set or retrieve the current time.

Remarks: The current time is fetched and assigned to the string variable if TIME$ is the expression in a LET or PRINT statement.

The current time is reset if TIME$ is the target of a string assignment.

1. If <string expression> is not a valid string, a "Type mismatch" error will result.

2. For <string expression> = TIME$, TIME$ returns an 8 character string in the form "hh:mm:ss" where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).

3. For TIME$ = <string expression>, <string expression> may be one of the following forms:

   a) "hh" Set the hour. Minutes and seconds default to 00.

   b) "hh:mm" Set the hour and minutes. Seconds default to 00.

   c) "hh:mm:ss" Sets the hour, minutes and seconds.

   If any of the values are out of range, an "Illegal Function Call" error is issued. The previous time is retained.

Example:

```
TIME$ = "08.00"
Ok
PRINT TIME$
08:00:04
Ok
```

The following program displays the current date and time on the 25th line of the screen and will "chime" on the hour in the manner broadcast by WWV.

```
10 KEY OFF:SCREEN 0:WIDTH 40:CLS
20 LOCATE 25,5
30 PRINT DATE$,,TIME$
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 0 THEN 1010
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20

1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4 :GOTO 20
```

Note: Changing TIME$ within BASIC resets the Hyperion's internal clock. This should be avoided. See the TIME command in the Hyperion User Guide for more information.

4.73 WAIT - Statement

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and the AND'ed with I. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to perform a system restart (power off, then back on).

Example:

```
100 WAIT 32,2
```

4.74 WHILE...WEND - Statements

Format:

```
WHILE <expression>
   .
   .
[<loop statements>]
   .
   . WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90  'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115 FLIPS=0
120 FOR I=1 TO J-1
130 IF A$ (I)>A$(I+1)  THEN
    SWAP A$ (I),A$(I+1):FLIPS=1
140 NEXT I
150 WEND
```

4.75 WIDTH - Statement

Format:

```
WIDTH <size>
WIDTH <file no.>,<size>
WIDTH <dev>,<size>
```

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: <size> is the new width. It is a valid numeric expression with a value in the range 1 to 255. The default width is 72 characters.

<file no.> is a valid numeric expression in the range 1 to 4. This is the number of an OPENed device file.

<dev> is a valid string expression returning the device identifier. Valid devices are: SCRN:, and LPT1:.

If <value> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor of the print head, as given by the POS of LPOS function, returns to zero after position 255.

Action:

```
WIDTH <size>
     or
WIDTH "SCRN:",<size>
```

Sets the Screen width. Only 40 or 80 column width is allowed.

NOTE: Changing the screen width causes the screen to be cleared.

If the Screen is in Medium Resolution Graphics Mode (SCREEN 1), WIDTH 80 forces the screen into High Res Graphics Mode (SCREEN 2).

If the Screen is in High Resolution Graphics Mode (SCREEN 2), WIDTH 40 forces the screen into Medium Res Graphics Mode (SCREEN 1).

```
WIDTH "LPT1:",<size>
```

Used as deferred width assignment for the Line Printer. This form of width stores the new width value without actually changing the current width setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS <number> will use this value for width while the file is open.

WIDTH <file no.>,<size>

If the file is open to LPT1:, the Line Printer's width is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1:.

Rules:

1. Valid widths for the Screen are 40 and 80. Valid widths for the Line Printer are 1 to 255.

   Any value entered outside of these ranges will result in an "Illegal Function CAll" error. The previous value is retained.

2. Width has no affect for the Keyboard (KYBD:).

3. The maximum printer width of many printers is 80. However, WIDTH does not complain about values between 80 and 255.

4. Specifying WIDTH 255 for the Line Printer (LPT1:) disables line folding. This has the effect of infinite width.

5. Changing SCREEN mode affects Screen width only when moving between SCREEN 2 and SCREEN 1 or SCREEN 0.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
     .
     .
     .
6020 WIDTH #1,40
```

In the preceeding example, line 10 stores a Line Printer width of 75 characters per line.

Line 20 opens the file #1 to the LIne Printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current Line Printer width to 40 characters per line.

```
SCREEN 1,1     Set Screen to Medium res. Color Graphics.
WIDTH 80       Change Screen to Hi-res. Graphics.
WIDTH 40       Change Screen back to Medium res.

SCREEN 0,1     Changes Screen to 80x25 Alpha Color Mode.
```

4.76 WRITE - Statement

Format: WRITE[<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

4.77 WRITE# - Statement

Format: WRITE#<file number>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

WRITE# outputs data to a sequential file only, while PRINT# may be used with a sequential or random file.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B4
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT31,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

PART I - BASIC INTERPRETER

SECTION 5 - BASIC FUNCTIONS

5.1 ABS - Function

Format: ABS(x)

Purpose: Returns the absolute value of the expression x.

Remarks: X may be any numeric expression

The absolute value of a number is always positive or zero.

Example:

```
OK
PRINT ABS(7*(-5))
 35
OK
```

The absolute value of -35 is positive 35.

5.2 ASC - Function

Format: ASC(x$)

Purpose: Returns the ASCII code for the first character of string x$.

Remarks: x$ may be any string expression

The result of the ASC function is a numerical value that is the ASCII code of the first character of the string x$. (See "Appendix A. ASCII Character Codes" for ASCII codes.) If x$ is null, an "Illegal function call" error is returned.

The CHR$ function is the inverse of the ASC function, and it converts the ASCII code to a character.

Example:

```
OK
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
OK
```

This example shows that the ASCII code for a capital T is 84. Print ASC("TEST") would work just as well.

5.3 ATN - Function

Format: ATN(x)

Purpose: Returns the arctangent of x.

Remarks: x may be a numeric expression of any numeric type, but the evaluation of ATN is always performed in single precision.

The ATN function returns the angle whose tangent is x. The result is a value in radians in the range -PI/2 to PI/2, where PI=3.141593.

If you want to convert radians to degrees, multiply by 180/PI.

Example:

```
OK
PRINT ATN(3)
 1.249046
OK

10 PI=3.141593
20 RADIANS=ATN(1)
30 DEGREES=RADIANS*180/PI
40 PRINT RADIANS,DEGREES
RUN
 .7853983      45
OK
```

The first example shows the use of the ATN function to calculate the arctangent of 3. The second example finds the angle whose tangent is 1. It is .7853983 radians, or 45 degrees.

5.4 CDBL - Function

Format: CDBL(x)

Purpose: Converts x to a double-precision number.

Remarks: x may be any numeric expression.

Rules for converting from one numeric precision to another are followed as explained in Type Conversion, section 1.9. Refer also to the CINT and CSNG functions for converting numbers to integer and single-precision.

Example:

```
Ok
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

The value of CDBL(A) is only accurate to the second decimal place after rounding. The extra digits have no meaning. This is because only two decimal places of accuracy were supplied with A.

5.5 CHR$ - Function

Format: CHR$(n)

Purpose: Converts an ASCII code to its character equivalent.

Remarks: n must be in the range 0 to 255.

The CHR$ function returns the one-character string with ASCII code n. (ASCII codes are listed in "Appendix G. ASCII Character Codes.") CHR$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR$(7) as a preface to an error message (instead of using BEEP). Look under "ASC Function", earlier in this section, to see how to convert a character back to its ASCII code.

Example:

```
OK
PRINT CHR$(66)
B
OK
```

The next example sets function key F1 to the string "AUTO" joined with Enter. This is a good way to set the function keys so the Enter is automatically done for you when you press the function key.

```
OK
KEY 1,"AUTO"+CHR$(13)
OK
```

The following example is a program which shows all the displayable characters, along with their ASCII codes, on the screen in 80-column width.

```
10 CLS
20 FOR I=1 to 255
30 ' ignore nondisplayable characters
40 IF (I>6 AND I<14) OR (I>27 AND I<32) THEN 100
50 COLOR 0,7 ' black on white
60 PRINT USING "##"; I; ' 2-digit ASCII code
70 COLOR 7,0 ' white on black
80 PRINT " "; CHR$(I); " ";
90 IF POS(0)>75 THEN PRINT ' go to next line
100 NEXT I
```

5.6 CINT - Function

Format: CINT(x)

Purpose: Converts x to an integer

Remarks: x may be any numeric expression. If x is not in the range -32768 to 32767, an "Overflow" error occurs.

x is converted to an integer by rounding the fractional portion.

See the FIX and INT functions, both of which also return integers. See also the CDBL and CSNG functions for converting numbers to single- or double-precision.

Example:

```
Ok
PRINT CINT(45.67)
 46
Ok
PRINT CINT(-2.89)
-3
Ok
```

Observe in both examples how rounding occurs.

5.7 COS - Function

Format: COS(x)

Purpose: Returns the trigonometric cosine function.

Remarks: x is the angle whose cosine is to be calculated. The value of x must be in radians. To convert from degrees to radians, multiply the degrees by PI/180, where PI=3.141593.

The calculation of COS(x) is performed in single precision.

Example:

```
OK
10 PI=3.141593
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
RUN
-1
-1
OK
```

This example shows, first, that the cosine of PI radians is equal to -1. Then it calculates the cosine of 180 degrees by first converting the degrees to radians (180 degrees happens to be the same as PI radians).

5.8 CSNG - Function

Format: CSNG(x)

Purpose: Converts x to a single-precision number.

Remarks: x is a numeric expression which will be converted to single-precision.

The rules outlined under "How BASIC Converts Numbers from One Precision to Another" in Chapter 3 are used for the conversion.

See the CINT and CDBL functions for converting numbers to the integer and double-precision data types.

Example:

```
OK
10 A# = 975.3421222#
20 PRINT A#; CSNG(A#)
RUN
 975.3421222  975.3421
OK
```

The value of the double-precision number A# is rounded at the seventh digit and returned as CSNG(A#).

5.9 CVI,CVS,CVD - Functions

Format:

```
CVI(2-byte string)
CVS(4-byte string)
CVD(8-byte string)
```

Purpose: Converts string variable types to numeric variable types, after the string variable has been created using MKI, etc.

Remarks: Numeric values that are read from a random file must be converted from strings into numbers. CVI converts a two-byte string to an integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

The CVI,CVS, and CVD functions do NOT change the bytes of the actual data. They only change the way BASIC interprets those bytes.

See also MKI$, MKS$, MKD$ in this section, and Appendix B.

Example:

```
70 FIELD #1,4 AS N$, 12 AS B$
80 GET #1
90 Y=CVS(N$)
```

This example uses a random file (#1) which has fields defined as in line 70. Line 80 reads a record from the file. Line 90 uses the CVS function to interpret the first four bytes (N$) of the record as a single-precision number. N$ was probably originally a number which was written to the file using the MKS$ function.

5.10 EOF - Function

Format: EOF(filenum)

Purpose: Indicates an end of file condition.

Remarks: Filenum is the number specified on the OPEN statement.

The EOF function is useful for avoiding an "Input past end" error. EOF returns -1 (true) if end of file has been reached on the specified file. A 0 (zero) is returned if end of file has not been reached.

EOF is meaningful only for a file opened for sequential input from diskette or cassette, or for a communications file. A -1 for a communications file means that the buffer is empty.

Example:

```
10 OPEN "DATA" FOR INPUT AS #1
20 C=0
30 IF EOF(1) THEN END
40 INPUT #1,M(C)
50 C=C+1: GOTO 30
```

This example reads information from the sequential file named DATA. Values are read into the array M until end of file is reached.

5.11 EXP - Function

Format: EXP(x)

Purpose: Calculates the exponential function.

Remarks: x may be any numeric expression

This function returns the mathematical number e raised to the x power. e is the base for natural logarithms. An overflow occurs if x is greater than 88.02969.

Example:

```
OK
10 X = 2
20 PRINT EXP(X-1)
RUN
 2.718282
OK
```

This example calculates e raised to the (2-1) power, which is simply e.

5.12 FIX - Function

Format: FIX(x)

Purpose: Truncates x to an integer.

Remarks: x may be any numeric expression.

FIX removes all digits after the decimal point and returns the value of the digits to the left of the decimal point.

The difference between FIX and INT is that FIX does not return the next lower number when x is negative.

Example:

```
OK
PRINT FIX(45.67)
 45
OK
PRINT FIX(-2.89)
-2
OK
```

Note that FIX does not round the decimal part when converting to an integer.

5.13 FRE - Function

Format: FRE(x)

FRE(x$)

Purpose:

Returns the number of bytes in memory that are not being used by BASIC. This number does not include the size of the reserved portion of the interpreter workarea (normally 2.5K to 4K-bytes).

Remarks: x and x$ are dummy arguments.

Since strings in BASIC can have variable lengths (each time you do an assignment to a string its length may change), strings are manipulated dynamically. For this reason, string space may become fragmented.

FRE with any string value causes a housecleaning before returning the number of free bytes. Housecleaning is when BASIC collects all of its useful data and frees up unused areas of memory that were once used for strings. The data is compressed so you can continue until you really run out of space.

BASIC automatically does a housecleaning when it is running out of usable workarea. You might want to use FRE("") periodically to get shorter delays for each housecleaning. Be patient: housecleaning may take a while.

CLEAR,n sets the maximum number of bytes for the BASIC workspace. FRE returns the amount of free storage in the BASIC workspace. If nothing is in the workspace, then the value returned by FRE will be 2.5K to 4K-bytes (the size of the reserved interpreter workarea) smaller than the number of bytes set by CLEAR.

Example:

```
OK
PRINT FRE(0)
 14542
OK
```

The actual value returned by FRE on your computer may differ from this example.

5.14 HEX$ - Function

Format: HEX$(n)

Purpose: Returns a string which represents the hexadecimal value of the decimal argument.

Remarks: n is a numeric expression in the range -32768 to 65535.

If n is negative, the two's complement form is used. HEX$(-n) equals HEX$(65536-n).

OCT$ is the function for octal conversion.

Example: The following example uses the HEX$ function to figure the hexadecimal representation for the two decimal values which are entered.

```
OK
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
OK
RUN
? 1023
 1023 DECIMAL IS 3FF HEXADECIMAL
OK
```

5.15 INP - Function

Format: INP(n)

Purpose: Returns the byte read from port n.

Remarks: n must be in the range 0 to 65535.

INP is the complementary function to the OUT statement (see "OUT Statement" in this chapter).

INP performs the same function as the IN instruction in assembly language. Refer to the IBM Personal Computer Technical Reference manual for a description of valid port numbers (I/O addresses).

Example:

```
100 A=INP(255)
```

This instruction reads a byte from port 255 and assigns it to the variable A.

5.16 INPUT$ - Function

Format: n$=INPUT$(x[,[#]filenum])

Purpose: Returns a string of x characters, read from the keyboard or from file number filenum.

Remarks: n is the number of characters to be read from the file.

filenum is the file number used on the OPEN statement. If filenum is omitted, the keyboard is read.

If the keyboard is used for input, no characters will be displayed on the screen. All characters (including control characters) are passed through, except Ctrl+Brk, which is used to interrupt the execution of the INPUT$ function. When responding to INPUT$ from the keyboard, it is not necessary to press <Rtn>.

The INPUT$ function enables you to read characters from the keyboard which are significant to the BASIC Screen Line Editor, such as RubOut (ASCII decimal value - 08). If you want to read these special characters, you should use INPUT$ or INKEY$.

For communications files, the INPUT$ function is preferred over the INPUT# and LINE INPUT# statements, since all ASCII characters may be significant in communications.

Example: The following program lists the contents of a sequential file in hexadecimal:

```
10 OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

The next example reads a single character from the keyboard in response to a question.

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

5.17 INSTR - Function

Format: INSTR([n,]x$,y$)

Purpose: Searches for the first occurrence of string y$ in x$ and returns the position at which the match is found. The optional offset n sets the position for starting the search in x$.

Remarks: n is a numeric expression in the range 1 to 255.

x$, y$ may be string variables, string expressions or string constants.

If n>LEN(x$), or if x$ is null, or if y$ cannot be found, INSTR returns 0. If y$ is null, INSTR returns n (or 1 if n is not specified).

If n is out of range, an "Illegal function call" error will be returned.

Example:

```
OK
10 A$ = "ABCDEB"
20 B$ = "B"
30 PRINT INSTR(A$,B$);INSTR(4,A$,B$)
RUN
 2  6
OK
```

This example searches for the string "B" within the string "ABCDEB". When the string is searched from the beginning, "B" is found at position 2; when the search starts at position 4, "B" is found at position 6.

5.18 INT - Function

Format: INT(x)

Purpose: Returns the largest integer that is less than or equal to x.

Remarks: x is any numeric expression.

This is called the "floor" function in some other programming languages.

See the FIX and CINT functions, which also return integer values.

Example:

```
OK
PRINT INT(45.67)
 45
OK
PRINT INT(-2.89)
-3
OK
```

This example shows how INT truncates positive integers, but rounds negative numbers upward (in a negative direction).

5.19 LEFT$ - Function

Format: LEFT$(x$,n)

Purpose: Returns the leftmost n characters of x$.

Remarks: x$ is any string expression.

n is a numeric expression which must be in the range 0 to 255. It specifies the number of characters which are to be in the result.

If n is greater than LEN(x$), the entire string (x$) is returned. If n=0, the null string (length zero) is returned.

Also see the MID$ and RIGHT$ functions.

Example:

```
OK
10 A$ = "CUSTOMER SUPPORT"
20 B$ = LEFT$(A$,8)
30 PRINT B$
RUN
SUPPORT
OK
```

In this example, the LEFT$ function is used to extract the first eight characters from the string "CUSTOMER SUPPORT".

5.20 LEN - Function

Format: LEN(x$)

Purpose: Returns the number of characters in x$.

Remarks: x$ is any string expression.

Unprintable characters and blanks are included in the count on the number of characters.

Example:

```
10 X$ = "DYNALOGIC"
20 PRINT LEN(X$)
RUN
 9
OK
```

There are 9 characters in the string "DYNALOGIC".

5.21 LOC - Function

Format: LOC(filenum)

Purpose: Returns the current position in the file.

Remarks: filenum is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record is a 128 byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change this with the /C: option on the BASIC command. If there are more than 255 characters in the buffer, LOC returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for you to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

This first example stops the program if we've gone past the 50th record in the file.

```
300 PUT #1,LOC(1)
```

The second example could be used to re-write the record that was just read.

5.22 LOF - Function

Format: LOF(filenum)

Purpose: Returns the number of bytes allocated to the file (length of the file).

Remarks: filenum is the file number used when the file was opened.

For diskette files created by BASIC, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned. For diskette files created outside BASIC (for example, by using EDLIN), LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, size-LOC(filenum), where size is the size of the communications buffer, which defaults to 256 but may be changed with the /C: option on the BASIC command. Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose.

Example: These statements will get the last record of the file named BIG, assuming BIG was created with a record length of 128 bytes:

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

5.23 LOG - Function

Format: LOG(x)

Purpose: Returns the natural logarithm of x.

Remarks: x must be a numeric expression which is greater than zero.

The natural logarithm is the logarithm to the base e.

Example: The first example calculates the logarithm of the expression 45/7:

```
OK
PRINT LOG(45/7)
 1.860752
OK
```

The second example calculates the logarithm of e and of e2:

```
OK
E= 2.718282
OK
? LOG(E)
 1
OK
? LOG(E*E)
 2
OK
```

5.24 LPOS - Function

Format: LPOS(n)

Purpose: Returns the current position of the print head within the printer buffer for LPT1:.

Remarks: n indicates which printer is being tested, as follows:

```
0 or 1 LPT1:
2      LPT2:
3      LPT3:
```

The LPOS function does not necessarily give the physical position of the print head on the printer.

Example: In this example, if the line length is more than 60 characters long we send a carriage return character to the printer so it will skip to the next line.

```
100 IF LPOS(0)>60 THEN LPRINT CHR$(13)
```

5.25 MID$ - Function

Format: MID$(<string exp1>,n[,m])=<string exp2>

where 1<n<255 and 0<m<255 and <string expl> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

5.26 MKI$, MKS$, MKD$ - Functions

Format:
```
MKI$ (integer expression)
MKS$ (single-precision expression)
MKD$ (double-precision expression)
```

Purpose: To convert numeric values to string values.

Remarks: Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

These functions differ from STR$ in that they do not actually change the bytes of the date, just the way BASIC interprets those bytes.

Refer also to the CVI, CVS, CVD Functions.

Example:
```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
    .
    .
    .
```

5.27 OCT$ - Function

Format: OCT$(X)

Purpose: To return a string which represents the octal value of the decimal argument.

Remarks: X is a numeric expression in the range of -32768 to 65535.

If X is negative, the two's complement form is used. That is, OCT$(-X) is the same as OCT$(65536-X).

Refer to the HEX$ function for hexadecimal conversion.

Example:

```
PRINT OCT$(24)
 30
Ok
```

5.28 PEEK - Function

Format: PEEK(I)

Purpose: To return the byte read from the indicated memory position.

Remarks: I is an integer in the range 0 to 65535. I is the offset from the current segment as defined by the DEF SEG statement, and indicates the address of the memory location to be read. (Refer to DEF SEG Statement)

The returned value will be an integer in the range 0 to 255.

PEEK is the complementary function to the POKE statement.

Example: A=PEEK(&H5A00)

5.29 POINT - Function

Format: POINT(x,y)

Purpose: To return the color of the specified point on the screen.

Remarks: (x,y) are the coordiantes of the point. Coordinates must be in absolute form.

Example:

```
10 SCREEN 2
20 IF POINT (1,1) = 0 THEN PRESET (1,1)
```

5.30 POS - Function

Format: POS(I)

Purpose: To return the current cursor column position.

Remarks: The current horizontal (column) position of the cursor is returned. The returned value will be in the range of 1 to 40 or 1 to 80, depending on the current WIDTH setting.

Example: IF POS(X)>60 THEN PRINT CHR$(13)

5.31 RIGHT$ - Function

Format: n$ = RIGHT$(x$,I)

Purpose: To return the rightmost n characters of string x$.

Remarks: x$ is any string expression.

If I=LEN(x$), x$ will be returned. If I=0, the null string (length zero) is returned.

Example:

```
10 A$="OTTAWA, ONTARIO"
20 PRINT RIGHT$(A$,7)
RUN
ONTARIO
Ok
```

Also see the MID$ and LEFT$ functions.

5.32 RND - Function

Format: RND[(X)]

Purpose: To return a rundom number between 0 and 1.

Remarks: The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded. (see RANDOMIZE). However, X,0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
       12  65  86  72  79
Ok
```

5.33 SCREEN - Function

Format: x = SCREEN(row,col [,z])

Purpose: The SCREEN Function returns the ordinal of the character from the screen at the specified row (line) and column.

Remarks: x is a numeric variable receiving the ordinal returned.

row is a valid numeric expression returning an unsigned Integer in the range 1 to 24.

col is a valid numeric expression returning an unsigned Integer in the range 1 to 40 or 1 to 80 depending upon the width.

z is a vlid numeric expression returning a boolean result.

The ordinal of the character at the specified coordinates is stored in the numeric variable. If the optional parameter <z> is given and non-zero, the color attribute for the character is returned instead.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" error.

Example:

```
100 X = SCREEN (10,10)    'If the character at 10,10 is
                          'A then return 65.

110 X = SCREEN (1,1,1)    'Return the color attribute of
                          'the character in the upper
                          'left hand corner of the screen.
```

5.34 SGN - Function

Format: SGN(X)

Purpose: To return the sign of X

Remarks: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100. If X is negative, 200 if X is 0 and 300 if X is positive.

5.35 SIN - Function

Format: SIN(X)

Purpose: Returns the trigonometric sine of X in radians.

Remarks: SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14158/2).

Example:

```
PRINT SIN(1.5)
Ok
```

5.36 SPACE$ - Function

Format: SPACE$(X)

Purpose: To return a string of spaces of the length X.

Remarks: The expression X is rounded to an integer and must be in the range 0 to 255.

Refer also to the SPC function.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

5.37 SPC - Function

Format: SPC(I)

Purpose: To print I blanks on the terminal.

Remarks: SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A ';' is assumed to follow the SPC(I) command.

If I>width, I is changed to (I mod width)

Example:

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

5.38 SQR - Function

Format: SQR(X)

Purpose: To return the square root of X.

Remarks: X must be >=0.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10          3.162278
 15          3.872984
 20          4.472136
 25          5
Ok
```

5.39 STR$ - Function

Format: STR$(X)

Purpose: To return a string representation of the value of X.

Remarks: X is any numeric expression.

Refer also to the VAL function.

Example:

```
5 PRINT LEN(STR$(34))
10 PRINT LEN("34")
RUN
 3
 4
```

5.40 STRING$ - Function

Format: STRING$(I,J)
STRING$(I,X$)

Purpose: To return a string of length I whose characters all have ASCII code J or the first character of X$.

Remarks: I,J are in the range 0 to 255.

X$ is any string expression.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

5.41 TAB - Function

Format: TAB(I)

Purpose: Spaces to position I on the terminal.

Remarks: If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","25.00"
RUN
NAME                    AMOUNT

G. T. JONES             $25.00
Ok
```

5.42 TAN - Function

Format: TAN(X)

Purpose: Returns the tangent of X in radians.

Remarks: TAN(X) is calculated in single precision.

Example:
```
10 Y = Q*TAN(X)/2
```

5.43 USR - Function

Format: USR[<digit>](X)

Purpose: Calls the assembly language subroutine with the argument X.

Remarks: <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
    .
    .
    .
```

5.44 VAL - Function

Format: VAL(X$)

Purpose: To return the numerical value of string X$.

Remarks: The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example

VAL(" -3)

returns -3.

Refer to the STR$ function for numeric to string conversion.

Example:

```
10 READ TITLES$,CITY$,STATES$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96669 THEN PRINT
TITLES$   TAB(25)   "OUT   OF   STATE"   30   IF
VAL<ZIP$)>=90801 AND  VAL(ZIP$)<=90815  THEN  PRINT
TITLES$ TAB(25) "LONG BEACH"
    .
    .
    .
```

5.45 VARPTR$ - Function

Format: VARPTR$(variable)

Purpose: Returns a character form of the address of a variable in memory. It is primarily for use with PLAY and DRAW in programs that will later be compiled.

Remarks:

VARPTR$ is a new function in BASIC release 1.10.

variable is the name of a variable existing in the program.

VARPTR$ returns a three-byte string in the form:

Byte 0	Byte 1	Byte 2
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2 integer

3 string

4 single-precision

8 double-precision

The returned value is the same as:

```
CHR$(type)+MKI$(VARPTR(variable))
```

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Release 1.00	1.10 Equivalent
PLAY"XA$;"	PLAY"X"+VARPTR$(A$)
PLAY"O=I;"	PLAY"O="+VARPTR$(I)

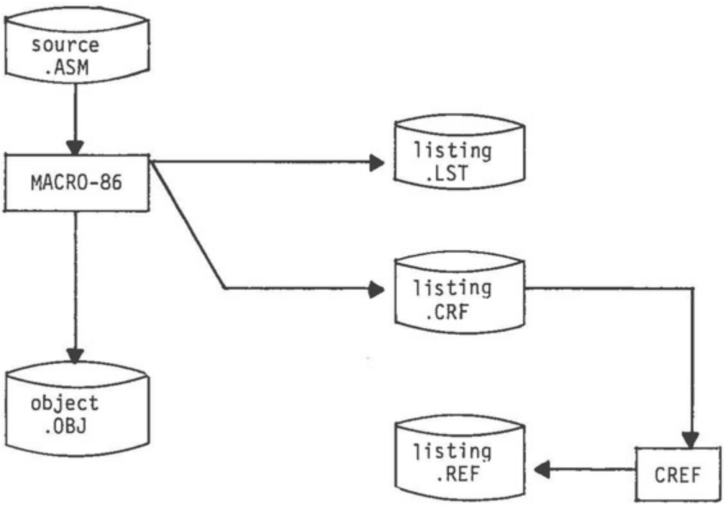
PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 1 - MACRO ASSEMBLER

MACRO ASSEMBLER

Macro Assembler will create, on command, a listing file and a cross-reference file. The listing file contains the beginning relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. And, the listing contains a symbol table which shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file receives the default filename extension .LST.

The cross reference file contains a compact representation of variables, labels, and symbols. The cross reference file receives the default filename extension .CRF. When this cross reference file is processed by CREF, the file is converted into an expanded symbol table that lists all the variables, labels, and symbols in slphabetical order, followed by the line number of in the source program where each is defined, followed by the line numbers where each is used in the program. The final cross reference listing receives the filename extension .REF. (Refer to the CREF chapter for further explanation and instructions.)


source
.ASM
MACRO-86
listing
.LST
listing
.CRF
object
.OBJ
listing
.REF
CREF

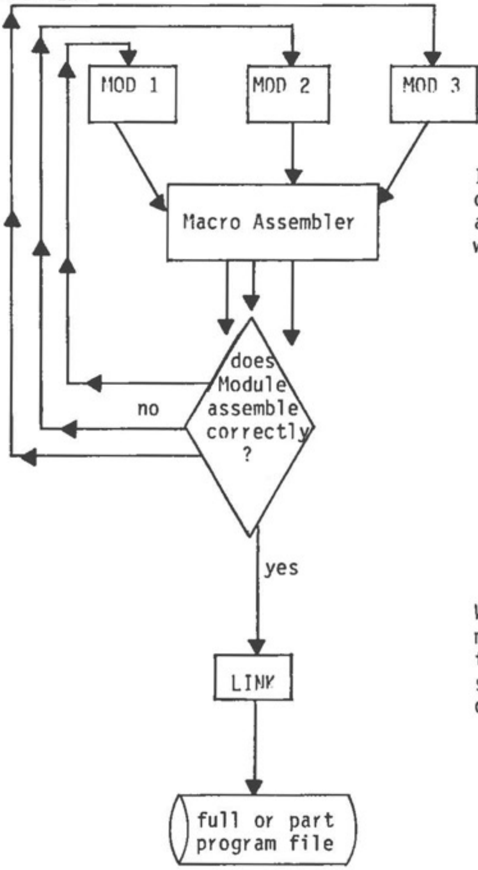
INTRODUCTION

Features and Benefits of Macro Assembler

Dynalogic's Macro Assembler is a very powerful assembler for the Hyperion. Macro Assembler incorporates many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from your Hyperion. Even though Macro Assembler is more complex than any other microcomputer assembler, it is easy to use.

Macro Assembler produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked, using LINK, to produce relocatable, executable object code. Relocatable code can be loaded anywhere in memory. Thus, the program can execute where it is most efficient, not only in a fixed range of memory addresses.

In addition, relocatable code means that programs can be created in modules, each of which can be assembled, tested, and perfected individually. This saves recoding time because testing and assembly is performed on smaller pieces of program code. Also, all modules can be error free before being linked together into larger modules or into the whole program. The program is not a huge monolith of code.



Individual modules can be edited and assembled until they work correctly

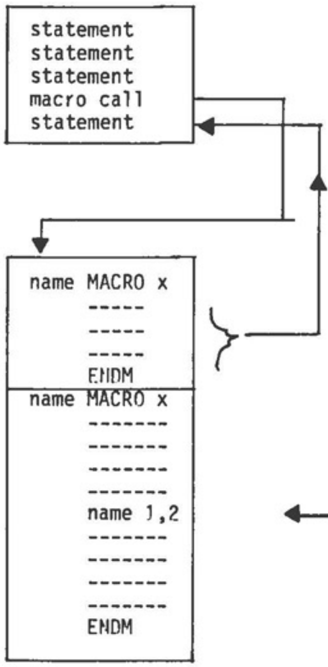
When the individual modules are ready, they can be linked singly or into one or more larger modules

Macro Assembler supports Microsoft's complete 8080 macro facility, which is Intel 8080 standard. The macro facility permits the writing of blocks of code for a set of instructions used frequently. The need for recoding these instructions each time they are needed is eliminated.

This block of code is given a name, called a macro. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, a simple "call" to the macro is placed in the source file. Macro Assembler expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are given only once, then other occurrences are one line calls.

Macros can be "nested", that is, one macro may be called from inside another macro. Nesting of macros is limited only by memory.

The macro facility includes repeat, indefinite repeat, and indefinite repeat character directives for programming repeat block operations. The MACRO directive can also be used to alter the action of any instruction or directive by using its name as the macro name. When any instruction or directive statement is placed in the program, Macro Assembler checks first the symbol table it created to see if the instruction or directive is a macro name. If it is, Macro Assembler "expands" the macro call statement by replacing it with the body or instructions in the macro's definition. If the name is not defined as a macro, Macro Assembler tries to match the name with an instruction or directive. The MACRO directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.



When the assembler encounters a macro call, it finds the MACRO block and replaces the call with the block of statements that define the macro

Nested MACRO call: name defined elsewhere as a macro, is "expanded" during assembly, as shown above.

Macro Assembler supports an expanded set of conditional directives. Directives for evaluating a variety of assembly conditions can test assembly results and branch where required. Unneeded or unwanted portions of code will be left unassembled. Macro Assembler can test for blank or nonblank arguments, for defined or not-defined symbols, for equivalence, for first assembly pass or second, and Macro Assembler can compare strings for identity or difference. The conditional directives simplify the evaluation of assembly results, and make programming the tested code for conditions easier as well as more powerful.

Macro Assembler's conditional assembly facility also supports conditionals inside conditionals ("nesting"). Conditional assembly blocks can be nested up to 255 levels.

If the condition in the expression (shown by <exp true>) is true, the IF block is assembled up to ELSE, then skips to ENDIF. If no ELSE, then simply assemble the whole conditional block

```
statement
statement
statement
IF <exp true>
   ---------
   ---------
   ---------
ELSE
   ---------
   ---------
   ---------
ENDIF
statement
statement
    .
    .
    .
```

If the condition in the expression is false, Macro Assembler skips to ELSE, then resumes assembly at the next statement. If ELSE is not used, skips to ENDIF and resumes assembly with next statement.

```
IF ...
   ------
   ------
   IF ...
      ------
      ------
      IF ...
         ------
         ------
      ENDIF
   ELSE
      ------
      ------
   ENDIF
   ------
   ------
ENDIF
```

Nesting of conditionals is allowed; up to 255 levels

Macro Assembler supports all the major 8080 directives found in Microsoft's MACRO-80 Macro Assembler. This means that any conditional, macro, or repeat blocks programmed under MACRO-80 can be used under Macro Assembler. Processor instructions and some directives (eg., .PHASE, CSEG, DSEG) within the blocks, if any, will need to be converted to the 8086 instruction set. All the major MACRO-80 directives (pseudo-ops) that are supported under Macro Assembler will assemble as is, as long as the expressions to the directives are correct for the processor and the program. The syntax of directives is unchanged. Macro Assembler is upward compatible, with MACRO-80 and with Intel's ASM86, except Intel codemacros and macros.

Macro Assembler provides some relaxed typing. Some 8086 instructions take only one operand type. If a typeless operand is entered for an instruction that accepts only one type of operand (e.g., in the instruction PUSH [BX],[BX] has no size, but PUSH only takes a word), it seems wasteful to return an error for a lapse of memory or a typographical error. When the wrong type choice is given, Macro Assembler returns an error message but generates the "correct" code. That is, it always puts out instructions, not just NOP's. For example, if you enter:

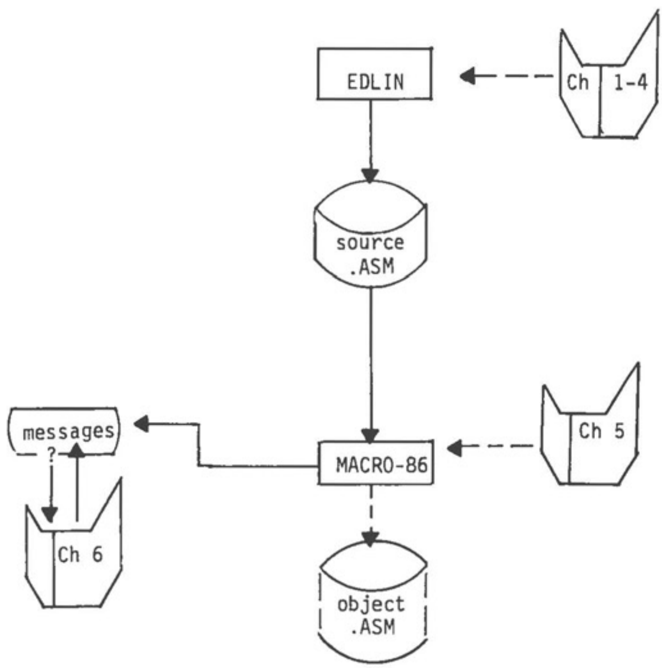
```
                          MOV       AL,WORDLBL
you may have                          (2)
meant one of                          MOV AL,BYTE PRR WORDLBL
three instructions:                 (3)
                                    MOV AL,<other>
                        (1)
                          MOV AX,WORDLBL
```

Macro Assembler generates instruction (2) because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size." Macro Assembler accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some mundane debugging chores. An error message is still returned, however, because you may have misstated the operand the Macro Assembler assumes is "correct."

Overview of Macro Assembler Operation

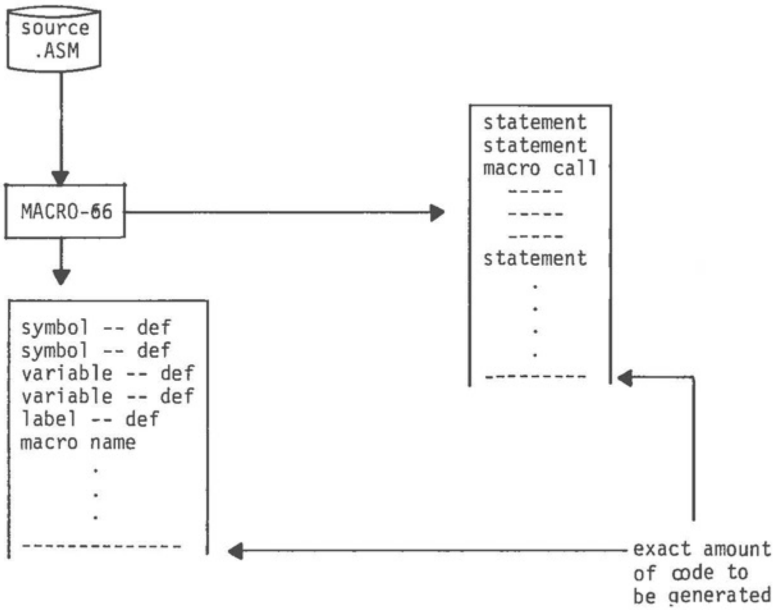
The first task is to create a source file. Use EDLIN (the resident editor in DOS), to create the Macro Assembler source file. Macro Assembler assumes a default filename extension of .ASM for the source file. Creating the source file involves creating instruction and directive statements that follow the rules and constraints described in this manual.

When the source file is ready, run Macro Assembler as described in Section II-5. Refer to Section II.6 for explanations of any messages displayed during or immediately after assembly.



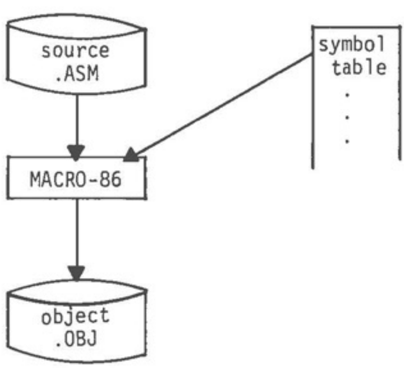
Macro Assembler is a two-pass assembler. This means that the source file is assembled twice. But slightly different actions occur during each pass. During the first pass, the assembler evaluates the statements and expands macro call statements, calculates the amount of code it will generate, and builds a symbol table where all symbols, variables, labels, and macros are assigned values. During the second pass, tha assembler fills in the symbol, variable, labels, and expression values from the symbol table, expands macro call statements, and emits the relocatable object code into a file with the default filename extension .OBJ. The .OBJ file is suitable for processing with LINK. (The .OBJ file can be stored as part of the user's library of object programs, which later can be linked with one or more explanation and instructions).

The source file can also be assembled without creating an .OBJ file. All other assembly steps are performed, but the object code is not sent to disk. Only erroneous source statements are displayed on the terminal screen. This practice is useful for checking the source code for errors. It is faster than creating an .OBJ file because no file creating or writing is performed. Modules can be test assembled quickly and errors corrected before the object code is put on disk. Modules that assemble with errors do not clutter the diskette.


source
.ASM
MACRO-86
statement
statement
macro call

-----
statement
symbol -- def
symbol -- def
variable -- def
variable -- def
label -- def
macro name
exact amount
of code to
be generated

PASS 2


source
.ASM
symbol
table
MACRO-86
object
.OBJ

1.1 CREATING A MACRO ASSEMBLER SOURCE FILE

To create a source file for Macro Assembler, you need to use an editor program, such as EDLIN, in Hyperion's DOS. You simply create a program file as you would for any other assembly or high-level programming language. Use the general facts and specific descriptions in this section and the three following sections when creating the file.

In this section, you will find discussions of the statement format and introductory descriptions of its components. In Section 3, you will find full descriptions of names: variables, labels, and symbols. In Section 4, you will find full descriptions of expressions and their components, operands and operators. In Section 5, you will find full descriptions of the assembler directives.

1.1.1 General Facts About Source Files

Naming Your Source File

When you create a source file, you will need to name it. A filename name may be any name that is legal for your operating system. Macro Assembler expects a specific three character filename extension, .ASM. Whenever you run Macro Assembler to assemble your source file, Macro Assembler assumes that your source filename has the filename extension .ASM. This is not required. You may name your source file with any extension you like. However, when you run Macro Assembler, you must remember to specify the extension. If you use .ASM, you will not need to specify the extension. (Because of this default action by Macro Assembler, it is impossible to omit the filename extension. When you assemble a source file without a filename extension, Macro Assembler will assume that the source has a .ASM extension because you would not be specifying an extension. When Macro Assembler searches the diskette for the file, it will not find the correct file and will either assemble the wrong file or will return an error message stating that the file cannot be found.)

Note, also, that Macro Assembler gives the object file it outputs the default extension .OBJ. To avoid confusion or the destruction of your source file, you will want to avoid giving a source file an extension of .OBJ. For similar reasons, you will also want to avoid the extensions .EXE, .LST, .CRF, and .REF.

Legal Characters

The legal characters for your symbol names are:

A-Z 0-9 ? @ _ $

Only the numerals (0-9) cannot appear as the first character of a name (a numeral must appear as the first character of a numeric value). Additional special characters act as operators or delimiters:

- : (colon) segment override operator
- . (period) operator for field name of Record or Structure; may be used in a filename only if it is the first character.
- [ ] (square brackets) around register names to indicate value in address in register not value (data) in register
- ( ) (parentheses) operator in DUP expressions and operator to change precedence of operator evaluation
- < > (angle brackets) operators used around initialization values for Records or Structure, around parameters in IRP macro blocks, and to indicate literals.

  The square brackets and angle brackets are also used for syntax notation in the discussions of the assembler directives (section 1.4.2). When these characters are operators and not syntax notation, you are told explicitly; for example, "angle brackets must be coded as shown."

Numeric Notation

The default input radix for all numeric values is decimal. The output radix for all listings is hexadecimal for code and data items and decimal for line numbers. The output radix can only be changed to octal radix by giving the /O switch when Macro Assembler is run (see Section 1.5.3, Command Switches). The input radix may be changed two ways:

1. The .RADIX directive (see Section 1.4.2.1, Memory Directives)

2. Special notation append to a numeric value:

Radix	Range	Notation	Example
Binary	0-1	B	01110100B
Octal	0-7	Q or O (letter)	735Q 621O
Decimal	0-9	(none) or D	9384 (default) 8149D (when .RADIX directive changes default radix to not decimal.)
Hexadecimal	0-9 A-F	H	0FFH 80H (first character must be numeral in range 0-9)

What's in a Source File?

A source file for Macro Assembler consists of instruction statements and directive statements. Instruction statements are made of 8086 instruction mnemonics and their operands, which command specific processes directly to the 8086 processor. Directive statements are commands to Macro Assembler to prepare data for use in and by instructions.

Statement format is described in Section 1.4.2.1. The parts of a statement are described in Sections 1.3-1.6 and in Sections 3 - 5. Statments are usually placed in block of code assigned to a specific segment (code, data, stack, extra). The segments may appear in any order in the source file. Within the segments, generally speaking, statements may appear in any order that creates a valid program. Some exceptions to random ordering do exist, which will be discussed under the affected assembler directives.

Every segment must end with an end segment statement (ENDS), every procedure must end with an end procedure statement (ENDP), and every structure must end with an end structure statement (ENDS). Likewise, the source file must end with an END statement that tells Macro Assembler where program execution should begin.

Section 1.3.1, Memory Organization, describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another and to your programming as a whole. This information is important and helpful for developing your programs. The information is presented in Section 4 as a prelude to the discussion of operands and operators.

1.1.2 Statement Line Format

Statements in source files follow a strict format, which allows some variations.

Macro Assembler directive statements consist of four "fields": Name, Action, Expression, Comment. For example:

```
FOO     DB          0D5EH        ;create variable FOO
                                 ;containing the value 0D5EH

Name    Action      Expression   ;Comment
```

Macro Assembler Instruction statements ususally consist of three "fields": Action, Expression, Comment. For Example:

```
MOV         CX,FOO          ;here's the count number

Action      Expression      ;Comment
```

An instruction statement may have a Name field under certain circumstances; see the discussion of Names below.

Names

The name field, when present, is the first entry on the statement line. The name may begin in any column, although mormally names are started in column one.

Names may be any length you choose. However, Macro Assembler considers only the first 31 characters significant when your source file is assembled.

One other significant use for names is with the MACRO directive. Although all the rules covering names, described in Section 3 apply the same to MACRO names, the discussion of macro names is better left to the sections on the macro facility.

Macro Assembler supports the use of names in a statement line for three purposes: to represent code, to represent data, and to represent constants.

To make a name represent code, use:

NAME: followed by an directive, instruction, or nothing at all
NAME LABEL NEAR (for use inside its own segment only)
NAME LABEL FAR (for use outside its own segment)
EXTRN NAME:NEAR (for use outside its own module but inside its own segment only)
EXTRN NAME:FAR (for use outside its own module and segment)

To make a name represent data, use:

NAME LABEL <size> (BYTE, WORD, etc.)
NAME Dx <exp>
EXTRN NAME:<size> (BYTE, WORD, etc.)

To make a name represent a constant, use:

NAME EQU <constant>
NAME = <constant>
NAME SEGMENT <attributes>
NAME GROUP <segment-names>

Comments

Comments are never required for the successful operation of an assembly language program, but they are strongly recommended.

If you use comments in your program, every comment on every line must be preceded by a semicolon. If you want to place a very long comment in your program, you can use the COMMENT directive. The COMMENT directive releases you from the required semicolon on every line (refer to COMMENT in Section 1.4.2.1).

Comments are used to document the processing that is supposed to happen at a particular point in a program. When comments are used in this manner, they can be useful for debugging, for altering code, or for updating code. Consider putting comments at the beginning of each segment, procedure, structure, module, and after each line in the code that begins a step in the processing.

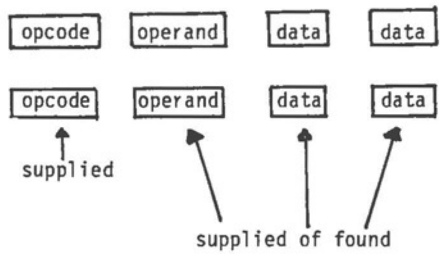
Comments are ignored by Macro Assembler. Comments do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code. Comments are not required for anything but human understanding.

Action

The action field contains either an 8086 instruction mnemonic or a Macro Assembler directive. Refer to Section 1.4.1 for some general discussion. The Macro Assembler directives are described in detail in Section 1.4.2.

If the name field is blank, the action field will be the first entry in the statement format. In this case, the action may appear starting in any column, 1 through maximum line length (less columns for action and expression).

The entry in the action field either directs the processor to perform a specific function or directs the assembler to perform one of its functions. Instructions command processor actions. An instruction my have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements.

(opcode is the action part of an instruction)

Directives give the assembler directions for I/O, memory organization, conditional assembly, listing and cross reference control, and definitions.

Expressions

The expression field contains entries which are operands and/or combinations of operands and operators.

Some instructions take no operands, some take one, and some take two. For two operand instructions, the expression field consists of a destination operand and a source operand, in that order, separated by a comma. For example:

| opcode | | dest-operand | , | source-operand |

For one operand instructions, the operand is a source or a destination operand, depending on the instruction. If one or both of the operands is omitted, the instruction carries that information in its internal coding.

Source operands are immediate operands, register operands, memory operands, or Attribute operands. Destination operands are register operands and memory operands.

For directives, the expression field usually consists of a single operand. For example:

| directive | | operand |

A directive operand is a data operand, a code (addressing) operand, or a constant, depending on the nature of the directive.

For many instructions and directives, operands may be connected with operators to form a longer operand that looks like a mathematical expression. These operands are called complex. Use of a complex operand permits you to specify addresses or data derived from several places. For example:

```
MOV     FOO[BX],AL
```

The destination operand is the result of adding the address represent by the variable FOO and the address found in register BX. The processor is instructed to move the value in register AL to the destination calculated from these two operand elements. Another example:

```
MOV     AX,FOO+5[BX]
```

In this case, the source operand is the result of adding the value represented by the symbol FOO plus 5 plus the value found in the BX register.

Macro Assembler supports the following operands and operators in the expression field (shown in order of precedence):

Operands

Immediate
FIELD
(incl. symbols)
Register
Memory
label
variables
THIS,
    simple
    indexed
    structures
Attribute
override
    PTR
    :(seg)
    SHORT
    HIGH
    LOW
value returning
    OFFSET
    SEG
    THIS
    TYPE
    .TYPE
    LENGTH
    SIZE
record specifying
    FIELD
    MASK
    WIDTH

Operators

LENGTH, SIZE, WIDTH, MASK,

[]. (). <>

segment override (:)

PTR, OFFSET, SEG, TYPE,

HIGH, LOW

*, /, MOD, SHL, SHR

+, -(unary), -(binary)

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

SHORT, .TYPE

NOTE

Some operators can be used as operands or as part of an operand expression.

1.2 NAMES: LABELS, VARIABLES AND SYMBOLS

Names are used in several capacities throughout Macro Assembler, wherever any naming is allowed or required.

Names are symbolic representations of values. The values may be addresses, data, or constants.

Names may be any length you choose. Hovever, Macro Assembler will truncate names longer than 31 characters when your source file is assembled.

Names may be defined and used in a number of ways. This section introduces you to the basic ways to define and use names. You will discover additional uses as you study the sections on Expression and Action, and as you use Macro Assembler.

Macro Assembler supports three types of names in statement lines: labels, variables, and symbols. This section covers how to define and use these three types of names.

1.2.1 Labels

Labels are names used as targets for JMP, CALL, and LOOP instructions. Macro Assembler assigns an address to each label as it is defined. When you use a label as an operand for JUMP, CALL, or LOO, Macro Assembler can substitute the attributes of the label for the label name, sending processing to the appropriate place.

Labels are defined one of four ways:

1. <name>:

   Use a name followed immediately by a colon. This defines the name as a NEAR label. <name>: may be prefixed to any instruction and to all directives that allow a Name field. <name>: may also be placed on a line by itself.

   Examples:

   ```
   CLEAR_SCREEN:   MOV   AL,20H
   FOO:    DB   0FH
   SUBROUTINE3:
   ```

2. <name> LABEL NEAR
   <name> LABEL FAR

Use the LABEL directive. Refer to the discussion of the LABEL directive in Section 1.4.2.1, Memory Directives.

NEAR and FAR are discussed under Type Attribute below.

Examples:

```
FOO   LABEL   NEAR
GOO   LABEL   FAR
```

3.
```
<name>   PROC   NEAR
<name>   PROC   FAR
```

   Use the PROC directive. Refer to the discussion of the PROC directive in Section 1.4.2.1, Memory Directives.

   NEAR is optional because it is the default if you enter only <name> PROC. NEAR and FAR are discussed under the Type Attribute below.

   Example:

```
REPEAT    PROC   NEAR
CHECKING   PROC   :same as CHECKING PROC NEAR
FIND_CHR   PROC   FAR
```

4.
```
EXTRN <name>:NEAR
EXTRN <name>:FAR
```

   Use the EXTRN directive.

   NEAR and FAR are discussed under the Type Attribute below.

   Refer to the discussion of the EXTRN directive in Section 1.4.2.1, Memory Directives.

```
EXTRN FOO:NEAR
EXTRN ZOO:FAR
```

A label has four attributes: segment, offset, type, and the CS ASSUME in effect when the label is defined. Segment is the segment where the label is defined. Offset is the distance from the beginning of the segment to the label's location. Type is either NEAR or FAR.

Segment

Labels are defined inside segments. The segment must be assigned to the CS segment register to be addressable. (The segment may be assigned to a group, in which case the group must be addressable through the CS register.) Therefore, the segment (or group) attribute of a symbol is the base address of the segment (or group) where it is defined.

Offset

The offset attribute is the number of bytes from the beginning of the label's segment to where the label is defined. The offset is a 16-bit unsigned number.

Type

Labels are one to two types: NEAR or FAR. NEAR labels are used for references from within the segment where the label is defined. NEAR labels may be referenced from more than one module, as long as the references are from a segment with the same name and attributes and has the same CS ASSUME.

FAR labels are used for references from segments with a different CS ASSUME or if there is more than 64K bytes between the label reference and the label definition.

NEAR and FAR cause Macro Assembler to generate slightly different code. NEAR labels supply their offset attribute only (a 2 byte pointer). FAR labels supply both their segment and offset attributes (a 4 byte pointer).

1.2.2 Variables

Variables are names used in expression (as operands to instructions and directives).

A variable represents an address where a specified value may be found.

Variable look much like labels and are defined in some ways alike. The differences are important.

Variables are defined three ways:

```
1. <name> <define-dir>                    ;no colon!
   <name> <struc-name> <expression>
   <name> <rec-name> <expression>
```

<define-dir> is any of the five Define directives: DB,DW,DD,DQ,DT

Example:

```
     START_MOVE   DW   ?
```

<struc-name> is a structure name defined by the STRUC directive.

<rec-name> is a record name defined by the RECORD directive.

Examples:

```
CORRAL STRUC
      .
      .
      .
       ENDS
HORSE  CORRAL     <'SADDLE'>
```

Note that HORSE will have the same size as the structure CORRAL.

```
GARAGE RECORD     CAR:8='P'

SMALL  GARAGE     10 DUP(<'Z'>)
```

Note that SMALL will have the same size as the record GARAGE.

See the Define, STRUC, and RECORD directives in Section

1.4.2.1, Memory Directives.

2. <name> LABEL <size>

Use the LABEL directive with one of the size specifiers.

<size> is one of the following size specifiers:

```
BYTE  - specifies 1 byte
WORD  - specifies 2 bytes
DWORD - specifies 4 bytes
QWORD - specifies 8 bytes
TBYTE - specifies 10 bytes
```

Example:

```
CURSOR  LABEL  WORD
```

See LABEL directive in Section 1.4.2.1, Memory Directives.

3. EXTRN <name>:<size>

Use the EXTRN directive with one of the size specifiers described above. See EXTRN directive in Section 1.4.2.1, Memory Directives.

Example:

```
EXTRN FOO:DWORD
```

As do labels, variables also have the three attributes segment, offset, and type.

Segment and offset are the same for variables as for labels. The type attribute is different.

Type

The type attribute is the size of the variable's location, as specified when the variable is define. The size depends on which Define directive was used or which size specifier was used to define the variable.

Directive	Type	Size
DB	BYTE	1 byte
DW	WORD	2 bytes
DD	DWORD	4 bytes
DQ	QWORD	8 bytes
DT	TBYTE	10 bytes

1.2.3 Symbols

Symbols are names defined without reference to a Define directive or to code. Like variables, symbols are also used in expression as operands to instructions and directives.

Symbols are defined in three ways:

1. <name> EQU <expression>

   Use the EQU directive. See EQU directive in Section 1.4.2.1, Memory Directives.

   <expression> may be another symbol, and instruction mnemonic, a valid expression, or any other entry (such as text or indexed references).

   Examples:

   ```
   FOO   EQU   7H
   ZOO   EQU   FOO
   ```

2. <name> = <expression>

   Use the equal sign directive. See Equal Sign directive in Section 1.4.2.1, Memory Directives.

   <expression> may be any valid expression.

   Examples:

   ```
   GOO   =   0FH
   GOO   =   $+2
   GOO   =   GOO+FOO
   ```

3. EXTRN <name>:ABS

   Use the EXTRN directive with type ABS. See EXTRN directive in Section 1.4.2.1, Memory Directives.

   Example:

   ```
   EXTRN BAZ:ABS
   ```

   BAZ must be defined by an EQU or = directive to a valid expression.

SECTION 3

EXPRESSIONS: OPERANDS AND OPERATORS

Section 1 provided a brief introduction to expressions. Basically, expression is the term used to indicate values on which an instruction or directive peforms its functions.

Every expression consists of at least one operand (a value). An expression may consist of two or more operands. Multiple operands are joined by operators. The result is a series of elements that look like a mathematical expression.

This chapter describes the types of operands and operators that Macro Assembler supports. The discussion of memory organization in a Macro Assembler program acts as a preface to the descriptions of operands and operators, and as a link to topics discussed in Section 2.

1.3.1 MEMORY ORGANIZATION

Most of your assembly language program is written in segments. In the source file, a segment is a block of code that begins with a SEGMENT directive statement and ends with an ENDS directive. In an assembled and linked file, a segment is any block of code that is addressed through the same segment register and is not more than 64K bytes long.

You should note that Macro Assembler leaves everything to do with segments to LINK. LINK resolves all references. For that reason, Macro Assembler does not check (because it cvannot) if your references are entered with the correct distance type. Values such as OFFSET are also left to the linker to resolve.

Although a segment may not be more than 64K bytes long, you may, as long as you observe the 64K limit, divide a segment among two or more modules. (The SEGMENT statement in each module must be the same in every aspect.

When the modules are linked together, the several segments become one. References to labels, variables, and symbols within each module acquire the offset from the beginning of the whole segment, not just from the beginning of their portion of the whole segment. (All divisions are removed.)

You have the option of grouping several segments into a group, using the GROUP directive. When you group segments, you tell Macro Assembler that you want to be able to refer to all of these segments as a single entity. (This does not eliminate segment identity, nor does it make values within a particular segment less imemdiately accessible. It does make value relative to a group base.) The value of grouping is that you can refer to data items without worrying about segment overrides and about changing segment registers often.

With this in mind, you should note that references within segments or groups are relative to a segment register. Thus, until linking is complete, the final offset of a reference is relocatable. For this reason, the OFFSET operator does not return a constant. The major purpose of OFFSET is to cause Macro Assembler to generate an immediate instruction; that is, to use the address of the value instead of the value itself.

There are two kinds of referneces in a program:

1. Code references - JMP, CALL, LOOPxx - These references are relative to the address in the CS register. (You cannot override this assignment.)

2. Data references - all other references - These references are usually relative to the DS register, but this assignment may be overridden.

   When you give a forward reference in a program statement, for example:

   ```
   MOV AX,<ref>
   ```

Macro Assembler first looks for the segment of the reference. Macro Assembler scans the segment registers for the SEGMENT of the reference then the GROUP, if any, of the reference.

However, the use of the OFFSET operator always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must override this restriction by using the GROUP name and the colon operator, for example:

```
MOV AX, OFFSET <group-name>:<ref>
```

If you set a segment register to a group with the ASSUME directive, then you may also override the restriction on OFFSET by using the register name, for example:

```
MOV AX,OFFSET DS:<ref>
```

The result of both of these statements is the same.

Code labels have four attributes:

1. segment - what segment the label belongs to

2. offset - the number of bytes from the beginning of the segment

3. type - NEAR or FAR

4. CS ASSUME - the CS ASSUME the label was coded under

When you enter a NEAR JMP or NEAR CALL, you are changing the offset (IP) in CS. Macro Assembler compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they are different, Macro Assembler returns an error (you must use a FAR JMP or CALL).

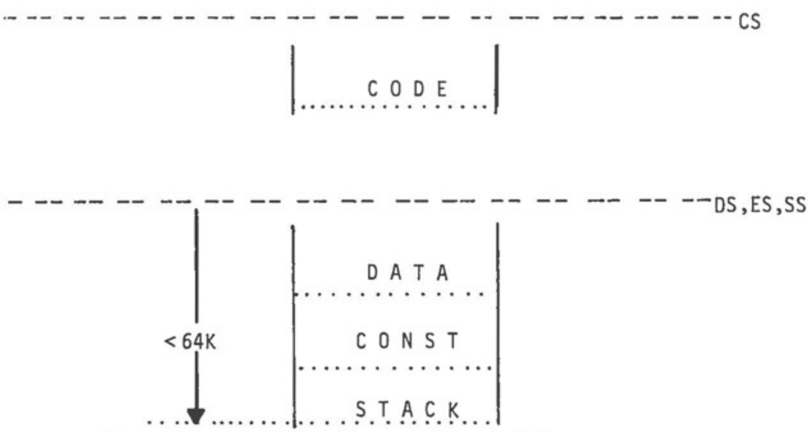
When you enter a FAR JMP or FAR CALL, you are changing both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

Let's take a common case. A segment called CODE; and a group (called DGROUP) that contains three segments (called DATA, CONST, and STACK).

The program statements would be:

```
DGROUP   GROUP    DATA,CONST,STACK
           ASSUME   CS:CODE,DS:DGROUP,SS:DGROUP,ES:DGROUP
           MOV      AX,DGROUP  ;CS initialized by entry;
                      ;as soon as possible, especially
                      ;before an DS relative references
              .
              .
              .
```

As a diagram, this arrangement could be represented as follows:



Given this arrangement, a statement like:

```
MOV AX,<variable>
```

causes Macro Assembler to find the best segment register to reach this variable. (The "best" register is the one that requires no segment overrides.)

A statement like:

```
MOV AX,OFFSET <variable>
```

tells Macro Assembler to return the offset of the variable relative to the beginning of the variable's segment.

If this <variable> is in the CONST segment and you want to reference its offset from the beginning of DG5ROUP, you need a statement like:

```
MOV AX,OFFSET DGROUP:<variable>
```

Macro Assembler is a two-pass assembler. During pass 1, it builds a symbol table and calculates how much code is generated but does not produce object code. If undefined items are found (including forward references), asseumptions are made about the reference so that the correct number of bytes are generated on pass 1. Only certain types of errors are displayed, errors involving items that must be defined on pass 1. No listing is produced unless you give a /D switch when you run the assembler. The /D switch produces a listing for both passes.

On pass 2, the assembler uses the values defined in pass 1 to generate the object code. Definitions of references during pass 2 are checked against the pass 1 value, which is in the symbol table. Also, the amount of code generated during pass 1 must match the amount generated during pass 2. If either is different, Macro Assembler returns a phase error.

Because pass 1 must keep correct track of the relative offset, some references must be known on pass 1. If they are not known, the relative offset will not be correct.

The following references must be known on pass 1:

1. IF/IFE <expression>
   If <expression> is not known on pass 1, Macro Assembler does not know to assemble the conditional block (or which part to assemble if ELSE is used). On pass 2, the assembler would know and would assemble, resulting in a phase error.

2. <expression> DUP(...)
   This operand explicitly changes the relative offset, so <expression> must be known on pass 1. The value in parentheses need not be known because it does not affect the number of bytes generated.

3. .RADIX <expression>
   Because this directive changes the input radix, constants could have a different value, which could cause Macro Assembler to evaluate IF or DUP statements incorrectly.

The biggest problem for the assembler is handling forward references. How can it know the kind of a reference which it still has not seen the definition? This is one of the main reasons for two passes. And, unless Macro Assembler can tell from the statement containing the forward reference what the size, the distance, or any other of its attributes are, the assembler can only take the safe route, (generate the largest possible instruction in some cases except for segment override or FAR). This results in extra code that does nothing. (Macro Assembler figures this out by pass 2, but it cannot reduce the size of the instructions without causing an error, so it puts out NOP instructions (90H).)

For this reason, Macro Assembler includes a number of operators to help the assembler. These operators tell Macro Assembler what size instruction to generate when it is faced with an ambiguous choice. As a benefit, you can also reduce the size of your program by using these operators to change the nature of the arguments to the instructions.

Some Examples

```
MOV AX,FOO ;FOO = forward constant
```

This statement causes Macro Assembler to generate a move from memory instruction on pass 1. By using the OFFSET operator, we can cause Macro Assembler to generate an immediate operand instruction.

```
MOV AX,OFFSET FOO ' OFFSET says use the address of FOO
```

Because OFFSET tells Macro Assembler to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code.

Similarly, if you have a CALL statement that calls to a label that may be in a different CS ASSUME, you can prevent problems by attaching the PTR operator to the label:

```
CALL FAR PTR <forward-label>
```

At the opposite extreme, you may have a JMP forward that is less than 127 bytes. You can save yourself a byte if you use the SHORT operator.

```
JMP SHORT <forward-label>
```

However, you must be sure that the target is indeed within 127 bytes or Macro Assembler will not find it.

The PTR operator can be used another way to save yourself a byte when using forward references. If you defined FOO as a forward constant, you might enter the statement:

```
MOV [BX],FOO
```

You may want to refer to FOO as a byte immediate. In this case, you could enter either of the statements (they are equivalent):

```
MOV BYTE PTR [BX],FOO

MOV [BX],BYTE PTR FOO
```

These statements tell Macro Assembler that FOO is a byte immediate. A smaller instruction is generated.

1.3.2 OPERANDS

An operand may be any one of three types: Immediate, Registers, or Memory operands. There is no restriction on comining the various types of operands.

The following list shows all the types and the items that comprise them:

- Immediate
  - Data items
    - Symbols
- Registers
- Memory operands
  - Direct
    - Labels
    - Variables
    - Offset (fieldname)
  - Indexed
    - Base register
    - Index register
    - [constant]
    - +displacement
  - Structure

1.3.2.1 Immediate Operands

Immediate operands are constant values that you supply when you enter a statement line. The value may be entered either as a data item or as a symbol.

Instructions that take two operands permit an immediate operand as the source operand only (the second operand in an instruction statement). For example:

```
MOV AX,9
```

Data Items

The default input radix is decimal. Any numeric values entered with numeric notation appended will be treated as a decimal value. Macro Assembler recognizes values in forms other than decimal when special notation is appended. These other values include ASCII characters as well as numeric values.

Data Form	Format Example
Binary	xxxxxxxxB 01110001B
Octal xxxO	xxxO 735O (letter O) 412O
Decimal xxxxxD	xxxxx 65535 (default) 1000D (when .RADIX changes input to nondecimal)
Hexadecimal	xxxxH 0FFFFH (first digit must be 0-9)
ASCII "xx"	'xx' 'OM' (more than two with DB only; both forms are synonomous)
10 real	xx.xxE+xx 25.23E-7 (floating point format)
16 real	x...xR 8F76DEA9R (first digit must be 0-9; The total number of digits must be 8, 16, or 20; or 9, 17, 21 if first digit is 0)

Symbols

Symbols names equated with some form of constant information may be used as immediate operands. Using a symbol constant in a statement is the same as using a numeric constant. Therefore, using the sample statement above, you could enter:

```
MOV AX,FOO
```

assume FOO was defined as a constant symbol. For example:

```
FOO EQU 9
```

1.3.2.2 Register Operands

The 8086 processor contains a number of registers. These registers are identified by two-letter symbols that the processor recognizes (the symbols are reserved).

The registers are appropriated to different tasks: general registers, pointer registers, counter registers, index registers, segment registers, and a flag register.

The general registers are two sizes: 8 bit and 16 bit. All other registers are 16 bit.

The general registers are both 8 bit and 16 bit registers. Actually, the 16 bit general registers are composed of a pair of 8 bit registers, one for the low byte (bits 0-7) and one for the high byte (bits 8-15). Note, however, that each 8 bit general register can be used independently from its mate. In this case, each 8 bit register contains bitso-7.

Segment registers are initialized by the user and contain segment base values. The segment register names (CS, DS, SS, ES) can be used with the colon segment override operator to inform Macro Assembler than an operand is in a different segment than specified in an ASSUME statement. (See the segment override operation in 3.3.1, Attribute Operators.)

The flag register is one 16-bit register containing nine 1 bit flags (six arithmetic flags and three control flags).

Each of the registers (except segment registers and flags) can be an operand in arithmetic and logical operations.

Register/Memory Field Encoding:

MOD=11		
R/M	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Mode

EFFECTIVE ADDRESS CALCULATION			
R/M	MOD=00	MOD=01	MOD=10
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16
011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16
100	[SI]	[SI]+D8	SI +D16
101	[DI]	[DI]+D8	DI +D16
110	DIRECT ADDRESS	[BP]+D8	BP +D16
111	[BX]	[BX]+D8	BX +D16

Note: D8 = a byte value; D16 = a word value

Other Registers:

Segment:	CS	code segment
	DS	data segment
	SS	stack segment
	ES	extra segment

Flags:	6 1-bit arithmetic flags		3 1-bit control flags
	CF	carry flag	DF direction flag
flag			
	AF	auxiliary flag	TF trap flag
	ZF	zero flag	
	SF	sign flag	

NOTE

The BX, BP, SI, and DI registers are also used as memory operands. The distinction is: when these registers are enclosed in square brackets [ ], they are memory operands, when they are not enclosed in square brackets, they are register operands. (See section 3.2.3, Memory Operands).

1.3.2.3 Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct Macro Assembler to an address to find some data or instruction.

A memory operand always consists of an offset from a base address.

Memory operands fit into three categories: those that use a base or index register (indexed memory operands), those that do not use a register (direct memory operands), and structure operands.

Direct Memory Operands

Direct memory operands do not use registers and consist of a single offset value. Direct memory operands are labels, simple variables, and offsets.

Memory operands can be used as destination operands as well as source operands for instructions that take two operands. For example:

```
MOV AX,FOO
MOV FOO,CX
```

Indexed Memory Operands

Indexed memory operands use base and index registers, constants, displacement values, and variables, often in combination. When you combine indexed operands, you create an address expression.

Indexed memory operands use square brackets to indicate indexing (by a register or by registers) or subscripting (for example FOO[5]). The square brackets are treated like plus signs (+). Therefore,

```
FOO[5] is equivalent to FOO+5
5[FOO] is equivalent to 5+FOO
```

The only difference between square brackets and plus signs occurs when a register name appears inside the square brackets. Then, the operand is seen as indexing.

The types of indexed memory operands are:

Base registers:	[BX] [BP] BP has SS as its default segment register; all others have DS as default.
Index registers:	[DI] [SI]
[constant]	immediate in square brackets [8], [FOO]
+Displacement	8-bit or 16-bit value. Used only with another indexed operand.

These elements may be combined in any order. The only restriction is that neither two base registers nor two indexed registers can be combined:

```
[BX+BP] ; illegal
[SI+DI] ; illegal
```

Some examples of indexed memory operand combination:

```
[BP+8]
[SI+BX][4]
16[DI+BP+3]
8[FOO]-8
```

More examples of equivalent forms:

```
5[BX][SI]
BX+5[SI]
```

```
[BX+SI+5]
[BX]5[SI]
```

Structure Operands

Structure operands take the form <variable>,<field>

<variable> is any name you give when coding a statement line that initializes a Structure field. The <variable> may be an anonymous variable, such as an indexed memory operand.

<field> is a name defined by a DEFINE directive within a STRUC block. <field> is a typed constant.

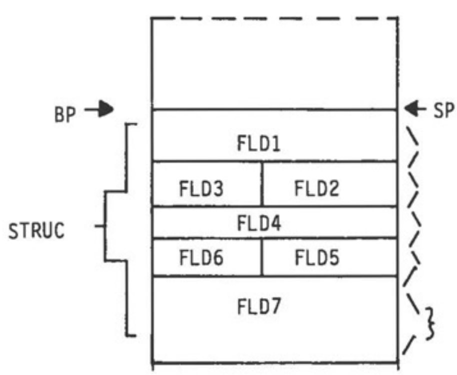
The period (.) must be enclosed.

Example:

```
ZOO      STRUC
GIRAFFE  DB  ?
ZOO      ENDS

LONG_NECK   ZOO <16>

MOV AL,LONG_NECK.GIRAFFE

MOV AL,[BX].GIRAFFE   ;anonymous variable
```

The use of structure operands can be helpful in stack operations. If you set up the stack segment as a structure, setting BP to the top of the stack (BP equal to SP), then you can access any value in the stack structure by fieldname indexed through BP; for example:

BP .FLD6



This method makes all values on the stack available all the time, not just the value at the top. Therefore, this method makes the stack a handy place to pass parameters to subroutines.

1.3.3 OPERATORS

An operator may be one of four types: attribute, artihmetic, relational, or logical.

Attribute operators are used with operands to override their attributes, return the value of the attributes, or to isolate fielsd of Records.

Arithmetic, relational, and logical operators are used to combine or compare operands.

1.3.3.1 Attribute Operators

Attribute operators used as operands perform one of three functions:

Override an operand's attributes,

Return the values of operand attributes,

Isolate record fields (record specific operators).

The following list shows all the attribute operators by type:

Override operators
- PTR
- colon (:) (segment override)
- SHORT
- THIS
- HIGH
- LOW

Value returing operators
- SEG
- OFFSET
- TYPE
- .TYPE
- LENGTH
- SIZE

RECORD specific operators
- Shift count (Field name)
- WIDTH
- MASK

Override operators

These operators are used to override the segment, offset, type, or distance of variables and labels.

Pointer (PTR)

<attribute> PTR <expression>

The PTR operator overrides the type (BYTE, WORD, DWORD) or the distance (NEAR, FAR) of an operand.

<attribute> is the new attribute; the new type or new distance.

<expression> is the operand whose attribute is to be overridden.

The most important and frequent use for PTR is to assure that Macro Assembler understands what attribute the expression is supplosed to have. This is expecially true for the type attribute. Whenever you place forward references in your program, PTR will make clear the distance or type of the expression. This way you can avoid phase errors.

The second use of PTr is to access data by type other than the type in the variable definition. Most often this occurs in the structures. If the structure is defined as WORD buy you want to access an item as a byte, PTR is the operator for this. However, a much easier method is to enter a second statement that defines the structure in bytes, too. This eliminates the need to use PTR for every reference to the structure. Refer to the LABEL directive in Section ???4.2.1, Memory Directives.

Examples:

```
CALL WOR PTR [BX] [SI]
MOV BYTE PTR ARRAY

ADD BYTE PTR FOO,9
```

Segment Override (:) (colon)

```
<segment-register>:<address-expression>
<segment-name>:<address-expression>
<group-name>:<address-expression>
```

The segment override operator overrides the assumed segment of an address expression (which may be a lable, a variable, or other memory operand).

The colon operator helps with forward references by telling the asembler to what a reference is relative (segment, group, or segment register).

Macro Assembler assumes that labels are addressable through the current CS register. Macro Assembler assumes that variable are addressable through the current DS register, or possibly the ES register, by default. If the operand is in another segment and you have not alerted Macro Assembler through the ASSUME directive, you will need to use a segment override operator. Also, if you want to use a nondefault relative base (that is, not the default segment register), you will need to use the segment override operator for forward references. Note that if Macro Assembler can reach an operand through a nondefault segment register, it will use it, but hte reference cannot be forward in this case.

<segment-register> is one of the four segment register names: CS, DS, SS, ES.

<segment-name> is a name defined by the SEGMENT directive.

<group-name> is a name defined by the GROUP directive.

Examples:

```
MOV AX,ES:[BX+SI]

MOV CSEG:FAR_LABEL,AX

MOV AX,OFFSET DGROUP:VARIABLE
```

SHORT

SHORT <label>

SHORT overrides NEAR distance attribute of lables used as targets for the JMP instruction. SHORT tells Macro Assembler that the distance between the JMP statement and the <label> specified as its operand is not more than 127 bytes either direction.

The major advantage of using the SHORT operator is to save a byte. Normally, the <label> carries a 2-byte pointer to its offset in its segment. Because a range of 256 bytes can be handled in a single byte, the SHORT operator eliminates the need for the extra byte (which would carry 00 or FF anyway). However, you must be sure that the target is within 127 bytes of the JMp instruction before using SHORT.

Example:

```
          JMP SHORT REPEAT
              .
              .
              .
 REPEAT:
```

THIS

THIS <distance> THIS <type>

The THIS operator creates an operand. The value of the operand depends on which argument you give THIS.

The argument to THIS may be:

1. A distance (NEAR or FAR)

2. A type (BYTE, WORD, or DWORD)

THIS <distance> creates an operand with the distance attribute you specify, an offset equal to the current loaction counter, and the segment attribute (segment base address) of the enclosing segment.

THIS <type> creates an operand with the type attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

Examples:

```
TAG EQU THIS BYTE same as TAG LABLE BYTE

SPOT_CHECK = THIS NEAR same as SPOT_CHECK LABEL NEAR
```

HIGH,LOW

HIGH <expression> LOW <expression>

HIGH and LOW are provided for 8080 assembly language compatibility. HIGH and LOW are byte isolation operators.

HIGH isolates the high 8 bits of an absolut 16-bit value or address expression.

LOW isolates the low 8 bits of an absolut 16-bit value or address expression.

Examples:

```
MOV AH,HIGH WORD_VALUE :get byte with sign bit

MOVE AL,LOW 0FFFFH
```

Value Returning Operators

These operators return the attribute values of the operands that follow them but do not override the attributes.

The value returning operators take labels and variables as their arguments.

Because variables in Macro Assembler have three attributes, you need to use the value returning operators to isolate single attributes, as follows:

SEG	isolates the segment base address
OFFSET	isolates the offset value
TYPE	isolates either type or distance
LENGTH and SIZE	isolate the memory allocation

SEG

SEG <label>
SEG <variable>

SEG returns the segment value (segment base address) of the segment enclosing the label or variable.

Examples:

```
MOV AX,SEG VARIABLE_NAME
MOV AX,<segment-variable>:<variable>
```

OFFSET

OFFSET <label> OFFSET **<variable>**

OFFSET returns the offset value of the variable or label within its segment (the number of bytes between the sement base address and the address where the label or variable is defined).

OFFSET is chiefly used to tell the assembler that the operand is an immediate.

NOTE

OFFSET does not make the value a constant. Only LINK can resolve the final value.

NOTE

OFFSET is not required with uses of the DW or DD directives. The assembler applies an implicit OFFSET to variables in address expressions following DW and DD.

Example:

```
MOVE BX,OFFSET FOO
```

You must be sure that the GROUP directive precedes any reference to a group name, including its use with OFFSET.

TYPE

TYPE <label>
TYPE <variable>

If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the bariable type, as follows:

```
BYTE  = 1
WORD  = 2
DWORD = 4
QWORD = 8
TBYTE = 10
STRUC = the number of bytes declared by STRUC
```

If the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Examples:

```
MOV AX,(TYPE FOO_BAR) PTR [BX+SI]
```

LENGTH

LENGTH <variable>

LENGTH accepts only one variable as its argument.

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for that variable.

If the variable is defined by a DUP expression, LENGTH returns the number of type units duplicated; that is, the number that precedes the first DUP in the expression.

If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples:

```
FOO DW 100 DUP(1)

MOVE CX,LENGTH FOO ;get number of elements
                   ;in array
                   ;LENGTH returns 100
```

```
BAZ DW 100 DUP(1,10 DUP(?))
```

LENGTH BAZ is still 100,
regardless of the expression following DUP.

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE

SIZE <variable>

SIZE returns the total number of bytes allocated for a variable.

SIZE is the product of the value of LENGTH times the value of TYPE.

Example:

```
FOO DW 100 DUP(1)

MOV BX,SIZE FOO  ;get total bytes in array
```

SIZE = LENGTH X TYPE
SIZE = 100 X WORD
SIZE = 100 X 2
SIZE = 200

Record Specific operators

Record specific operators used to isolate fields in a record.

Records are defined by the RECORD directive (see Section 1.4.2.1, Memory Directives). A record may be up to 16 bits long. The record is defined by fields, which may be from ont to 16 bits long. To isolate one of the three characteristics of a record fields, you use one of the record specific operators, as follows:

Shift count number of bits from low end of record to low end of field (number of bits to right shift the record to lowest bits of record).

WIDTH the number of bits wide the field or record is (number of bits the field or record contains)

MASK value of record if field contains its maximum value and all other fields are zero (all bits in field contain 1; all other bits contain 0)

In the following discussions of the record specific operators, the following symbols are used:

FOO a record defined by the RECORD directive FOO RECORD FIELD1:3,FIELD2:6,FIELD3:7

BAZ a variable used to allocate FOO BAZ FOO < >.

FIELD1, FIELD2, and FIELD3 are the fields of the record FOO.

Shift-count - (Record fieldname)

<record-fieldname>

The shift count is derived from the record fieldname to be isolated.

The shift count is the number of bits the field must be right shifted to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (FOO) contains three fields (FIELD1, FIELD2, and FIELD3), the record can be diagrammed as follows:

FIELD1 has a shift count of 13.
FIELD2 has a shift count of 7.
FIELD3 has a shift count of 0.

When you want to isolate the value in one of these fields, you enter its name as an operand.

Example:

```
MOV DX,BAZ
MOV CL,FIELD2
SHR DX,CL
```

FIELD2 is now right shifted, ready for access.

MASK

MASK <record-fieldname>

MASK accepts a field name as its only argument.

MASK returns a bit-mask defined by 1 for bits positions included by the field and 0 for bit positions not included. The value return represents the maximum value for the record when the field is masked.

Using the diagram used for shift count, MASK can be diagrammed as:

```
|0|0|0|1|1|1|1|1|1|0|0|0|0|0|0|0|  <-- MASK
     1   |   F   |   8   |   0
```

The MASK of FIELD2 equals 1F80H.

Example:

```
MOV DX,BAZ
AND DX,MASK FIELD2
```

FIELD2 is now isolated.

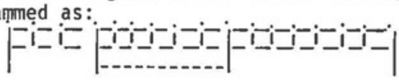
WIDTH

WIDTH <record-fieldname>
WIDTH <record>

When a <record-fieldname> is given as the argument, WIDTH returns the width of a record field as the number of bits in the record field.

When a <record> is given as the argument, WIDTH returns the width of a record as the number of bits in the record.

Using the diagram under Shift count, WIDTH can be diagrammed as:



WIDTH = 6

The WIDTH of FIELD1 equals 3. The WIDTH of FIELD2 equals 6. The WIDTH of FIELD3 equals 7.

Example:

```
MOVE CL,WIDTH FIELD2
```

The number of bits in FIELD2 is now in the count register.

1.3.3.2 Arithmetic Operators

Eight arithmetic operators proved the common mathematical functions (add, subtract, divide, multiply, modulo, negation), plus two shift operators.

The arithmetic operators are used to combine operands to from an expression that results in a data item or an address.

Except for + and - (binary), operands must be constants.

For plus (+), one operand must be a constant.

For minus (-), the first (left) operand may be a nonconstant, or both operands may be nonconstants. But, the right may not be a nonconstant if the left is constant.

*	Multiply
/	Divide
MOD	Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolut.

Example:

```
MOV AX,100 MOD 17
```

The value moved into AX will be 0FH (decimal 15).

SHL — Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be left shifted.

Example:

```
MOV AX,0110B SHL 5
```

The value moved into AX will be 011000000B (0C0H)

- (Unary Minus) Indicates that following value is negative, as in a negative integer.

+ Add. One operand must be a constant; one may be a nonconstant.

- Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be nonconstants. But, the right may be a nonconstant only it the left is also a nonconstant and in the same segment.

1.3.3.3 Relational Operators

Relational operators compare two constant operands.

If the relationship between the two operands matches the operator, FFFFH is returned.

If the relationship between the two operands does not match the operator, a zero is returned.

Relational operators are most often used with conditional directives and conditional instructions to direct program control.

EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less than or Equal.Returns true if the left operand is less than or equal to the right operand
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.

1.3.3.4 Logical Operators

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate for the logical relationship defined by the logical operator.

Logical operators can be used two ways:

1. To combine operands in a logical relationship. In this case, all bits in the operands will have the same value (either 0000 or FFFFH). In fact, it is best to use these values for true (FFFFH) and false (0000) for the symbols you will use as operands because in conditionals aything nonzero is true.

2. In bitwise operations. In this case, the bits are different, and the logical operators act the same as the instructions of the same name.

NOT — Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.

AND — Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.

OR — Logical OR. Returns true if either operator is true of it both are true. Returns false is both operators are false. Both operands must be absolute values.

XOR — Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operatos are true or if both operators are false. Both operands must be absolute values.

1.3.3.5 Expression Evaluation: Precedence of Operators

Expressions are evaluated higher precedence operators first, then left to right for equal precedence operators.

Parentheses can be used to alter precedence.

For example:

```
MOV AX,101B SHL 2*2     =   MOV AX,00101000B

MOV AX,101B SHL (2*2)   =   MOV AX,01010000B
```

SHL and * are equal precedence. Therefore, their functions are performed in the order the operators are encountered (left to right).

Precedence of Operators

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
Entries inside: parenthesis ( )
angle brackets < >
square brackets [ ]
structure variable operand: <variable>.<field>
2. segment override operator: colon (:)
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MODD, SHL, SHR
6. +, - (both unary and binary)
7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT, .TYPE

SECTION 4

ACTION: INSTRUCTIONS AND DIRECTIVES

The action field contains either an 8086 instruction mnemonic or a Macro Assembler directive.

Following a name field entry (if any), action field entries may begin in any column. Specific spacing is not required. The only benefit of consistent spacing is imporoved readability. If a statement does not have a name field entry, the action field is the first entry.

The entry in the action field either directs the processor to perform a specific function or directs the assembler to perform one of its functions.

1.4.1 INSTRUCTIONS

Instructions command processor actions. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instrucion. For example:

$\boxed{\text{opcode}}$ $\boxed{\text{operand}}$ $\boxed{\text{data}}$ $\boxed{\text{data}}$

$\boxed{\text{opcode}}$ $\boxed{\text{operand}}$ $\boxed{\text{addr}}$ $\boxed{\text{addr}}$

supplied supplied or found

supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements.

(opcode equates to the binary code for the action of an instruction)

This manual does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this, you will need to consult other text. For now, the following text exist:

1. Morse, Stephen P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.

2. Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA: Osbourne/McGraw-Hill, 1980

3. The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1980

1.4.2 DIRECTIVES

Directives give the assembler directions for input and output, memory organization, conditional assembly, listing and cross reference control, and definitions.

The directives have been divided into groups by the function they perform. Within each group, the directives are described alphabetically.

The groups are:

Memory Directives
: Directives in this group are used to organize memory. Because there is no "miscellaneous" group, the memory directives group contains some directives that do not, strictly speaking, organize memory, such as COMMENT.

Conditional Directives
: Directives in this group are used to test conditions of assembly before proceding with assembly of a block of statements. This group contains all of the IF (and related) directives.

Macro Directives
: Directives in this group are used to create blocks of code called macros. This group also includes some special operators and directives that are used only inside macro blocks. The repeat directives are considered macro directives for descriptive purposes.

Listing Directives
: Directives in this group are used to control the format and, to some extent, the content of listings that the assembler produces.

Here below is an alphabetical list of all the directives that Macro Assembler supports:

ASSUME	EVEN	IRPC	.RADIX
	EXITM		RECORD
COMMENT	EXTERN	LABEL	REPT
	GROUP	.LFCOND	.SALL
DB		.LIST	SEGMENT
DD	IF		.SFCOND
DQ	IFB	MACRO	STRUC
DT	IFDEF		SUBTTL
DW	IFDIF	NAME	
	IFE		.TFCOND
ELSE	IFIDN	ORG	TITLE
END	IFNB	%OUT	
ENDIF	IFNDEF		.XALL
ENDM		PAGE	.XCREF
ENDP	IF1	PROC	.XLIST
ENDS	IF2	PUBLIC	
EQU	IRP	PURGE	

1.4.2.1 Memory Directives

ASSUME

ASSUME <seg-reg>:<seg-name>[,...]

or

ASSUME NOTHING

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register. When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from 1 to 4 arguments to ASSUME.

The valid <seg-reg> entries are:

CS, DS, ES, and SS.

The possible entries for <seg-name> are:

1. the name of a segment declared with the SEGMENT directive

2. the name of a group declared with the GROUP directive

3. an expression: either SEG <variable-name> or SEG <label-name> (see SEG operator, Section 1.3.2)

4. the key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement.

If ASSUME is not used or if NOTHING is entered for <seg-name>, each reference to variables, symbols, labels, and so forth in a particular segment must b prefixed by a segment register. For example, DS:FOO instead of simply FOO.

Example:

```
ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHIN
```

COMMENT

COMMENT<delim><text><delim>

The first non-blank character encountered after COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter>.

COMMENT permits you to enter comments about your program without entering a semicolon (;) before each line.

If you use COMMENT inside a macro block, the comment block will not appear on your listing unless you also place the .LALL directive in you source file.

Example:

Using an asterisk as the delimiter, the format of the comment block would be:

```
COMMENT  *
any amount of text entered
here as the comment block
     .
     .
     .      *   ;return to normal mode
```

DEFINE BYTEDEFINE WORD
DEFINE DOUBLEWORD
DEFINE QUADWORD
DEFINE TENBYTES

```
<varname>     DB     <exp>[,<exp>,...]
<varname>     DW     <exp>[,<exp>,...]
<varname>     DD     <exp>[,<exp>,...]
<varname>     DQ     <exp>[,<exp>,...]
<varname>     DT     <exp>[,<exp>,...]
```

The DEFINE directives are used to define variables or to initialize portions of memory.

If the optional <varname> is entered, the DEFINE directives define the name as a variable. If <varname> has a dolon, it becomes a NEAR label instead of a variable. (See also, Section 1.2.1, Labels, and Section 1.2.2, Variable.)

The DEFINE directives allocate memory in units specified by the second letter of the directive (each define directive may allocate one or more of its units at a time):

DB allocates one byte (8 bits)
DW allocates one word (2 bytes)
DD allocates two words (4 bytes)
DQ allocates four words (8 bytes)
DT allocates ten bytes

<exp> may be one or more of the following:

1. a constant expression

2. the character ? for indeterminate initialization. Usually the is used to reserve space without placing any particular value into it. (It is the equivalent of the DS pseudo-op in MACRO-80).

3. an address expression (for DW and DD only)

4. an ASCII string (longer than 2 characters for DB only)

5. <exp>DUP(?)
When this type of expression is the only argument to a define directive, the define directive produces an uninitialized datablock. This expression with the ? instead of a value results in a smaller object file because only the segment offset is changed to reserve space.

6. <exp>DUP(<exp>[,...])
This expression, like item 5, produces a data block, but initialized with the value of the second <exp>. The first <exp> must be a constant greater than zero and must not be a forward reference.

Example - Define Byte (DB):

```
NUM_BASE     DB    16
FILLER       DB    ?                          ;initialized with
                                              ;indetermined value
ONE_CHAR     DB    'M'
MULT_CHAR    DB    'MARC MIKE ZIBO PAUL BILL'
MSG          DB    'MSGTEST',13,10            ;message, carriage return,
                                              ;and linefeed
BUFFER       DB    10 DUP(?)                  ;indeterminate block
TABLE        DB    100DUP(5 DUP(4),7)
             ;100 copies of bytes with values 4,4,4,4,4,7
NEW_PAGE     DB    OCH                        ;form feed character
ARRAY        DB    1,2,3,4,5,6,7
```

Example - Define Word (DW):

```
ITEMS        DW    TABLE,TABLE+10,TABLE+20
SEGVAL       DW   0FFF0H
BSIZE        DW    4 * 128
LOCATION     DW    TOTAL + 1
AREA         DW    100 DUP(?)
CLEARED      DW    50 DUP(0)
SERIES       DW    2 DUP(2,3 DUP(BSIZE))
             ;two words with the byte values
             ;2,BSIZE,BSIZE,BSIZE,2,BSIZE,BSIZE,BSIZE
DISTANCE     DW    START_TAB - END_TAB
             ;difference of two labels is a constant
```

Example - Define Doubleword (DD):

```
DBPTR           DD    TABLE                    ;16-bit OFFSET, then 16-bit
                                               ;SEG base value
SEC_PER_DAY     DD    60*60*24                 ;arithmetic is performed
                                               ;by the assembler
LIST            DD    'XY',2 DUP(?)
HIGH            DD    4294967295               ;maximum
FLOAT           DD    6.735E2                  ;floating point
```

Example - Define Quadword (DQ):

```
LONG_REAL       DQ    3.141597                 ;decimal makes it real
STRING          DQ    'AB'                     ;no more than 2 characters
HIGH            DQ    18446744073709661615     ;maximum
LOW             DQ    -18446744073709661615    ;minimum
SPACER          DQ    2 DUP(?)                 ;uninitialized data
FILLER          DQ    1 DUP(?,?)               ;initialized with
                                               ;indeterminate value
HEX_REAL        DQ    0FDCBA9A98765432105R
```

Example - Define Tenbytes (DT):

```
ACCUMULATOR     DT    ?
STRING          DT    'CD'                     ;no more than 2 characters
PACKED-DECIMAL  DT    1234567890
FLOATING_POINT  DT    3.1415926
```

END

END [<exp>]

The END statement specifies the end of the program.

If <exp> is present, it is the start address of the program. If several modules are to be linked, only the main module may specify the start of the program with the END <exp> statement.

If <exp> is not present, then no start address is passed to LINK for that program or module.

Examples:

```
END    START ;START is a label somewhere in the program
```

EQU

<name> EQU <exp>

EQU assigns the value of <exp> to <name>. If <exp> is an external symbol, an error is generated. If <name> already has a value, an error is generated. If you want to be able to redefine a <name> in your program, use the equal sign (=) directive instead.

In many cases, EQU is used as a primitive text substitution, like a macro.

<exp> may be any one of the following:

1. A symbol. <name> becomes an aliaa for the symbol in <exp>. Shown as an Alias in the symbol table.
2. An instruction name. Shown as an Opcode in the symbol table.
3. A valid expression. Shown as a Number or L (label) in the symbol table.
4. Any other entry, including text, index references, segment prefix and operands. Shown as Text in the symbol table.

Examples:

```
FOO   EQU   BAZ                  ;must be defined in this
                                 ;module or an error results
B     EQU   [BP+8]               ;index reference (Text)
P8    EQU   DS:[BP+8]            ;segment prefix
                                 ;and operand (Text)
CBD   EQU   AAD                  ;an instruction name
(Opcode)
ALL   EQU   DEFREC<2,3,4>        ;DEFREC = record name
                                 ;<2,3,4> = initial values
                                 ;for fields of record
EMP   EQU   6                    ;constant value
FPV   EQU   6.3E7                ;floating point (text)
```

Equal Sign

<name> = <exp>

<exp> must be a valid expression. It is shown as a Number or L (label) in the symbol table (same as <exp> type 3 under the EQU directive above).

The equal sign (=) allows the user to set and to redefine symbols. The equal sign is like the EQU directive, except the user can redefine the symbol without generating an error. Redefinition may take place more than once and redefinition may refer to a previous definition.

Examples:

```
FOO  =    5              ;the same as FOO EQU 5
FOO  EQU  6;             ;error, FOO cannot be
                         ;redifined by EQU
FOO  =    7              ;FOO can be redefined
                         ;only by another =
FOO  =    FOO+3          ;redefinition may refer
                         ;to a previous definition
```

EVEN

EVEN

The EVEN command causes the program counter to go to an even boundary; that is, to an address that begins a word. If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP instruction so that the counter will reach an even boundary.

An error results if EVEN is used with a byte aligned segment.

Examples:

Before: The PC points to 0019 hex (25 decimal)

```
EVEN
```

After: The PC points to 1A hex (26 decimal) 0019 hex now contains an NOP instruction.

EXTRN

EXTRN <name>:<type>[,...]

<name> is a symbol that is defined in another module. <name> must have been declared PUBLIC in the module where <name> is defined.

<type> may be any one of the following, but must be a valid type for <name>:

1. BYTE, WORD, or DWORD

2. NEAR or FAR for labels or procedures (defined under a PROC directive)

3. ABS for pure numbers (implicit size is WORD, but includes BYTE).

Unlike the 8080 assembler, placement of the EXTRN directive is significant. If the directive is given with a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments the use either:

    ASUME <seg-reg>:SEG <name>

or an explicit segment prefix.

NOTE

If a mistake is made and the symbol is not in the segment, LINK will take the offset relative to the given segment, if possible. If the real segment is more tha 64K bytes away from the reference, LINK may find the definition. If the real segment is more than 64K bytes away, LINK will fail to make the link between the reference and the definition and will not return an error message.

Examples:

In Same Segment:	In Another Segment:
In Module 1:	**In Module 1:**

```
CSEG   SEGMENT                    CSEGA  SEGMENT
       PUBLIC TANG                       PUBLIC TAGF
       .                                 .
       .                                 .
       .                                 .
TAGN:                             TAGF:
       .                                 .
       .                                 .
       .                                 .
CSEG   ENDS                       CSEGA  ENDS
```

In Module 2:	**In Module 2**

```
CSEG   SEGMENT                           EXTRN TAGF:FAR
       EXTRN TAGE:NEAR            CSEGV  SEGMENT
       .                                 .
       .                                 .
       .                                 .
       JMP TAGN                   CSEGB  ENDS
CSEG   ENDS
```

GROUP

```
<name>    GROUP    <seg-name>[,...]
```

The GROUP directive collects the segments named after GROUP (<seg-name>s) under one name. The GROUP is used by LINK so that it knows which segments should be loaded together (the order the segments are named here does not influence the order the segments are loaded; that is handled by the CLASS designation of the SEGMENT directive, or by the order you name object modules in response to the LINK Object module prompt).

All segments in a GROUP must fit into 64K bytes of memory. The assembler does not check this at all, but leaves the checking to LINK.

<seg-name> may be one of the following:

1. A segment name, assigned by a SEGMENT directive. The name may be a forward reference.

2. An expression: either SEG <var>
   or SEG <label>
   Both of these entries resolve themselves to a segment name (see SEG operator, Section 3.2)

Once you have defined a group name, you can use the name:

1. As an immediate value:

   ```
   MOV AX,DGROUP
   MOV DS,AX
   ```

   DGROUP is the paragraph address of the base of DGROUP.

2. In ASSUME statement:

   ```
   ASSUME DS:DGROUP
   ```

   The DS register can now be used to reach any symbol in any segment of the group.

3. As an operand prefix (for segment override):

```
MOV BX,OFFSET DGROUP:FOO
DW DGROUP:FOO
DD DGROUP:FOO
```

DGROUP: forces the offset to be relative to DGROUP, instead of to the segment in which FOO is defined.

Example: (Using GROUP to combine segments):

In Module A:

```
CGROUP   GROUP     XXX,YYY
XXX      SEGMENT
         ASSUME    CS:CGROUP
         .
         .
         .
XXX      ENDS
YYY      SEGMENT
         .
         .
         .
YYY      ENDS
         END
```

In Module B:

```
CGROUP   GROUP     ZZZ
ZZZ      SEGMENT
         ASSUME    CS:CGROUP
         .
         .
         .
ZZZ      ENDS
         END
```

INCLUDE

INCLUDE <filename>

The INCLUDE directive inserts source code from an alternate assembly language source file into the current source file during assembly. Use of the INCLUDE directive eliminates the need to repeat an often-used sequence of statements in the current source file.

The <filename> is any valid file specification for the operating system. If the device designation is other than the default, the source filename specification must include it. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the INCLUDE directive statement. when end-of-file is reached, assembly resumes with the next statement following the INCLUDE directive.

Nested includes are allowed (the file inserted with an INCLUDE statement may contain an INCLUDE directive). However, this is not a recommended practice as a large amount of memory may be required.

The file specified must exist. If the file is not found, an error is returned, and the assembly aborts.

On a Macro Assembler listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See Section 1.5.4, Formats of Listings and Symbol Tables, for a description of listing file formats.

Examples:

```
INCLUDE ENTRY
INCLUDE B:RECORD.TST
```

LABEL

<name> LABEL <type>

By using LABEL to define a <name>, you cause the assebler to associate the current segment offset with <name>.

The item is assigned a length of 1.

<type> varies depending on the use of <name>. <name> may be used for code or for data.

1. For code: (for example, as a JMP or CALL operand)

   <type> may be either NEAR or FAR. <name> cannot be used in data manipulation instructions without using a type override.

   If you want, you can define a NEAR label using the <name>: form (the LABEL directive is not used in this case). If you are defining a BYTE or WORD NEAR label, you can place the <name>: in front of a Define directive.

   When using a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

   Example - For Code:

```
SUBRTF  LABEL    FAR
SUBRT:  (first instruction) ;colon = NEAR label
```

2. For data:

<type may be BYTE, WORD, DWORD, <structure-name>, or <record-name>. When STRUC or RECORD name is used, <name> is assigned the size of the structure or record.

Example - For Data:

```
BARRAY   LABEL   BYTE
ARRAY    DW      100 DUP (0)
         .
         .
         .
         ADD     AL,BARRAY [99]  ;ADD 100th byte to AL
         ADD     AX,BARRAY [98]  ;ADD 50th word to AX
```

By defining the array two ways, you can access entries either by byte or by word. Also, you can use this method for STRUC. If allows you to place your data in memory as a table, and to access it without the offset of the STRUC.

Defining the array two way also permits you to avoid using the PTR operator. The double defining method is especially effective if you access the data different ways. It is easier to give the array a second name than to remember to use .PRT.

NAME

NAME <module-name>

<module-name> must not be a reserved word. The module name may be any length, but Macro Assembler uses only the first six characters and truncates the rest.

The module name is passed to LINK, but otherwise has no significance for the assembler. Macro Assembler does check if more than one module name has been declared.

Every module has a name. Macro Assembler derives the module name from:

1. a valid NAME directive statement

2. If the module does not contain a NAME statement, Macro Assembler uses the first six characters of the TITLE directive statement. The first six characters must be legal as a name.

Examples:

```
NAME CURSOR
```

ORG

ORG <exp>

The location counter is set to the value of <exp>, and the assembler assigns genereated code starting with that value.

All names used in <exp> must be known on pass 1. The value of <exp> must either evaluate to an absolute or must be in the same segment as the location counter.

Example:

```
ORG      120H       ;2-byte absolue value
                    ;maximum=0FFFFH
ORG      $+2        ;skip two bytes
```

Example - ORG to a boundary (conditional):

```
CSEG     SEGMENT   PAGE
BEGIN    =         $
         .
         .
         .
IF ($-BEGIN) MOD 256                  ;if not already on
                                      ;256 byte boundary
         ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
ENDIG
```

See Section 1.4.2.2, Conditional Directives, for an explanation of conditional assembly.

PROC

```
<procname>  PROC   [NEAR]
                or FAR
               .
               .
               .
              RET
<procname>  ENDP
```

The default, if no operand is specified, is NEAR. Use FAR if:

> the procedure name is an operating system entry point
>
> the procedure will be called from code which has another ASSUME CS value.

The PROC block should contain a RET statement.

The PROC directive serves as a structuring device to make your programs more understandable.

The PROC directive, through the NEAR/FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL and RETs to generate a NEAR or a FAR RET. PROC is used, therefore, for coding simplification so that the user does not have to worry about NEAR or FAR for CALLs and RETs.

A NEAR CALL or RETURN changes the IP but not the CS register. A FAR CALL or RETURN changes both the IP and the CS registers.

Procedures are executed either in-line, from a JMP, or from a CALL.

PROCs may be nested, which means that they are put in line.

Combining the PUBLIC directive with a PROC statement (both NEAR and FAR), permits you to make external CALLs to the procedure or to make other external references to the procedure.

Examples:

```
              PUBLIC        FAR_NAME
FAR_NAME      PROC          FAR
              CALL          NEAR_NAME
              RET
FAR_NAME      ENDP

              PUBLIC        NEAR_NAME
NEAR_NAME     PROC          NEAR
              .
              .
              .
              RET
NEAR_NAME     ENDP
```

The second subroutine above can be called directly from a NEAR segment (that is, a segmen addressable through the same CS and within 64K):

```
CALL NEAR_NAME
```

A FAR segment (that is, any other segment that is not a NEAR segment) must call to the first subroutine, which then calls the second; an indirect call:

```
CALL FAR_NAME
```

PUBLIC

PUBLIC <symbol>[,...]

Place a PUBLIC directive statement in any module that contains symbols you wnat to use in other modules without defining the symbol again. PUBLIC makes the listed symbol(s), which are defined in the module where the PUBLIC statement appears, available for use by other modules to be linked with the module that defines the symbol(s). This information is passes to LINK.

<symbol> may be a number, a variable, a label including PROC labels).

<symbol> may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than 2 bytes.

Examples:

```
                PUBLIC     GETINFO
GETINFO         PROC       FAR
                PUSH       BP                    ;save caller's regis
                MOV        BP,SP                 ;get address parame
                                                 ;body of subroutine
                POP        BP                    ;restore caller's reg
                RET                              ;return to caller
GETINFO         ENDP
```

Example - Illegal PUBLIC:

```
                PUBLIC PIE_BALD,HIGH_VALUE
PIE_BALD        EQU          3.1416
HIGH_VALUE      EQU          999999999
```

The default input base (or radix) for all constants is decimal. The .RADIX directive permits you to change the input radix to any base in the range 2 to 16.

<exp> is always in decimal radix, regardless of the current input radix.

Example:

```
        MOV        BX,0FFH
        .RADIX     16
        MOV        BX,0FF
```

The two MOVs in this example are identical.

The .RADIX directive does not affect the generated code values placed in the .OBJ, .LST, or .CRF output files.

The .RADIX directive does not affect the DD, DQ, or DT directives. Numeric values entered in the expression of these directives are always evaluated as decimal unless a data type suffix is appended to the value.

Example:

```
              .RADIX   16
NUM_HAND      DT       773   ;773 = decimal
HOT_HAND      DQ       773Q  ;773 = octal here only
COOL_HAND     DD       773H  ;now 773 = hexadecimal
```

RECORD

<recordname> RECORD <fieldname>:<width>[=<exp>],[...]

<fieldname> is the name of the field. <width> specifies the number of bits in the field defined by <fieldname>. <exp> contains the initial (or default) value for the field. Forward references are not allowed in a RECORD statement.

<fieldname> becomes a value that can be used in expressions. When you use <fieldname> in an expression, its value is the shift count to move the field to the far right. Using the MASK operator with the <fieldname> returns a bit mask for that field.

<width> is a constant in the range 1 to 16 that specifies the number of bits contained in the field defined by <fieldname>. The WIDTH operator returns this value. If the total width of all declared fields is larger than 8 bits, then the assembler uses two bytes. Otherwise, only one byte is used.

The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is right shifted so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

For Example:

```
FOO RECORD HIGH:4,MID:3,LOW:3
```

Initially, the bit map would be:

```
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|<HIGH->|<MID>|<LOW>|
```

Totals bits >8 means use a word; but total bits <16 means right shift, place undeclared bits at high end of word. Thus:

```
 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0  <---MASK
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|           |<HIGH->|<MID>|<LOW>|
    not     |-------|----------->
  declared  | WIDTH |shift count|
```

<exp> contains the initial value for the field. If the field is at least 7 bits wide, the user can use an ASCII character as the <exp>.

For example:

```
HIGH:7='0'
```

To initialize records, use the same method used for DB. The format is:

```
[<name>] <recordname> <[exp] [,...]>
or
[<name>]        <recordname>        [<exp>
DUP(<[exp][,...]>)
```

The name is optional. When given, name is a label for the first byte or word of the record storage area.

The recordname is the name used as a label for the RECORD directive.

The exp (both forms) contains the values you want placed into the fields of the record. In the latter case, the parentheses and angle brackets are required only around the second exp (following DUP). If [exp] is left blank, either the default values applies (the value given in the original record definition), or the value is indeterminant (when not initialized in the original record definition). For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields.

For example:

```
FOO <,,7>
```

From the previous example, the 7 would be placed into the LOW field of the record FOO. The fields HIGH and MID would be left as declared (in this case, uninitialized).

Records may be used in expressions (as an operand) in the form:

```
recordname<[value[,...]]>
```

The value entry is optional. The angle brackets must be coded as shown, even if the optional values are not given. A value entry is the value to be placed into a field of the record. For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields, as shown above.

Examples:

```
FOO     RECORD    HIGH:5,MID:3,LOW:3
        .
        .
        .
BAX     FOO       <>  ;leave indeterminate here
JANE    FOO       10 DUP(<16,8>)  ;HIGH=16,MID=8
                  ;LOW=?
        .
        .
        .
        MOV       DX,OFFSET JANE[2]
                  ;get beginning record address
        AND       DX,MASK MID
        MOV       CL,MID
        SHR       DX,CL
        MOV       CL,WIDTH MID
```

SEGMENT

```
<segname>   SEGMENT [<align>] [<combine>] [<'class'>]
                .
                .
                .
<segname>   ENDS
```

At runtime, all instructions that generate code and data are in (separate) segments. Your program may be a segment, part of a segment, several segments, parts of several segments, or a combination of these. If a program has no SEGMENT statement, and LINK error (invalid object) will result at link time.

The <segment name> must be an unique, legal name. The segment name must not be a reserved word.

<align> may be PARA (paragraph - default), BYTE, WORD, or PAGE.

<combine> may be PUBLIC, COMMON, AT <exp>, STACK, MEMORY, or no entry (which defaults to not combinable, called Private in the LINK manual).

<class> name is used to group segments at link time.

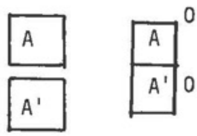
All three operands are passed to LINK.

The alignment tells the linker on what kind of boundary you want the segment to begin. The first address of the segment will be, for each aligment type:

```
PAGE - address is xxx00H (low byte is 0)
PARA - address is xxxx0H (low nibble is 0)
          bit map -  x x x x 0 0 0 0
WORD - address is xxxxeH (e=even number;low
       bit is 0)
          bit map -  x x x x x x x 0
BYTE - address is xxxxxH (place anywhere)
```

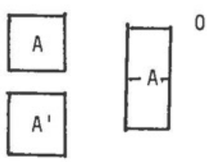
The combine type tells LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or Private)



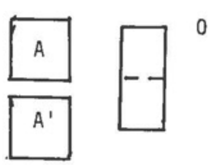
Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

Public and Stack



Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine type stack is treated the same as public. However, the Stack Pointer is set to the first address of the first stack segment. LINK requires at least one stack segment.)

Common



Common segments of the same name and class name are loaded overlapping on another. There is only one base address for all ocmmon segments of the same name. The length of the common area is the length of the longest segment.

Memory

Ostensibly, the memory combine type causes the segment(s) to be placed as the highest segments in memory. The first memory combinable segment encounter is placed as the highest segment in memory. Subsequent segments are treated the same as Common segments.

NOTE

> This feature is not supported by LINK. LINK treats Memory segments the same as Public segments.

AT <exp>

The segment is placed at the PARAGRAPH address specified in <exp>. The expression may not be a forward reference. Also, the AT type may not be used to force loading at fixed addresses. Rather, the AT combine type permits labels and variables to be defined at fiexed offests within fixed areas of storage, such as ROM or the vector space in low memory.

NOTE

> This restriction is imposed by LINK and DOS.

Class names must be enclosed in quotation marks. Class names may be any legal naem. Refer to LINK for more discussion.

Segment definitions may be nested. When segments are nested, the assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, the assembler takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. Overlapping segments are not permitted.

For example:

```
A SEGMENT
    .
    .
    .
B SEGMENT
    .
    .
    .
B ENDS
    .
    .
    .
A ENDS
```

```
A SEGMENT
    .
    .
    .
B SEGMENT
    .
    .
    .
    .
B ENDS
A SEGMENT
    .
    .
    .
A ENDS
```

The following arrangement is not allowed:

```
A SEGMENT
    .
    .
B SEGMENT
    .
    .
A ENDS                          : this is illegal!
    .
    .
B ENDS
```

Example:

In module A:

```
SEGA  SEGMENT  PUBLIC 'CODE'
      ASSUME   CS:SEGA
      .
      .
      .
SEGA  ENDS
      END
```

In module B:

```
SEGA  SEGMENT  PUBLIC 'CODE'
      ASSUME   CS:SEGA
      .        ; LINK adds this segment to same
      .        ;named segment in module A (and
      .        ;others) if class name is the same.
SEGA  ENDS
      END
```

STRUC

```
<structurename>     STRUC
                      .
                      .
                      .
<structurename>     ENDS
```

The STRUC directive is very much like RECORD, except STRUC has a multiple byte capability. The allocation and initialization of a STRUC block is the same as for RECORD.

Inside the STRUC/ENDS block, the Define directives (DB,DW,DD,DQ,DT) may be used to allocate space. The Define directives and comments set off by semicolons (;) are the only statement entries allowed inside a STRUC block.

Any label on a Define directive inside a STRUC/ENDS block becomes a <fieldname> of the structure. (This is how structure fieldnames are defined.) Initial values given to fieldnames in the STRUC/ENDS block are default values for the various fields. These values of the fields are one of two types: overridable or not overridable. A simple field, a field with only one entry (but not a DUP expression), is overridable. A multiple field, a field with more than one entry is not overridable. For example:

```
FOO DB 1,2         ;is not overridable
BAZ DB 10 DUP(?)   ;is not overridable
ZOO DB 5           ;is overridable
```

If the <exp> following the Define directive contains a string, it may be overriden by another string. However, if the overriding string is shorter than the initial string, the assembler will pad with spaces. If the overriding string is longer, the assembler will truncate the extra characters.

Usually, structure fields are used as operands in some expression. The format for a reference to a structure field is:

```
<variable>.<field>
```

<variable> represents an anonymous variable, usually set up when the structure is allocated. To allocate a structure, use the structure name as a directive with a label (the anonymous variable of a structure reference) and any override values in angle brackets:

```
FOO  STRUCTURE
     .
     .
     .
FOO  ENDS

GOO  FOO <,7,,'JOE'>
```

.<field> represents a label given to a DEFINE directive inside a STRUC/ENDS block (the period must be coded as shown). The value of <field> will be the offset within the addressed structure.

Examples:

To define a structure:

```
S       STRUC
FIELD1  DB      1,2             ;not overridable
FIELD2  DB      10 DUP(?)       ;not overridable
FIELD3  DB      5               ;overridable
FIELD4  DB      'DOBOSKY'       ;overridable
```

The Define directives in this example define the fields of the structure and the order corresponds to the order values are given in the initialization list when the structure is allocated. Every Define directive statement line inside a STRUC block defines a field, whether or not the field is named.

To allocate the structure:

```
DBAREA  S       <,,7,'ANDY'>    ;overrides 3rd and 4th
                                ;fields only
```

To refer to a structure:

```
MOV   AL,[BX].FIELD3
MOV   AL,DBAREA.FIELD3
```

1.4.2.2 Conditional Directives

Conditional directives allow users to design blocks of code which test for specific conditions then proceed accordingly.

All conditionals follow the format:

```
IFxxxx [argument]
   .
   .
   .
[ELSE
   .
   .
   .]
ENDIF
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a Code 8, Not in conditional block error.

Each conditional block may include the optional ELSE directive, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. An ELSE is always bound to the most recent, open IF. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a Code 7, Already had ELSE clause error.

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid Phase errors and incorrect evaluation. For IF and IFE the expression must involve values which were previously defined, and the expression must be Absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

The assembler evaluates the conditional statement to TRUE (which equals any non-zero value), or to FALSE (which equals 0000H). If the evaluation matches the condition defined in the conditional statement, the assembler either assembles the whole conditional block, or, if the conditional block contains the optional ELSE directive, assembles from IF to ELSE; the ELSE to ENDIF portion of the block is ignored. If the evaluation does not match, the assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE directive, assembles only the ELSE to ENDIF portion; the IF to ELSE portion is ignored.

IF <exp>

If <exp> evaluates to nonzero, the statements within the conditional block are assembled.

IFE <exp>

If <exp> evaluates to 0, the statements in the conditional block are assembled.

IF1 Pass 1 Conditional

If the assembler is in pass 1, the statements in the conditional block are assembled. IF1 takes no expression.

IF2 Pass 2 Conditional

If the assembler is in pass 2, the statements in the conditional block are assembled. IF2 takes no expression.

IFDEF <symbol>

If the <symbol> is defined or has been declared External, the statements in the conditional block are assembled.

IFNDEF <symbol>

If the <symbol> is not defined or not declared External, the statements in the conditional block are assembled.

IFB <arg>

The angle brackets around <arg> are required.

If the <arg> is blank (none given) or null (two angle brackets with nothing in between, <>), the statements in the conditional block are assembled.

IFB (and IFNB) are normally used inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy will be replaced by a parameter passed by the macro call. If the macro call does not specify a parameter to replace the dummy following IFB, the expression is blank, and the block will be assembled. (IFNB is the opposite case.) Refer to section 4.2.3, Macro Directives, for a full explanation.

IFNB <arg>

The angle brackets around <arg> are required.

If <arg> is not blank, the statements in the conditional block are assembled.

IFBN (and IFB) are normally used inside macro blocks. The expression following the IFNB directive is typically a dummy symbol. When the macro is called, the dummy will be replaced by a parameter passed by the macro call. If the macro call specifies a parameter to replace the dummy following IFNB, the expression is not blank, and the block will be assembled. (IFB is the opposite case.) Refer to section 1.4.2.3, Macro Directives for a full explanation.

IFIDN <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are requird.

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFIDN (and IFDIF) are normally used inside macro blocks. The expression following the IFIDN directive is typically two dummy symbols. When the macro is called, the dummys will be replaced by parameters passed by the macro call. If the amcro call specifies two identical parameters to replace the dummys, the block will be assembled. (IFDIF) is the opposite case.) Refer to section 1.4.2.3, Macro Directives, for a full explanation.

IFDIF <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

IFDIF and <IFIDN> are normally used inside macro blocks. The expression following the IFDIF directive is typically two dummy symbols. When the macro is called, the dummys will be replaced by parameters passed by the macro callg. If the macro call specifies two different parameters to replace the dummys, the block will be asesmbled. (IFIDN is the opposite case.)

ELSE

The ELSE directive allows you to generate alternate code when the opposite condition exists. May be used with any of the conditional directives. Only one ELSE is allowed for each IFxxxx conditional directive. ELSE takes no expression.

ENDIF

This directive terminates a conditional block. An ENDIF directive must be given for every IFxxxx directive used. ENDIF takes no expression. ENDIF closes the most recent, unterminated IF.

1.4.2.3 Macro Directives

The macro directives allow you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition directive or one of the repetition directives and end with the ENDM directive. All of the macro directives may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro directives of the Macro Assembler include:

macro definition:
    MACRO

termination:
    ENDM
    EXITM

unique symbols within macro blocks:
    LOCAL

undefine a macro:
    PURGE

repetitions:
    REPT (repeat)
    IRP (indefinite repeat)
    IRPC (indefinite repeat character)

The macro directives also include some special macro operators:

    &
    ;;
    !
    %

Macro Definition

```
<name> MACRO [<dummy>,...]
        .
        .
        .
       ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the amcro's definition

<name> is like a LABEL and conforms to the rules for forming symbols. After the macro has been defined, <name> is used to invoke the macro.

A <dummy> is formed as any other name is formed. A <dummy> is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. You should include all dummys used inside the macro block on this line. The number of dummys is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. Macro Assembler interprets a series of dummys the same as any list of symbol names.

NOTE

A dummy is always recognized exclusively as a dummy. Even if a register name (such as AX or BH) is used as a dummy, it will be replaced by a parameter during expansion.

One alternative is to list no dummys:

<name> MACRO

This type of macro block allows you to call the block repeatedly, even if you do not want or need to pass parameters to the block. In this case, the block will not contain any dummys.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable or symbol. When Macro Assembler evaluates a statement, it first looks at the macro table it builds during pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: Macro Assembler evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, Macro Assembler will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., -MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

```
<name> [<parameter>,...]
```

<name> is the <name> of the MACRO block. A <parameter> replaces a <dummy> on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter. For example:

```
FOO 1,2,3,4,5
```

passes five parameters to the macro, but:

```
FOO <1,2,3,4,5>
```

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummys in the MACRO definition. If there are more parameters than dummys, the extras are ignord. If there are fewer, the extra dummys will be made null. The assembled code will include the macro block after each macro call statement.

EXAMPLE:

```
GEN     MACRO    XX,YY,ZZ
        MOV      AX,XX
        ADD      AX,YY
        MOV      ZZ,AX
        ENDM
```

If you then enter a macro call statement:

```
GEN     DUCK,DON,FOO
```

assembly generates the statements:

```
MOV     AX,DUCK
ADD     AX,DON
MOV     FOO,AX
```

On your program listing, these statements will be preceded by a plus sign to indicate that they came from a macro block.

End Macro

ENDM

ENDM tells the assembler that the MACRO or Repeat block is ended

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM directive. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM also causes an error.

If you wish to be able to exit from a MACRO or repeat before expansion is completed, use EXITM.

Exit Macro

EXITM

The EXITM directive is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional directive.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

Examples:

```
FOO     MACRO   X
X       =       0
        REPT    X
X       =       X+1
        IFE     X-0FFH  ;test x
        EXITM   ;if true, exit REPT
        ENDIF
        DB      X
        ENDM
        ENDM
```

LOCAL

Local <dummy>[<dummy>...]

The LOCAL directive is allowed only inside a MACRO definition block. A LOCAL statement must precede all other types of statements in the macro definition.

When LOCAL is executed, the assembler creates a unique symbol for each <dummy> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ??0000 to ??FFFF. Users should avoid the form ??nnnn for their own symbols.

Example:

```
0000                FUN      SEGMENT
                             ASSUME  CS:FUN,DS:FUN
                    FOO      MACRO   NUM,Y
                             LOCAL   A,B,C,D,E
                    A:       DB      7
                    B:       DB      8
                    C:       DB      Y
                    D:       DW      Y+1
                    E:       DW      NUM+1
                             JMP     A
                             ENDM
                             FOO     0C00H,0BEH
0000  07          + ??0000:  DB      7
0001  08          + ??0001:  DB      8
0002  BE          + ??0002:  DB      0BEH
0003  00BF        + ??0003:  DW      0BEH+1
0005  0C01        + ??0004:  DW      0C00H+1
0007  EB F7       +          JMP     ??0000
                             FOO     03C0H,0FFH
0009  07          + ??0005:  DB      7
000A  08          + ??0006:  DB      8
000B  FF          + ??0007:  DB      0FFH
000C  0100        + ??0008:  DW      0FFH+1
000E  03C1        + ??0009:  DW      03C0H+1
0010  EB F7       +          JMP     ??0005
0012                FUN      ENDS
                             END
```

PURGE

PURGE <macro-name>[...]

PURGE deletes the definition of the macro(s) listed after it.

PURGE provides two benefits:

1. It frees text space of the macro body.

2. It returns any instruction mnemonics or directives that were redefined by macros to their original function.

3. It allows you to "edit out" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method allows you to use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you will not use in this program.

   It is not necessary to PURGE a macro before redefining it. Simply place another MACRO statement in your program, reusing the macro name.

Examples:

```
INCLUDE  MACRO.LIB
PURGE    MAC1
MAC1                      ;tries to invoke purged macro
                          ;returns a syntax error
```

Repeat Directives

The directives in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the Repeat directives and MACRO directive are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

2. MACRO allows parameters to be pased to the MACRO block when a MACRO is called; hence, parameters can be changed.

Repeat directive parameters must be assigned as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be peformed for every program execution, then Repeat directives are convenient. With the MACRO directive, you must call in the MACRO each time it is needed.

Note that each Repeat directive must be matched with the ENDM directive to terminate the repeat block.

Repeat

```
REPT <exp>
  .
  .
  .
ENDM
```

Repeat block of statements between REPT and ENDM <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains an External symbol or undefined operands, an error is generated.

Example:

```
          X  =     0
             REPT  10     ;generates DB 1 - DB 10
          X  =     X+1
             DB    X
             ENDM
```

assembles as:

```
0000         X  =     0
                REPT  10     ;generates DB 1 - DB 10
             X  =     X+1
                DB    X
                ENDM
0000'   01   +  DB    X
0001'   02   +  DB    X
0002'   03   +  DB    X
0003'   04   +  DB    X
0004'   05   +  DB    X
0005'   06   +  DB    X
0006'   07   +  DB    X
0007'   08   +  DB    X
0008'   09   +  DB    X
0009'   0A   +  DB    X
                      END
```

Indefinite Repeat

```
IRP <dummy>,<parameters inside angle brackets>
  .
  .
  .
ENDM
```

Parameters must be enclosed in angle brackets.

Parameters may be any legal symbol, string, numeric, or character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of <dummy> in the block. If a parameter is null (i.e., <>), the block is processed once with a null parameter.

Examples:

```
IRP     X,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

This example generates the same bytes (DB 1 - DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

```
FOO     MACRO     X
        IRP       Y,<X>
        DB        Y
        ENDM
        ENDM
```

When the macro call statement

```
FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion becomes:

```
IRP     Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

The angle brackets around the parameters are removed, and all items are passed as a single parameter.

Indefinite Repeat Character

```
IRPC <dummy>,<string>
  .
  .
  .
ENDM
```

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block.

Example:

```
IRPC    X,0123456789
DB      X+1
ENDM
```

This example generates the same code (DB 1 - DB 10) as the two previous examples.

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.

{ Ampersand concatenates text or symbols. (The & may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by &. To form a symbol from text and a dummy, put & between them.

For example:

```
ERRGEN     MACRO  X
ERROR&X    PUSH   BX
           MOV    BX,'&X'
           JMP    ERROR
           ENDM
```

The call ERRGEN A will then generate:

```
ERRORA:    PUSH   B
           MOV    BX,'A'
           JMP    ERROR
```

In Macro Assembler, the ampersand will not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, extra ampersands may be needed. You need to supply as many ampersands as there are levels of nesting.

For example:

```
Correct form                 Incorrect form

FOO    MACRO  X              FOO    MACRO  X
       IRP    Z,<1,2,3>             DB     Z,<1,2,3>
X&&Z   DB     Z              X&Z    Z
       ENDM                  ENDM
       ENDM                  ENDM
```

When called, for example, by FOO BAZ, the expansion would be (correctly in the left column, incorrectly in the right):

1. MACRO build, find dummies and change to dl

```
       IRP    Z,<1,2,3>             IRP    Z,<1,2,3>
dl&Z   DB     Z              dlZ    DB     Z
       ENDM                         ENDM
```

2. MACRO expansion, substitute parameter text for dl

```
       IRP    Z,<1,2,3>             IRP    Z,<1,2,3>
BAZ&Z  DB     Z              BAZZ   DB     Z
       ENDM                         ENDM
```

3. IRP build, find dummies and change to dl

```
BAZ&dl DB     dl             BAZZ   DB     dl
```

4. IRP expansion, substitute parameter text for dl

```
BAZ1   DB     1              BAZZ   DB     1
BAZ2   DB     2              BAZZ   DB     2
BAZ3   DB     3              BAZZ   DB     3

                               ;here it's an error,
                               ;multi-defined symbol
```

<text> Angle brackets cause Macro Assembler to treat the text between the angle brackets as a single literal. Placing either the parameters to a macro call or the list of parameters following the IRP directive inside angle brackets causes two results:

1. All text within the angle brackets are seen as a single parameter, even if commas are used.

2. Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows.

One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets around parameters as there are levels of nesting.

;; In a macro or repeat block, a comment preceded by two semicolons is not saved as a part of the expansion.

The default listing condition for macros is .XALL (see section 1.4.2.4, Listing Directives, below). Under the influence of .XALL, comments in macro blocks are not lisetd because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments macro and repeat blocks are saved and listed. This can be the cause of an out of memory error. To avoid this error, place double semicolons before comments inside macro and repeat blocks, unless you specifically want a comment to be retained.

! An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must evaluate to an absolute (non-relocatable) constant.

Example:

```
PRINTE  MACRO   MSG,N
        %OUT    * MSG,N *
        ENDM
SYM1    EQU     100
SYM2    EQU     200
        PRINTE  <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the string (SYM1 + SYM2) to be substituted for the dummy N. The result would be:

```
     %OUT    * SYM1 + SYM2 = (SYM1 + SYM2)
```

When the % is placed in front of the parameter, the assembler generates:

```
     %out    * SYM1 + SYM2 = 300 *
```

1.4.3 LISTING DIRECTIVES

Listing directives perform two general functions: format control and listing control. Format control directives allow the programmer to insert page breaks and direct page headings. Listing control directives turn on and off the listing of all or part of the assembled file.

PAGE

```
PAGE [<length>][,<width>]
PAGE [+]
```

PAGE with no arguments or with the optional [,+] argument causes the assembler to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

The PAGE directive with either the length or width arguments does not start a new listing page.

The value of <length>, if included, becomes the new page length (measured in lines per page) and must be in the range 10 to 255. The default page length is 50 lines per page.

The value of <width>, if included, becomes the new page width (measured in characters) and must be in the range 60 to 132. The default page width is 80 characters.

The plus sign (+) increments the major pge number and resets the minor page number to 1. Page numbers are in the form Major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Example:

```
    .
    .
    .
PAGE +    ;increment Major, set minor to 1
    .
    .
    .
PAGE 58,60    ;page length=58 lines,
              ;width=60 characters
```

TITLE

TITLE <text>

TITLE specifies a title to be listed on the first line of each page. The <text> may be up to 60 characters long. If more than one TITLE is given, an error results. The first six characters of the title, if legal, are used as the module name, unless a NAME directive is used.

Example:

```
TITLE PROG1 -- 1st Program
    .
    .
    .
```

If the NAME directive is not used, the module name is now PROG1 -- 1st program. This title text will appear at the top of every page of the listing.

SUBTITLE

SUBTTL <text>

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The <text> is truncated after 60 characters.

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the <text> from the previous SUBTTL with the <text> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <text>.

Example:

```
SUBTTL SPECIAL I/O ROUTINE
    .
    .
    .
SUBTTL
    .
    .
    .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

%OUT

%OUT <text>

The text is listed on the terminal during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

%OUT will output on both passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See Section 4.2.2, Conditional Directives, for descriptions of the IF1 and IF2 directives.

Example:

```
%OUT *Assembly half done*
```

The assembler will send this message to the terminal screen when encountered.

```
IF1
%OUT *Pass 1 started*
ENDIF

IF2
%OUT *Pass 2 started*
ENDIF
```

.LIST
.XLIST

.LIST lists all lines with their code (the default condition).

.XLIST suppresses all listing.

If you specify a listing file following the Listing prompt, a listing file with all the source statements included will be listed.

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing directives. So, nothing will be listed, even if another listing directive (other than .LIST) is encountered.

Example:

```
  .
  .
  .
.XLIST      ;listing suspended here
  .
  .
  .
.LIST       ;listing resumes here
```

.SFCOND

.SFCOND suppresses portions of the listing containing conditional expressions that evaluate as false.

.LFCOND

.LFCOND assures the listing of conditional expressions that evaluate false. This is the default condition.

.TFCOND

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch when running the assembler. When /X is used, .TFCOND will cause false conditionals to list. When /X is not used, .TFCOND will suppress false conditionals.

.XALL

.XALL is the default.

.XALL lists source code and object code produced by a macro, but source lines which do not generate code are not listed.

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code. Comments preceded by two semicolons (;;) will not be listed.

.SALL

.SALL suppresses listing of all text and object code produced by macros.

.CREF
.XCREF

.CREF
.XCREF [<variable list>]

.CREF is the default condition. .CREF remains in effect until Macro Assembler encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until Macro Assembler encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Use .CREF to restart the creation of a cross reference file after using the .XCREF directive.

If you include one or more variables following .XCREF, these variables will not be placed in the listing or cross reference file. All other cross referencing, however, is not affected by an .XCREF directive with arguments. Separate the variables with commas.

Neither .CREF nor .XCREF without arguments takes effect until you specify a cross reference file when running the assembler. .XCREF <variable list> suppresses the variables from the symbol table listing regardless of the creation of a cross reference file.

Examples:

```
.XCREF CURSOR,FOO,GOO,BAZ,ZOO
          ;these variables will not be
          ;in the listing or cross reference file
```

Section 5

ASSEMBLING A MACRO ASSEMBLER SOURCE FILE

Assembling with the Macro Assembler requires two types of commands: a command to invoke Macro Assembler and answers to command prompts. In addition, four switches control alternate Macro Assembler features. Usually, the user will enter all the commands to Macro Assembler on the keyboard. As an option commands, answers to the command prompts and any switches may be contained in a Batch file (see the Hyperion User Guide for Batch file instructions.). Some Command Characters are provided to assist the user while entering assembler commands.

1.5.1 INVOKING MACRO ASSEMBLER

Macro Assembler may be invoked two ways. By the first method, the user enters the commands as answers to individual prompts. By the second method, the user enters all commands on the line used to invoke Macro Assembler.

Summary of Methods to invoke Macro Assembler

Method 1	MASM
Method 2	MASM <source>,<object>,<listing>,<cross-ref>[/switch]

1.5.1.1 Method 1: MASM

Enter:

```
MASM
```

Macro Assembler will be loaded into memory. Then, Macro Assembler returns a series of four text prompts that appear one at a time. The user answers the prompts as commands to Macro Assembler to perform specific tasks.

At the end of each line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, Macro Assembler defaults to not performing the function described for the switches in the chart below.

The command prompts are summarized here and described in detail in Section 1.2.2, Command Prompts. Following the summary of prompts is a summary of switches, which are described in more detail in Section 1.2.3, Switches.

PROMPT	RESPONSES
Source filename [.ASM]:	List .ASM file to be assembled. (no default: filename response required)
Object filename [source.OBJ]	List filename for relocatable object code. (default: source-filename.OBJ]
Source listing [NUL.LST]:	List filename for listing (default: no listing file]
Cross reference [NUL.CRF]	List filename for cross reference file (used with CREF to create a cross reference listing). (default: no cross reference file)

SWITCH	ACTION
/D	Produces a listing on both assembler passes.
/O	Show generated object code and offsets in octal radix on listing
/X	Suppress the listing of false conditionals. Also used with the .TFCOND directive.

Command Characters

Macro Assembler provides two Command Characters.

; Use a single semicolon (;) followed immediately by a carriage return, at any time after responding to the first prompt (from Source filename on) to select default responses to the remaining prompts. This feature saves time and overrides the need to enter a series of carriage returns.

Note: Once the semicolon has been entered, the user can no longer respond to any of the prompts for that assembly. Therefore, do not use the semicolon to skip over some prompts. For this, use carriage return.

Example:
```
Source filename [.ASM]: FUN<Rtn>
Object filename [FUN.OBJ]:  ;<Rtn>
```

The remaining prompts will not appear, and Macro Assembler will use the default values (including no listing and no cross-reference file).

To achieve exactly the same result, you could alternatively enter:

```
Source filename [.ASM]: FUN;<Rtn>
```

This response produces the same files as the previous example.

Ctrl+Brk Use Ctrl+Brk at any time to abort the assembly. If you enter an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press Ctrl+Brk to exit Macro Assembler then reinvoke Macro Assembler and start over. If the error has been typed and not entered, you may delete the erroneous characters, but for that line only.

1.5.1.2 Method 2: MASM <filenames>[/switches]

Enter:

```
MASM <source>,<object>,<listing>,<cross-ref>[/switch]
```

Macro Assembler will be loaded into memory. Then Macro Assembler immediately begins assembly. The entries following MASM are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

where: <source> is the source filename.

<object> is the name of the file to receive the relocatable output.

<listing> is the name of the file to receive the listing.

<cross-ref> is the name of the file to receive the cross-reference output.

/switch are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <cross-ref>, as shown).

To select the default for a field, simply enter a second comma without space in between (see the example below).

Example: MASM FUN,,FUN/D/X,FUN

This example causes Macro Assembler to be loaded, then causes the source file FUN.ASM to be assembled. Macro Assembler then outputs the relocatable object ocde to a file named FUN.OBJ (default caused by two commas in a row), creates a listing file named FUN.LST for both assembly passes but with false conditionals suppressed, and creates a cross-reference file named FUN.CRF. If names were not listed for listing and cross-reference, these files would not be created. If listing file switches are given but no filename, the switches are ignored.

1.5.2 MACRO ASSEMBLER COMMAND PROMPTS

Macro Assembler is commanded by entering responses to four text prompts. When you have entered a reponse to the current prompt, the next appears. When the last prompts has been answered, Macro Assembler begins assembly automatically without further command. When assembly is finished, Macro Assembler exits to the operating system. When the operating system prompts is displayed, Macro Assembler has finished successfully. If the assembly is unsuccessful, Macro Assembler returns the appropriate error message.

Macro Assembler prompts the user for the names of source, object, listing, and cross-reference files.

All command prompts accept a file specification as a response. You may enter:

a filename only
a device designation only
a filename and an extension
a device designation and a filename
or a device designation, filename, and extension.

You may not enter only a filename extension.

<u>Source filename [.ASM]:</u>
Enter the filename of your source program. Macro Assembler assumes by default that the filename extension is .ASM, as shown in square brackets in the prompt text. If your source program has any other filename extension, you must enter it along with the filename. Otherwise, the extension may be omitted.

<u>Object filename [source.OBJ]:</u>
Enter the filename you want to receive the generated object code. If you simply press the carriage return key when this prompt appears, the object file will be given the same name as the source file, but with the filename extension .OBJ. If you want your object file to have a different name or a different filename extension, you must enter your choice(s) in response to this prompt. If you want to change only the filename but keep the .OBJ extension, enter the filename only. To change the extension only, you must enter both the filename and the extension.

Source listing [NUL.LST]:
Enter the name of the file, if any, you want to receive the source listing. If you press the carriage return key, Macro Assembler does not produce this listing file. If you enter a filename only, the listing is created and placed in a file with the name you enter plus the filename extension .LST. You may also enter your own extension.

The source listing file will contain a list of all the statements in your source program and will show the code and offsets generated for each statement. The listing will also show any error messages generated during the session.

Cross reference [NUL.CRF]:
Enter the name of the file, if any, you want to receive the cross reference file. If you press only the carriage return key, Macro Assembler does not produce this cross reference file. If you enter a filename only, the cross reference file is created and placed in a file with the name you enter plus the filename extension .CRF. You may also enter your own extension.

The cross reference file is used as the source file for the CREF Cross Reference Facility. CREF converts this cross reference file into a cross reference listing, which you can use to aid you during program debugging.

The cross reference file contains a series of control symbols that identify records in the file. CREF uses these control symbols to create a listing that shows all occurrences of every symbol in your program. The occurrence that defines the symbol is also identified.

1.5.3 MACRO ASSEMBLER COMMAND SWITCHES

The three switches control alternate assembler functions. Switches must be enetered at the end of a prompt response, regardless of which method is used to invoke Macro Assembler. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is entered at the end of one reponse, each switch must be preceded by the slash mark (/). You may not enter only a switch as a reponse to a command prompt.

Switch	Function
/D	Produce a source listing on both assembler passes. The listings will, when compared, show where in the program phase errors occur and will, possibly, give you a clue to why the errors occur. The /D switch does not take effect unless you command Macro Assembler to create a source listing (enter a filename in reponse to the source listing command prompt).
/O	Output the listing in octal radix. The generated code and the offsets shown on the listing will all be given in octal. The actual code in the object file will be the same as if the /O switch were not given. The /O switch affects only the listing file.
/X	Suppress the listing of false conditionals. If your program contains conditional blocks, the listing file will show the source statements but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not use generated code, use the /X switch to suppress the blocks that evaluate false from your listing. The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives.

If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, the second restores listing of false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When Macro Assembler encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing.

Of course, the /X switch has no effect if no listing is created. See additional discussion under the .TFCOND directive in Section 1.4.

The following chart illustrates the various effects of the conditional listing directives in combination with the /X switch.

PSEUDO-OP	NO /X	/X
(none)	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

Summary of Listing Symbols

R = linker resolves entry to left of R

E = External

---- = segment name, group name, or segment variable used in MOV AX, <---->, DD <---->, JMP <---->, and so on.

= = statement has an EQU or = directive

nn: = statement contains a segment override

nn/ = REPxx or LOCK prefix instruction. Example.

```
003C  F3/A5   REP  MOVSW  ;move DS:SI to ES:SI until CX=0
```

[= DUP expression; xx is the value in parentheses
 xx following DUP, for example: DUP(?) places ??
 where xx is shown here

\+ = line comes from a macro expansion

C = line comes from file named in INCLUDE directive statement.

ENTX   PASCAL entry for initializing programs

```
0000                         STACK       SEGMENT WORD STACK  'STACK'
= 0000                       HEAPbeg     EQU     THIS BYTE
      Indicates EQU or = directive

done                                             ;Base of heap before init
0000       14 [                          DB      20 DUP (?)
                ?? shows value in parentheses
                   ]
               Indicates DUP expression

= 0014                        SKTOP      EQU     THIS BYTE
0014                         STACK       ENDS


0000                          MAINSTARTUP  SEGMENT  'MEMORY'
                              DGROUP     GROUP   DATA,STACK,CONST,HEAP,MEMORY
                                         ASSUME  CS:MAINSTARTUP,DS:DGROUP
                                                 ES:DGROUP,SS:DGROUP

                                         PUBLIC  BEGX00  ;Main entry


0000                          ;
0000   B8   ---- R            BEGAX00    PROC     PAR
                                         MOV      AX,DGROUP
                                                  ;get assumed data segment
value
0003   8E D8                             MOV      DS,AX ;Set DS seg

0005   8C 06 0022 R                      MOV      CESX00,ES

       generated                name     action   expression    comment
         code
offset


000C   26: 8B 1E 0002                    MOV      BX,ES:2  ;highest paragraph
                       segment override
```

ENTX PASCAL entry for initializing programs

```
0011  2B D8                        SUB     BX,AX   ;Get #paras for D8
0013  81 FB 1000                   CMP     BX,4096   ;More than 64K?
0017  7E 03                        JLE     SMLSTK    ;No, use what we have
0019  BB 1000                      MOV     BX,4096   ;Can only address 64K

001C                     SMLSTK:   REPT    4
                                   SHL     BX,1
                                     ;Convert para to offset
                                   ENDM

001C  D1 E3                 +      SHL     BX,1
                                     ;Convert para to offset
001E  DI E3                 +      SHL     BX,1
                                     ;Convert para to offset
0020  D1 E3                 +      SHL     BX,1
                                     ;Convert para to offset
0022  D1 E3                 +      SHL     BX,1
                                     ;Convert para to offset

          macro     these lines    macro        number of
          block     from macro     directives   repetitions

0024  8B E3                        MOV     SP,BX
                                           ;Set stack to top of memory
                                           .
                                           .
                                           .
0069  EA 0000 ---- R               JMP     FAR PTR STARTmain

                 signal to linker                segment variable

              linker resolves: indicates segment name, group name
                 or segment variable used in MOV AX,<---->;
                 DD <---->; JMP <----> etc. (See other
                 example in this listing.)

006E                     BEGX00    ENDP

                                           .
                                           .
                                           .
007E                     MAINSTARTUP       ENDS

0000                     ENTXCM    SEGMENT WORD 'CODE'
                                   ASSUME  CS:ENTXCM
                                   PUBLIC  ENDX00,DOSX00
```

ENTX PASCAL entry for initializing program

```
0000                          STARTmain  PROC     FAR ;This code remains
0000   9A 0000 ---- E                    CALL     ENTBGOO
                                                  ;call main program

0005                          ENDXOO     LABEL    FAR
                                                    ;termination entry point
0005   9A 0000 ---- E                    CALL     ENDOO
                                                    ;user system termination
000A   9A 0000 ---- E                    CALL     ENDYOO
                                                    ;close all open files
000F   9A 0000 ---- E                    CALL     ENDUOO
                                                    ;file system termination
0014  C7 06 0020 R 0000                  MOV      DOSOFF,0

offset

                   linker       External
                   signal;       symbol
                   goes with
                   number to left: shows DOSOFF is in segment
                                                    ;return to DOS
00 2E 0020 R                  STARTmain  ENDP

                                         .
                                         .
                                         .
0037                          ENTXCM     ENDS
                                         END      BEGXOO
```

Differences Between Pass 1 Listing and Pass 2 Listing

If you give the /D switch when you run Macro Assembler to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that was assembled without reporting any errors. When the source file was reassembled using the /D switch, an error was produced on pass 1, but not on pass 2 (which is when errors are usually reported).

Example:

During Pass 1 a jump with a forward reference:

```
0017   7E 00                           JLE       SMLSTK  ;No, use what we have
 E r r o r ---              9: Symnbol not defined
0019   bb 1000                         MOV       BX,4096 ;Can only address 64K
001C                        SMLSTK:    REPT      4
```

During Pass 2 this same instruction is fixed up and does not return an error:

```
0017   7E 03                           JLE       SMLSTK  ;No, use what we have
0019   BB 1000                         MOV       BX,4096 ;Can only address 64K
```

Notice that the JLE instructions code now contains 03 instead of 00, a jump of 3 bytes.

The same amount of code was producing during both passes, so there was no phase error. The only different is one of content instead of size, in this case.

1.5.4.3 Symbol Table Format

The symbol table portionof a listing separates all "symbols" into their respective categories, showing appropriate descriptive data. This data gives you an idea how your program is using various symbolic values. Use this information to help you debug.

Also, you can use a cross reference listing, produced by CREF, to help you locate uses of the various "symbols" in your program.

On the next page is a complete symbol table listing. Following the complete listing, sections from different symbol tables are shown with explanatory notes.

For all sections of symbol tables: this rule applies: if there are no symbolic values in your program for a particular category, the heading for the category will be omitted from the symbol table listing. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

```
Assembler    date    PAGE    Symbols - 1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXMP1M.ASM)

Macros

              Name         Length

BIOSCALL . . . . . . .     0002
DISPLAY. . . . . . . .     0005
DOSCALL. . . . . . . .     0002
KEYBOARD . . . . . . .     0003
LOCATE . . . . . . . .     0003
SCROLL . . . . . . . .     0004


Structures and Records:

              Name        Width    #fields
                          Shift    Width  Mask  Initial

PARMLIST . . . . . . .     001C
  BUFSIZE. . . . . . .     0000
  NAMESIZE . . . . . .     0001
  NAMETEXT . . . . . .     0002
  TERMINATOR . . . . .     001B


Segments and Groups:

              Name         Size    Align    combine    class

CSEG . . . . . . . . .     0044    PARA     PUBLIC     'CODE'
STACK. . . . . . . . .     0200    PARA     STACK      'STACK'
WORKAREA . . . . . . .     0031    PARA     PUBLIC     'DATA'


Symbols:

              Name         type     Value     Attr

CLS. . . . . . . . . .     N PROC   0036      CSEG     Length =000E
MAXCHAR. . . . . . . .     Number   0019
MESSG. . . . . . . . .     L BYTE   001C      WORKAREA
PARMS. . . . . . . . .     L 001C   0000      WORKAREA
RECEIVR. . . . . . . .     L FAR    0000               External
START. . . . . . . . .     F PROC   0000      CSEG     Length =0036


Warning   Severe
Errors    Errors
0         0
```

Macros:

```
              Name       Length <----- number of 32 byte block
                                       macro occupies
BIOSCALL. . . . . . .    0002          in memory
DISPLAY . . . . . . .    0005
DOSCALL . . . . . . .    0002
KEYBOARD. . . . . . .    0003
LOCATE. . . . . . . .    0003
SCROLL. . . . . . . .    0004
  ^
  |
names of macros
```

This section of the symbol table tells you the names of your macros and how big they are in 32 byte block units. In this listing, the macro DISPLAY is 5 block long or (5 x 32 bytes =) 160 bytes long.

Structures and records:

Examples for Structures

```
                  Name        Width    # fields                        This line
                              Shift    Width    Mask    Initial  <---  for fields
PARMLIST . . . . . .          001C     0004                            (indented)
  BUFSIZE. . . . . .          0000
  NAMESIZE . . . . .          0001
  NAMETEXT . . . . .          0002
  TERMINATOR . . . .          001B                            Number of fields in
                                                               Structure
  field name of             Offset of field
  PARMLIST Structure          into structure
                                                            The number of bytes
                                                          wide of structure
```

```
Example for Records
                  Name        Width    # fields                        This line
                              Shift    Width    Mask    Initial        for fields
BAZ. . . . . . . . .      --> 0008     0003 <------------------Number of fields in record
  FLD1 . . . . . . .          0006     0002     00C0    0040
  FLD2 . . . . . . .          0003     0003     0038    0000 <---initial value
  FLD3 . . . . . . .          0000     0003     0007    0003
BAZ1 . . . . . . . .      --> 000B     0002                      MASK of field
  BZ1. . . . . . . .          0003     0008     07F8    0400        (maximum value)
  BZ2. . . . . . . .          0000     0003     0007    0002

             number of             shift     number of bits
        bits in record             count       if field
                                 to right
```

This section lists your Structures and/or Records and their fields. The upper line of column headings applies to Structure names, Record names, and to field names of Structures. The lower line of column headings applies to field names of records.

For structures:

Width (upper line) shows the number of bytes your Structure occupies in memory.
fields shows how many fields comprise your Structure.

For Records:

Width (upper line) shows the number of bits the Record occupies.

fields shows how many fields comprise your Record.

For Fields of Structures:

Shift shows the number of bytes the field is offset into the Structure.

The other columns are not used for fields of Structures.

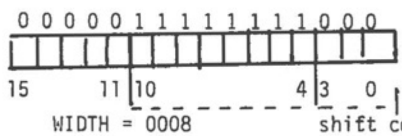
For Fields of Records

Shift is the shift count to the right

Width (lower line) shows the number of bits this field occupies.

Mask shows the maximum value of record, expressed in hexadecimal, if one field is masked and ANDed (field is set to all 1's and all other fields are set to all 0's).

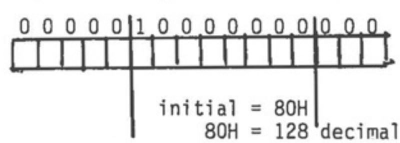
Using field BZ1 of the Record BAZ1 above to illustrate:



Initial shows the value specified as the initial value for the field, if any.

When naming the field you specified: fieldname:# = value

fieldname is the name of the field. # is the width of the field in bits. value is the initial value you want this field to hold. The symbol table shows this value as if it is placed in the field and all other fields are masked (equal 0). Using the example and diagram from above:



Initial = 0400

Segments and groups:

```
             Name          Size  align  combine class
                                                 _called Private
                                                /  in LINK section
AAAX00 . . . . . . . .     0000  WORD   NONE    'CODE <------segment
DGROUP . . . . . . . .     GROUP <----------------------------group
  DATA . . . . . . . .     0024  WORD   PUBLIC   'DATA'    |
  STACK. . . . . . . .     0014  WORD   STACK    'STACK'   |      segments
  CONST. . . . . . . .     0000  WORD   PUBLIC   'CONST'   |------of
  HEAP . . . . . . . .     0000  WORD   PUBLIC   'MEMORY'  |      DGROUP
  MEMORY . . . . . . .     0000  WORD   PUBLIC   'MEMORY'  |
ENTXCM . . . . . . . .     0037  WORD   NONE     'CODE'
MAIN_STARTUP . . . .       007E  PARA   NONE     'MEMORY'
                           ----  ----------------------
                            |               |
                          length     statement line entries
                            of
                         segments
```

For Groups:
the name of the group will appear under the Name column, beginning in column 1 with the applicable Segment names indented 2 spaces. The word Group will appear under the size column.

For Segments:
the segment names may appear in column 1 (as here) if you do not declare them part of a group. If you declare a group, the segment names will appear indented under their group name.

For all Segments, whether pat of a group or not:

Size is the number of bytes the Segment occupies.

Align is the type of boundary where the segment begins:

> PAGE = page - address is xxx00H (low byte = 0); begins on a 256 byte boundary.
> PARA = paragraph - address is xxxx0H (low nibble = 0); default
> WORD = word - address is xxxxeH (e = even number)
> low bit of low byte = 0)
> bit map - x x x x x x x 0
> BYTE = byte - address is xxxxxH (anywhere)

Combine describes how LINK (Linker Utility) will combine the various segments.

Class is the class name under which LINK will combine segments in memory.

Symbols:

```
            Name             Type     Value   Attr

FOO. . . . . . . . . .       Number   0005
FOO1 . . . . . . . . .       Text     1.234
FOO2 . . . . . . . . .       Number   0008
FOO3 . . . . . . . . .       Alias    FOO
FOO4 . . . . . . . . .       Text     5[BP][DI]
FOO5 . . . . . . . . .       Opcode
```

Symbols:

```
            Name             Type      Value    Attr

BEGH00 . . . . . . . .       L WORD    0012     DATA    Global
BEG000 . . . . . . . .       L FAR     0000       External
BEGX00 . . . . . . . .       F PROC    0000     MAIN_STARTUP  Global Length = 006E
CESX00 . . . . . . . .       L WORD    0022     DATA    Global
CLNE00 . . . . . . . .       L WORD    0002     DATA    Global
CRCX00 . . . . . . . .       L WORD    001C     DATA    Global
CRDX00 . . . . . . . .       L WORD    001E     DATA    Global
CSXE00 . . . . . . . .       L WORD    0000     DATA    Global
CURH00 . . . . . . . .       L WORD    0014     DATA    Global
DOSOFF . . . . . . . .       L WORD    0020     DATA
DOSX00 . . . . . . . .       F PROC    001E     ENTXCM  Global Length = 0019
ENDH00 . . . . . . . .       L WORD    0016     DATA    Global
END000 . . . . . . . .       L FAR     0000             External
ENDU00 . . . . . . . .       L FAR     0000             External
ENDX00 . . . . . . . .       L FAR     0005     ENTXCM  Global
ENDY00 . . . . . . . .       L DFAR    0000             External
ENTG00 . . . . . . . .       L FAR     0000             External
FREX00 . . . . . . . .       F PROC    006E     MAIN_STARTUP Global Length = 0010
HDRF00 . . . . . . . .       L WORD    0006     DATA    Global
HDRV00 . . . . . . . .       L WORD    0008     DATA    Global
HEAPBEG. . . . . . . .       BYTE      0000     STACK           EQU statements
INIU00 . . . . . . . .       L FAR     0000             External showing segment
PNUX00 . . . . . . . .       L WORD    0004     DATA    External
RECE00 . . . . . . . .       L WORD    0004     DATA    Global
REFE00 . . . . . . . .       L WORD    000C     DATA    Global
REPE00 . . . . . . . .       L WORD    000E     DATA    Global
RESE00 . . . . . . . .       L WORD    000A     DATA    Global
SKTOP. . . . . . . . .       BYTE      0014     STACK
SMLSTK . . . . . . . .       L NEAR    001C     MAIN_STARTUP
STARTMAIN. . . . . . .       F PROC    0000     ENTXCM  Length = 001E
STKB00 . . . . . . . .       L WORD    0018     DATA    Global
STKH00 . . . . . . . .       L WORD    001A     DATA    Global
```

If Macro Assembler knows this length as one of the type lengths (BYTE, WORD, DWORD, QWAORD, TBYTE), it shows that type name here.

This section lists all other symbolic values in your program that do not fit under the other categories.

Type shows the symbol's type:

L = Label
F = Far
N = Near
PROC = Procedure
Number
Alias All defined by EQU or = directive
Text
Opcode

These entries may be combined to form the various types shown in the example.

For all procedures, the length of the procedure is given after its attribute (segment)

You may also see an entry under type like:

L 0031

This etnry results from code such as the following:

BAZ LABEL FOO

where FOO is a STRUC that is 31 bytes long.

BAZ will be shown in the symbol table with the L 0031 entry. Basically, Number (and some other similar entries) indicates that the symbol was defined by an EQU or = directive.

Value (usually) shows the numeric value the symbol represents. (In some cases, the Value column will show some text -- when the symbol was defined by EQU or = directive.)

Attr always shows the segment of the symbol, if known. Otherwise, the Attr column is blank. Following the segment name, the table will show either External, Global or a blank (which means not declared with either the EXTRN or PUBLIC directive). The last entry applies to PROC types only. This is a length = entry, which is the length of the procedure.

If type is **Number** , Opcode, Alias, or Text, the Symbols section of the listing will be structured differently. Whenever you see one of these four entries under type, the symbol was created by an EQU directive or an = directive. All information that follows one of these entries is considered its "value', even if the "value" is simply text.

Each of the four types shows a value as shown:

Number shows a constant numeric value

Opcode shows a blank. The symbol is an alias for an instruction mnemonic. Sample directive statement: FOO EQU ADD

Alias shows a symbol name which the named symbol equals. Sample directive statement: FOO EQU BAX

Text shows the "text" the symbol repreesnts. "Text" is any other operand to an EQU directive that does not fit one of the other three categories above. Sample directive statements: GOO EQU 'WOW' BAZ EQU DS:8[BX] ZOO EQU 1.234

Section 6

Macro Assembler Messages

Most of the messages output by Macro Assembler are error messages. The nonerror messages output by Macro Assembler are the banner messages, and the end of (successful) assembly message. These nonerror messages are classified here as operating messages. The error messages are classified as assembler errors, I/O handler errors, and runtime errors.

1.6.1 OPERATING MESSAGES

Banner Message and Command Prompts:

```
MACRO-86 v1.0 Copyright (C) Microsoft, Inc.

Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing [NUL.LST]:
Cross reference [NUL.CRF]:
```

End of Assembly Message:

```
Warning   Fatal
Errors    Errors
n         n           (n=number of errors)

(the system prompt)
```

1.6.2 ERROR MESSAGES

If the assembler encounters errors, error messages are output, along with the numbers of warning and fatal errors, and control is returned to your disk operating system. The message is output either to your terminal screen or to the listing file if you command one to be created.

Error messages are divided into three categories: assembler errors, I/O handler errors, and runtime errors. In each category, messages are listed in alphabetical order with a short explanation where necessary. At the end of this chapter, the error messages are listed in a single numerical order list but without explanations.

Assembler Errors

Already defined locally (Code 23)

Tried todefine a symbol as EXTERNAL that had already been defined locally.

Already had ELSE clause (code 7)

Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

Already have base register (Code 46)

Trying to double base register.

Already have index register (Code 47)

Trying to double index address

Block nesting error (Code 0)

Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is close of an outer level of nesting with inner level(s) still open.

Byte register is illegal (Code 58)

Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

Can't override ES segment (Code 67)

Trying to override the ES segment in an instruction where this override is not legal. For example, store string.

Can't reach with segment reg (Code 68)

There is no assume that makes the variable reachable.

Can't use EVEN on BYTE segment (Code 70)

Segment was declared to be byte segment and attempt to use EVEN was made.

Circular chain of EQU aliases (Code 83)

An alias EQU eventually points to itself.

Constant was expected (Code 42)

Expecting a constant and received something else.

CS register illegal usage (Code 59)

Trying to use the CS register illegally. For example, XCHG CS,AX.

Directive illegal in STRUC (Code 78)

All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

Division by 0 or overflow (Code 29)

An expression is given that results in a divide by 0.

DUP is too large for linker (Code 74)

Nesting of DUPs was such that too large a record was created for the linker.

Extra characters on line (Code 1)

This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

Field cannot be overridden (Code 80)

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Forward needs override (Code 71)

This message not currently used

Forward reference is illegal (Code 17)

Attempt to forward reference something that must be defined in pass 1.

Illegal register value (Code 55)

The register value specified does not fit into the "reg" field (the reg field is greater than 7).

Illegal size for item (Code 57)

Size of referenced item is illegal. For exmple, shift of a double word.

Illegal use of external (Code 32)

Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

Illegal use of register (Code 49)

Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

Illegal value for DUP count (Code 72)

DUP counts must be a constant that is not 0 or negative.

Improper operand type (Code 52)

Use of an operand such that the opcode cannot be generated.

Index displ. must be constant (Code 54)

Label can't have seg. override (Code 65)

Illegal use of segment override.

Left operand must have segment (Code 38)

Use something in right operand that required a segment in the left operand. (For example, ":.")

More values than defined with (Code 76)

Too many fields given in REC or STRUC allocation.

Must be associated with code (Code 145)

Use of data related item where code item was expected.

Must be associated with data (Code 44)

Use of code related item where data related item was executed. For example, MOV AX,<code-label>.

Must be AX or AL (Code 60)

Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

Must be index or base register (Code 48)

Instruction requires a base or index register and some other register was specified in square brackets [ ].

Must be declared in pass 1 (Code 13)

Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

Must be in segment block (Code 69)

Attempt to generate code when not in a segment

Must be record field name (Code 33)

Expecting a record field name but got something else.

Must be record or field name (Code 34)

Expecting a record name or field name and received something else.

Must be register (Code 18)

Register unexpected as operand but user furnished symbol -- was not a register.

Must be segment or group (Code 20)

Expecting segment or group and something else was specified.

Must be structure field name (Code 37)

Expecting a structure field name but received something else.

Must be symbol type (Code 22)

Must be WORD, DW, DU, BYTE, or TB but received something else.

Must be var, label or constant (Code 36)

Expecting a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)

Use of one of the prefix instructions without specifying any opcode after it.

Near JMP/CALL to different CS (Code 64)

Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)

Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

No or unreachable CS (Code 62)

Trying to jump to a label that is unreachable.

Normal type operand expected (Code 41)

Received STRUCT, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Not proper align/combine type (Code 25)

SEGMENT parameters are incorrect.

One operand must be const (Code 39)

This is an illegal use of the addition operator.

Only initialize list legal (Code 77)

Attempt to use STRUC name without angle brackets, < >.

Operand combination illegal (Code 63)

Specification of a two-operand instruction where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)

Illegal use of the subtraction operator.

Operand must have segment (Code 43)

Illegal use of SEG directive.

Operand must have size (Code 35)

Expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)

Access of operand is impossible because it is not in the current IP segment.

Operand types must match (Code 31)

Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

Operand was expected (Code 27)

Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)

Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)

In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

Override with DUP is illegal (Code 79)

In a STRUC initialization statement, you tried to use DUP in an override.

Phase error between passes (Code 6)

The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes.

Redefinition of symbol (Code 4)

This error occurs on pass 2 and succeeding definitions of a symbol.

Reference to mult defined (Code 26)

The instruction references something that has been multi-defined.

Register already defined (Code 2)

This will only occur if the assembler has internal logic errors.

Register can't be forward ref (Code 82)

Relative jump out of range (Code 53)

Relative jumps must be within the range -128 +127 of the current instruction, and the specific jump is beyond this range.

Segment parameters are changed (Code 24)

List of arguments to SEGMENT were not identical to the first time this segment was used.

Shift count is negative (Code 30)

A shift expression is generated that results in a negative shift count.

Should have been group name (Code 13)

Expecting a group name but something other than this was given.

Symbol already different kind (Code 15)

Attempt to define a symbol differently from a previous definition.

Symbol already external (Code 73)

Attempt to define a symbol as local that is already external.

Symbol has no segment (Code 21)

Trying to use a variable with SEG, and the variable has no known segment.

Symbol is multi-defined (Code 5)

This error occurs on a symbol that is later redefined.

Symbol is reserved word (Code 16)

Attempt to use an assembler reserved word illegally. (For example, to declare MOV as a variable.)

Symbol not defined (Code 9)

A symbol is used that has no definition.

Symbol type usage illegal (Code 14)

Illegal use of a PUBLIC symbol.

Syntax error (Code 10)

The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)

The type specified is of an unacceptable size.

Unknown symbol type (Code 3)

Symbol statement has something in the type field that is unrecognizable.

Usage of ? (indeterminate) bad (Code 75)

Improper use of the "?". For example, ? + 5.

Value is out of range (Code 50)

Value is too large for expected use. For example, MOV AL,5000.

Wrong type of register (Code 19)

Directive or instruction expected one type of register, but another was specified. For example, INC CS.

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 2 - LINK

2.1 INTRODUCTION

Features and Benefits of LINK

LINK is a relocatable linker designed to link together separately produced program files.

For all the necessary and optional commands, LINK prompts the user. The user's answers to the prompts are the commands for LINK.

The output file from LINK (Run file) is not bound to specific memory addresses and, therefore, can be loaded and executed at any convenient address by the user's operating system.

LINK uses a dictionary-indexed library search method, which substantially reduces link time for sessions involving library searches.

LINK is capable of linking files totaling 384K bytes

LINK combines several object modules into one relocatable load module, or Run file.

As it combines modules, LINK resolves external references between object modules and can search multiple library files for definitions for any external references left unresolved.

LINK also produces a list file that shows external references resolved and any error messages.

LINK uses available memory as much as possible. When available memory is exhausted, LINK then creates a disk file and becomes a virtual linker.

2.2 DEFINITIONS

Three terms will appear in some of the error messages listed in Section 3.9. These terms describe the underlying functioning of LINK. An understanding of the concepts that define these terms provides a basic understanding of the way LINK works.

1. **Segment**

A Segment is a contiguous area of memory up to 64K bytes in length. A Segment may be located anywhere in 8086 memory on a "paragraph" (16 byte) boundary. The contents of a Segment are addressed by a Segment-register/offset pair.

2. **Group**

A Group is a collection of Segments which fit within 64K bytes of memory. The Segments are named to the Group by the assembler, by the compiler, or by you. The Group name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler.

The Group is used for addressing Segments in memory. Each Group is addressed by a single Segment register. The Segments within the Group are addressed by the Segment register plus an offset. LINK checks to see that the object modules of a Group meet the 64K byte constraint.

3. **Class**

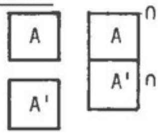
A Class is a collection of Segments. The naming of Segments to a Class controls the order and relative placement of Segments in memory. The Class name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler. The Segments are named to a Class at compile time or assembly time. The Segments of a Class are loaded into memory contiguously. The Segments are ordered within a Class in the order LINK encounters the Segments in the object files. One Class precedes another in memory only if a Segment for the first Class precedes all Segments for the second Class in the input to LINK. Classes may be loaded across 64K byte boundaries. The Classes will be divided into Groups for addressing.

How LINK Combines and Arranges Segments

LINK works with four combine types, which are declared in the source module for the assembler or compiler: private, public, stack, and common. LINK does not automatically place memory combine type as the highest segments.

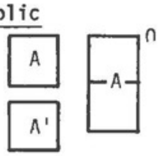
LINK combines segments for these combine types as follows:

Private



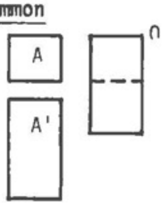
Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

Public



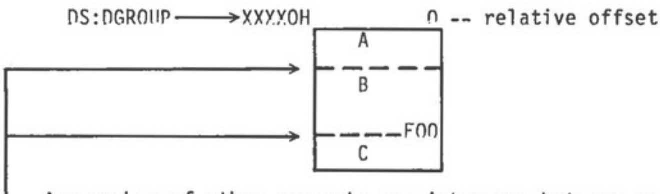
Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine types stack and memory are treated the same as public. However, the Stack Pointer is set to the first address of the first stack segment.)

Common



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Placing segments in a Group in the assembler provides offset addressing of items from a single base address for all segments in that Group.



Any number of other segments may intervene between segments of a group. Thus, the offset of FOO may be greater than the size of segments in group combined, but no larger than 64K.

An operand of DGROUP:FOO returns the offset of FOO from the beginning of the first segment of DGROUP (segment A here).

Segments are grouped by declared class names. LINK loads all the segments belonging to the first class name encountered, then loads all the segments of the next class name encountered, and so on until all classes have been loaded.

If your program contains:

```
A SEGMENT 'FOO'
B SEGMENT 'BAZ'
C SEGMENT 'BAZ'
D SEGMENT 'ZOO'
E SEGMENT 'FOO'
```

They will be loaded as:

```
'FOO'
   A
   E
'BAZ'
   B
   C
'ZOO'
   D
```

If you are writing assembly language programs, you can exercise control over the ordering of classes in memory by writing a dummy module and listing it first after the LINK Object Modules prompt. The dummy module declares segments into classes in the order you want the classes loaded.

WARNING

Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs. Allow the compiler and the linker to perform their tasks in the normal way.

For example:

```
A  SEGMENT   'CODE'
A  ENDS
B  SEGMENT   'CONST'
B  ENDS
C  SEGMENT   'DATA'
C  ENDS
D  SEGMENT   'STACK'
D  ENDS
E  SEGMENT   'MEMORY'
E  ENDS
```

You should be careful to declare all classes to be used in your program in this module. If you do not, you lose absolute control over the ordering of classes.

Also, if you want Memory combine type to be loaded as the last segments of your program, you can use this method. Simply add MEMORY between SEGMENT and 'MEMORY' in the E segment line above. Note, however, that these segments are loaded last only because you imposed this control on them, not because of any inherent capability in the linker or assembler operations.

2.3 FILES THAT LINK USES

LINK works with one or more input files, produces two output files, may create a virtual memory file, and may be directed to search one to eight library files. For each type of file, the user may give a three part file specification. The format for LINK file specifications is:

drv:filename.ext

where: **drv:** is the drive designation. Permissible drive designations for LINK are A: through O:. The colon is always required as part of the drive designation.

filename is any legal filename of one to eight characters.

ext is a one to three character extension to the filename. The period is always required as part of the extension.

Input Files

If no extensions are given in the input (Object) file specifications, LINK recognizes by default:

File	Default Extension
Object	.OBJ
Library	.LIB

Output Files

LINK appends to the output (Run and List) files the following default extensions:

File	Default Extension
Run	.EXE (may not be overridden)
List	.MAP (may be overridden)

2.4 VM.TMP File

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary file and names it VM.TMP. If LINK needs to create VM.TMP, it displays the message:

```
VM.TMP has been created
Do not change diskette in drive, <drv:>
```

Once this message is displayed, the user must not remove the diskette from the default drive until the link session ends. If the diskette is removed, the operation of LINK is unpredictable, and LINK might return the error message:

```
Unexpected end of file on VM.TMP
```

LINK uses VM.TMP as a virtual memory. The contents of VM.TMP are subsequently written to the file named following the Run File: prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

WARNING

Do not use VM.TMP as a file name for any file. If the user has a file named VM.TMP on the default drive and LINK requires the VM.TMP file, LINK will delete the VM.TMP on disk and create a new VM.TMP. Thus, the contents of the previous VM.TMP file will be lost.

2.5 RUNNING LINK

Running LINK requires two types of commands: a command to invoke LINK and answers to command prompts. In addition, six switches control alternate LINK features. Usually, the user will enter all the commands to LINK on the terminal keyboard. As an option, answers to command prompts and any switches may be contained in a Response File. Some special command characters are provided to assist the user while entering linker commands.

2.6 INVOKING LINK

LINK may be invoked three ways. By the first method, the user enters the commands as answers to individual prompts. By the second method, the user enters all commands on the line used to invoke LINK. By the third method, the user creates a Response File that contains all the necessary commands.

Summary of Methods to invoke LINK
Method 1 - LINK Method 2 - LINK <filenames> [\switches] Method 3 - LINK @<filespec>

2.6.1 Method 1: LINK

Enter:

LINK

LINK will be loaded into memory. Then, LINK returns a series of four text prompts that appear one at a time. The user answers the prompts as commands to LINK to perform specific tasks.

At the end of each line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, LINK defaults to not performing the functions described for the switches in the chart below.

The command prompts are summarized here and described in more detail in Section 2.7, Command Prompts. Following the summary of prompts is a summary of switches, which are described in more detail in Section 2.8, Switches.

PROMPT	RESPONSES
Object Modules [.OBJ]	List .OBJ files to be linked, separated by a blank spaces or plus signs (+). If plus sign is last character entered, prompt will reappear. (No default: response required).
Run File [Object-file.EXE]:	List filename for executable object code. (default: first Object filename.EXE).
List File [Run-file.MAP]:	List filename for listing (default: RUN filename).
Libraries [ ]:	List filenames to be searched, separated by blank spaces or plus signs (+). If plus sign is last character entered, prompt will reappear. (default: no search)

SWITCH	ACTION
/DSALLOCATE	Load data at high end of Data Segment. Required for Pascal and FORTRAN programs.
/HIGH	Place Run file as high as possible in memory. Do not use with Pascal or FORTRAN programs.
/LINENUMBERS	Include line numbers in List file.
/MAP	List all global symbols with definitions.
/PAUSE	Halt linker session and wait for <Rtn>.
/STACK:<number>	Set fixed stack size in Run file.

Command Characters

LINK provides three command characters:

\+ Use the plus sign (+) to separate entries and to extend the current physical line following the Object Modules and Libraries prompts. (A blank space may be used to separate object modules.)

To enter a large number of responses (each of which may also be very long), enter a plus sign/carriage return at the end of the physical line (to extend the logical line). If the plus sign/carriage return is the last entry following these two prompts, LINK will prompt the user for more modules names. When the Object Modules or Libraries prompt appears again, continue to enter responses. When all the modules to be linked have been listed, be sure the response line ends with a module name and a carriage return and not a plus sign/carriage return.

Example:

```
Object Modules [.OBJ]: FUN TEXT TABLE
CARE+<Rtn>
Object          Modules               [.OBJ]
FOO+FLIPFLOP+JUNQUE+<Rtn>
Object Modules [.OBJ]: CORSAIR<Rtn>
```

; Use a single semicolon (;) followed immediately by a carriage return at any time after the first prompt (from Run File on) to select default responses to the remaining prompts. This feature saves time and overrides the need to enter a series of carriage returns.

NOTE

Once the semicolon has been entered, the user can no longer respond to any of the prompts for that link session. Therefore, do not use the semicolon to skip over some prompts. For this, use <Return>.

Example:

```
Object Modules [.OBJ]: FUN TEXT TABLE CARE <CR>
Run module [FUN.EXE]:  ;<CR>
```

The remaining prompts will not appear, and LINK will use the default values (including FUN.MAP for the List File.)

Ctrl+Brk Use Ctrl+Brk at any time to abort the link session. If you enter an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press Ctrl+Brk to exit LINK, then reinvoke LINK and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

2.6.2 Method 2: LINK <filename>[/switches]

Enter: LINK <object-list>,<runfile>, <listfile>,
<lib-list>[/switch...]

The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

where: **object list** is a list of object modules, separated by plus signs

runfile is the name of the file to receive the executable output

Listfile is the name of the file to receive the listing.

Lib list is a list of library modules to be searched

/switch are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <lib list>, as shown).

To select the default for a field, simply enter a second command without spaces in between (see the example below).

Example: LINK FUN+TEXT+TABLE+CARE/P/M,,FUNLIST,COBLIB.LIB

This example causes LINK to be loaded, then causes the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ to be loaded. LINK then pauses (caused by the /P switch). When the user presses any key, LINK links the object modules, produces a global symbol map (the /M switch), defaults to FUN.EXE run file, creates a list file named FUNLIST.MAP, and searches the library file COBLIB.LIB.

2.6.3 Method 3: LINK @<filespec>

Enter: LINK @<filespec>

where: **filespec** is the name of a Response File. A Response File contains answers to the LINK prompts (shown under method 1 for invoking), and may also contain any of the switches. Method 3 permits the user to conduct the LINK session without interactive (direct) user responses to the LINK prompts.

IMPORTANT

Before using method 3 to invoke LINK, the user must first create the Response File.

A Response File has text lines, one for each prompt. Responses must appear in the same order as the command prompts appear.

Use switches and Special Command Characters in the Response File the same way as they are used for responses entered on the terminal keyboard.

When the LINK session begins, each prompt will be displayed in turn with the responses from the response file. If the response file does not contain answers for all the prompts, either in the form of filenames or the semicolon special character or carriage returns, LINK will, after displaying the prompt which does not have a response, wait for the user to enter a legal response. When a legal response has been entered, LINK continues the link session.

Example:

```
FUN TEXT TABLE CARE /PAUSE/MAP FUNLIST
COBLIB.LIB
```

This Response File will cause LINK to load the four files. LINK will pause before creating and producing a public symbol map to permit the user to swap diskettes (see discussion under /PAUSE in Section 3.4, Switches, before using this feature). When the user presses any key, the output files will be named FUN.EXE and FUNLIST.MAP, LINK will search the library file COBLIB.LIB, and will use the default settings for the flags.

2.7 COMMAND PROMPTS

LINK is commanded by entering responses to four text prompts. When you have entered a response to the current prompt, the next appears. When the last prompt has been answered, LINK begins linking automatically without further command. When the link session is finished, LINK exits to the operating system. When the operating system prompt is displayed, LINK has finished successfully. If the link session is unsuccessful, LINK returns the appropriate error message.

LINK prompts the user for the names of object, run, list files, and for libraries. The prompts are listed in their order of appearance. For prompts which can default to preset responses, the default response is shown in square brackets ([]) following the prompt. The Object Modules: prompt is followed by only a filename extension default response because it has no preset filename response and requires a filename from the user.

Object Modules [.OBJ]:

Enter a list of the object modules to be linked. LINK assumes by default that the filename extension is .OBJ. If an object module has any other filename extension, the extension must be given here. Otherwise, the extension may be ommitted.

Modules must be separated by plus signs (+).

Remember that LINK loads Segments into Classes in the order encountered (see Section 2.2, Definitions). Use this information for setting the order in which the object modules are entered.

Run File [First-Object-Filename.EXE]:

The filename entered will be created to store the Run (executable) file that results from the link session. All Run files receive the filename extension .EXE, even if the user specifies an extension (the user specified extension is ignored).

If no response is entered to the Run File: prompt, LINK uses the first filename entered in response to the Object Modules: prompt as the RUN filename.

Example:

```
Run File [FUN.EXE]: B:PAYROLL/P
```

This response directs LINK to create the Run file

PAYROLL.EXE on drive B:. Also, LINK will pause, which allows the user to insert a new diskette to receive the Run file. The default response is the Run filename with the default filename extension .MAP.

Libraries [ ]:

The valid responses are one to eight library filenames or simply a carriage return. (A carriage return only means no library search.) Library files must have been created by a library utility. LINK assumes by default that the filename extension is .LIB for library files.

Library filenames must be separated by blank spaces or plus signs (+).

LINK searches the library files in the order listed to resolve external references. When it finds the module that defines the external symbol, LINK processes the module as another object module.

If LINK cannot find a library file on the diskettes in the disk drives, it returns the message:

```
Cannot find library <library-name>
Enter new drive letter:
```

Simply press the letter for the drive designation (for example B).

LINK does not search within each library file sequentially. LINK uses a method called dictionary indexed library search. This means that LINK finds definitions for external references by index access rather than searching from the beginning of the file to the end for each reference. This indexed search reduces substantially the link time for any sessions involving library searches.

2.8 SWITCHES

The six switches monitor alternate linker functions. Switches must be entered at the end of a prompt response, regardless of which method is used to invoke LINK. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is entered at the end of one response, each switch must be preceded by the slash mark (/).

All switches may be abbreviated, from a single letter through the whole switch name. The only restriction is that an abbreviation must be a sequential sub-string from the first letter through the last entered; no gaps or tranpositions are allowed. For example:

Legal	Illegal
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE

Use of the /DSALLOCATE switch directs LINK to load all data (DGroup) at the high end of the Data Segment. Otherwise, LINK loads all data at the low end of the Data Segment. At runtime, the DS pointer is set to the lowest possible address and allows the entire DS segment to be used. Use of the /DSALLOCATE switch in combination with the default load low (that is, the /HIGH switch is not used), permits the user application to allocate dynamically any available memory below the area specifically allocated within DGroup, yet to remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

NOTE

The user's application program may dynamically allocate up to 64K bytes (or the actual amount available) less the amount allocated within DGroup.

/HIGH

Use of the /HIGH switch causes LINK to place the Run image as high as possible in memory. Otherwise, LINK places the Run file as low as possible.

NOTE

Do not use the /HIGH switch with Pascal or FORTRAN programs.

/LINENUMBERS

Use of the /LINENUMBERS switch directs LINK to include in the List file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the List file.

NOTE

Not all compilers produce object modules that contain line number information. In these cases, of course, LINK cannot include line numbers.

/MAP

/MAP directs LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, LINK will list only errors (which includes undefined globals).

The symbols are listed alphabetically. For each symbol, LINK lists its value and its segment:offset location in the Run file. The symbols are listed at the end of the List file.

/PAUSE

The /PAUSE switch causes LINK to pause in the link session when the switch is encountered. Normally, LINK performs the linking session without stop from beginning to end. The pause allows the user to swap the diskettes before LINK outputs the Run (.EXE) file.

When LINK encounters the /PAUSE switch, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

LINK resumes processing when the user presses any key.

CAUTION

Do not swap the diskette which will receive the List file, or the diskette used for the VM.TMP file, if created.

/STACK:<number>

number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If the /STACK switch is not used for a link session, LINK calculates the necessary stack size automatically.

If a value from 1 to 511 is entered, LINK uses 512.

All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, LINK will return a WARNING: NO STACK STATEMENT error message.

2.9 ERROR MESSAGES

All errors cause the link session to abort. Therefore, after the cause is found and corrected, LINK must be rerun.

ATTEMPT TO ACCESS DATA OUTSIDE OF SEGMENT BOUNDS, POSSIBLY BAD OBJECT MODULE
Cause: probably a bad object file

BAD NUMERIC PARAMETER
Cause: Numeric value not in digits

CANNOT OPEN TEMPORARY FILE
Cause: LINK is unable to create the file VM.TMP because the disk directory is full.
Cure: insert a new diskette. Do not change the diskette that will receive the list.MAP file.

ERROR: DUP RECORD TOO COMPLEX
Cause: DUP record in assembly language module is too complex.
Cure: simplify DUP record in assembly language program.

ERROR: FIXUP OFFSET EXCEEDS FIELD WIDTH
Cause: an assembly language instruction refers to an address with a short instruction instead of a long instruction.
Cure: edit assembly language source and reassemble

INPUT FILE READ ERROR
Cause: probably a bad object file

INVALID OBJECT MODULE
Cause: object module(s) incorrectly formed or incomplete (as when assembly was stopped in the middle).

SYMBOL DEFINED MORE THAN ONCE
Cause: LINK found two or more modules that define a single symbol name.

PROGRAM SIZE OR NUMBER OF SEGMENTS EXCEEDS CAPACITY OF LINKER

Cause: the total size may not exceed 384K bytes and the number of segments may not exceed 255

REQUESTED STACK SIZE EXCEEDS 64K

Cure: specify a size less than or equal to 64K bytes with the /STACK switch

SEGMENT SIZE EXCEEDS 64K

64K bytes is the addressing system limit.

SYMBOL TABLE CAPACITY EXCEEDED

Cause: very many, very long names entered; exceeding approximately 25K bytes

TOO MANY EXTERNAL SYMBOLS IN ONE MODULE

The limit is 256 external symbols per module.

TOO MANY GROUPS

The limit is 10 Groups

TOO MANY LIBRARIES SPECIFIED

The limit is 8.

TOO MANY PUBLIC SYMBOLS

The limit is 1024.

TOO MANY SEGMENTS OR CLASSES

The limit is 256 (Segments and Classes taken together)

UNRESOLVED EXTERNALS: <list>

The external symbols listed have no defining module among the modules or libraries files specified.

VM READ ERROR

Cause: a disk problem; not LINK caused.

WARNING: NO STACK SEGMENT
Cause: none of the object modules specified contains a statement allocating stack space, but the user entered the /STACK switch.

WARNING; SEGMENT OF ABSOLUTE OR UNKNOWN TYPE
Cause: a bad object module or an attempt to link modules LINK cannot handle (e.g., an absolute object module).

WRITE ERROR IN TMP FILE
Cause: no more disk space remaining to expand VM.TMP file.

WRITE ERROR ON RUN FILE
Cause: usually, not enough disk space for Run file.

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 3 - DEBUG

3.1 INTRODUCTION

DEBUG is a debugging program used to provide a controlled testing environment for binary and executable object files. Note that text editors such as IN:SCRIBE are used to alter source files; DEBUG is the counterpart for binary files. DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then to immediately reexecute the program to check the validity of the changes.

All DEBUG commands may be aborted at any time by pressing Ctrl+Brk. Ctrl+NumLock suspends the display, so that the user can read it before the output scrolls away. Entering any key other than Ctrl+Brk or Ctrl+NumLock restarts the display. All of these commands are consistent with the control character functions available at the DOS command level.

3.2 INVOCATION

To invoke DEBUG, enter:

```
DEBUG [<filespec> [<arglist>]]
```

For example, if a **<filespec>** is specified, then the following is a typical invocation:

```
DEBUG FILE.EXE
```

DEBUG then loads FILE.EXE into memory starting at 100 hexadecimal in the lowest available segment. The BX:CX registers are loaded with the number of bytes placed into memory. The DEBUG prompt is a right angle bracket (>).

An **<arglist>** may be specified if **<filespec>** is present. These are filename parameters and switches that are to be passed to the program <filespec>. Thus, when **<filespec>** is loaded into memory, it is loaded as if it had been invoked with the command:

```
<filespec> <arglist>
```

Here, **<filespec>** is the file to be debugged, and the <arglist> is the rest of the command line that is used when **<filespec>** is invoked and loaded into memory.

If no <filespec> is specified, then DEBUG is invoked as follows:

```
DEBUG
```

DEBUG then returns with the prompt, signaling that it is ready to accept user commands. Since no filename has been specified, current memory, disk sectors, or disk files can be worked on by invoking later commands.

3.3 COMMANDS

Each DEBUG command consists of a single letter followed by one or more parameters. Additionally, the control character and the special editing functions described in Section 2 all apply inside DEBUG.

If a syntax error occurs in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow (^) and the word error.

For example:

```
dcs:100 cs:110
     ^ error
```

All commands and parameters may be entered in either upper or lower case. Any combination of upper and lower case may be used in commands.

The DEBUG commands are summarized in Table 3.1 and are described in detail with examples following the description of command paramters.

Table 3.1 - DEBUG Commands

DEBUG Command	Function
C<range> <address>	Compare
D[<range>]	Dump
E<address> [<list>]	Enter
F<range> <list>	Fill
G[=<address> [<address>...]	Go
H<address> <address>	Hex
I<value>	Input
L[<address> [<drive><record><record>]]	Load
M<range> <address>	Move
N<filespec>	Name
O<value> <byte>	Output
Q	Quit
R[<register-name>]	Register
S<range> <list>	Search
T[=<address>][<value>]	Trace
U[<range>]	Unassemble
W[<address> [<drive><record><record>]]	Write

3.4 PARAMETERS

As the summary above shows, all DEBUG commands accept paramters, except the Quit command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent.

```
dcs:100 110
d cs: 100 110
d,cs:100,110
```

Also, entries may be made in any combination upper or lower case.

PARAMETER	DEFINITION
<drive>	A one digit hexadecimal value to indicate which drive a file will be loaded from or written to. The valid values are 0-3. These values designate the drives as follows: 0=A:, 1=B:, 2=C:, 3=D:.
<byte>	A two digit hexadecimal value to be placed in or read from an address or register.
<record>	A 1 to 3 digit hexadecimal value used to indicate the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors. However, their numbering differs since they represent the entire disk space.
<value>	A hexadecimal value up to four digits used to specify a port number or the number of times a command should repeat its function.
<address>	A two part designation consisting of either an alphabetic segment register designation or a four digit segment address plus an offset value. The segment designation or segment address may be omitted, in which case the default segment is used. DS is the default segment for all commands except G, L, T. U, and W, for which the default segment is CS. All numeric values are hexadecimal. For example: CS:0100 04BA:0100

The colon is required between a segment designation (whether numeric or alphabetic) and an offset.

<range> Two <address>s: e.g., <address> <address>; or one <address>, an L, and a <value>: e.g., <address> L <value> where <value> is the number of lines the command should operate on; or simply <address>, and L80 is assumed. The last form can not be used if another hex value follows the <range>, since the hex value would be interpreted as the second <address> of the <range>.

Examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The following is illegal:

```
CS:100 CS:110
       ^ error
```

The limit for <range> is 10000 hex. To specify a <value> of 10000 hex within four digits, enter 0000 (or 0).

<list> A series of <byte> values or of <string> line. <list> must be the last parameters on the command line.

Example:

```
fcs:100 42 45 52 54 41
```

<string> Any number of characters enclosed in quote marks. Quote marks may be either single (') or double ("). Within <string>s, the opposite set of quote marks may be used freely as literals. If the delimiter quote marks must appear within a <string>, the quote marks must be doubled. For example, the following strings are legal:

```
'This is a "string" is okay.'
'This is a ''string'' is okay.'
```

However, this string is illegal:

```
'This is a 'string' is not.'
```

Similarly, these string are legal:
"This is a 'string' is okay."
"This is a ""string"" is okay."

However, this string is illegal:

"This is a "string" is not."

Note that the double quotations are not necessary in the following strings:

"This is a ''string'' is not necessary."
'This is a ""string"" is not necessary.'

The ASCII values of the characters in the string are used as a <list> of byte values.

NAME: Compare

SYNTAX: C<range> <address>

FUNCTION: Compare the portion of memory specified by <range> to a portion of the same size beginning at <address>.

COMMENTS If the two areas of memory are identical, there is no display and DEBUG returns with the MS-DOS prompt. If there are differences, they are displayed as:

<address1> <byte1> <byte2> <address2>

EXAMPLE: The following commands have the same effect:

```
C100,1FF 300
```

or

```
C100L100 300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

NAME: Dump

SYNTAX: D[<range>]

FUNCTION: Display the contents of the specified region of memory.

COMMENTS: If a range of addresses is specified, the contents of the range are displayed. If the D command is entered without paramters, 128 bytes are displayed at the first address (DS:100) after that displayed by the previous Dump command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows sixteen bytes with a hyphen between the eighth and ninth bytes. At times, displays are split in this manner to fit them on the page. Each displayed line, except possibly the first, begins on a 16-byte boundary.

If the user enters the command:

```
dcs:100 110
```

DEBUG displays:

```
04BA:0100 42 45 52 54 41 ... 4E 44 BERTA T. BORLAND
```

If the following command is entered:

```
D
```

the display is formatted as described above. Each line of the display begins with an address; incremented by 16 from the address on the previous line. Each subsequent D (entered without parameters) displays the bytes immediately following those last displayed.

If the user enters the command:

```
DCS:100 L 20
```

the display is formatted as described above, but 20H bytes are displayed.

If the user enters the command:

```
DCS:100 115
```

the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed.

NAME: Enter

SYNTAX: E<address>[<list>]

FUNCTION: Enter byte values into memory at the specified <address>.

COMMENTS: If the optional <list> of values is entered, the replacement of byte values occurs automatically. (If an error occurs, no byte values are changed.) If the <address> is entered without the optional <list>, DEBUG displays the address on the next line and waits for the user's input. At this point, the Enter command waits for you to peform one of the following actions:

1. Replace a byte value with a value the user types in. The user simply types the value after the current value. If the value typed in is not a legal hexadecimal value or if more than two digits are typed, the illegal or extra character is not echoed.

2. Press the space bar to advance to the next byte. To change the value, simply enter the new value as described in (1.) above. If the user spaces beyond an eight-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.

3. Type a hyphen (-) to return to the preceding byte. If the user decides to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When the hyphen is typed, a new line is started with the address and its byte value displayed.

4. Press the <Rtn> key to terminate the Enter command. The <Rtn> key may be pressed at any byte position.

EXAMPLE:

Assume the following command is entered:

```
ECS:100
```

DEBUG displays:

```
04BA:0100 EB._
```

To change this value to 41, enter "41" as shown:

```
04BA:0100 EB.41_
```

To step through the subsequent bytes, press the space bar to see:

```
04BA:0100 EB.41     10.     00.     BC._
```

To change BC to 42:

```
04BA:0100 EB.41     10.     00.     BC.42_
```

Now, realizing that 10 should be 6F; enter the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100 EB.41     10.     00.     BC.42-
04BA:0102 00.-
04BA:0101 10.6F
```

Pressing the <Rtn> key ends the Enter command and returns to the DEBUG command level.

NAME: Fill

SYNTAX: F<range> <list>

FUNCTION: Fill the addresses in the <range> with the values in the <list>.

COMMENTS: If the <range> contains more bytes than the number of values in the <list>, the <list> will be used repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range> the extra values in the <list> will be ignored. If any of the memory in the <range> is not valid <bad or nonexistent>, the error will be propagated into all succeeding locations.

EXAMPLE: Assume the following command is entered:

```
F04BA:100 L 100 42 45 52 54 41
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values are repeated until all 100H bytes are filled.

NAME: Go

SYNTAX: G[=<address>][<address>...]

FUNCTION: Execute the program currently in memory.

COMMENTS: If the Go command is entered alone, the program executes as if the program had run outside DEBUG.

If =<address> is set, execution begins at the address specified. The equal sign (=) is required, so that DEBUG can distinguish the start =<address> from the breakpoint <address>es.

With the other optional addresses set, execution stops at the first <address> encountered, regardless of that address' position in the list of addresses to halt execution, no matter which branch the program takes. When program execution reaches a breakpoint, the registers, flags, and decoded instruction are displayed for the last instruction executed. (The result is the same as if you had entered the Register command for the breakpoint address.)

Up to ten breakpoints may be set. Breakpoints may be set only at addresses containing the first byte of an opcode. If more than 10 breakpoints are set, DEBUG returns the BP Error message.

The user stack pointer must be valid and have six bytes available for this command. The G command uses an IRET instruction to cause a jump to the program under test. The user stackpoint is set, and the user Flags, Code Segment register, and Instruction Pointer are pushed on the user stack. (Thus, if the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es). When an instruction with the breakpoint code is encountered all breakpoint addresses are restored to their original instructions. If execution is not halted at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

EXAMPLE: Assume the following command is entered:

GCS:7550

The program current in memory executes up to the address 7550 in the CS segment. Then DEBUG displays registers and flags, after which the Go command is terminated.

After a breakpoint has been encountered, if you enter the Go command again, then the program executes just as if the user had entered the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

NAME: **Hex**

SYNTAX: H<value> <value>

FUNCTION: Perform hexadecimal arithmetic on the two parameters.

COMMENTS: First, DEBUG adds the two parameters, then subtracts the second parameter from the first. The result of the arithmetic is displayed on one line; first the sum, then the difference.

EXAMPLE: Assume the following command is entered:

```
H19F 10A
```

DEBUG peforms the calculations and then returns the results:

```
02A9 0095
```

NAME: **Input**

SYNTAX: I<value>

FUNCTION: Input and display one byte from the port specified by <value>.

COMMENTS: A 16-bit port address is allowed.

EXAMPLE: Assume the following command is entered:

```
I2F8
```

Assume also that the byte at the port is 42H. DEBUG inputs the byte and displays the value:

```
42
```

NAME: Load

SYNTAX: L[<address> [<drive> <record> <record>]]

FUNCTION: Load a file into memory

COMMENTS: Set BX:CX to the number of bytes read. The file must have been named with either the DEBUG invocation command or with the N command. Both the invocation and the N commands format a filename properly in the normal format of a file control block at CS:5C.

If the L command is given without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If the L command is given with an address parameter, loading begins at the memory <address> specified. If L is entered with all parameters, absolute disk sectors are loaded, not a file. The <record>s are taken from the <drive> specified (the drive designation is numeric here--0=A:, 1=B:, 2=C:, etc.); DEBUG begins loading with the first <record> specified, and continues until the number of sectors specified in the second <record> have been loaded.

EXAMPLE: Assume the following commands are entered:

```
A:DEBUG
>NFILE.COM
```

Now, to load FILE.COM, enter:

```
L
```

DEBUG loads the file and returns the DEBUG prompt. Assume you want to load only portions of a file or certain records from a disk. To do this, enter:

```
L04ba:100 2 0F 6D
```

DEBUG then loads 109 (6D hex) records beginning with logical record number 15 into memory beginning at address 04BA:0100. When the records have been loaded, DEBUG simply returns the prompt.

If the file has a .EXE extension, then it is relocated to the load address specified in the header of the .EXE file: the <address> parameter is always ignored for .EXE files. Note that the header itself is stripped off the .EXE file before it is loaded into memory. Thus, the size of a .EXE file on disk will differ from its size in memory.

If the file named by the Name command or specified on invocation is a .HEX file, then entering the L command with no parameters causes loading of the file beginning at the address specified in the .HEX file. If the L command includes the option <address>, DEBUG adds the <address> specified in the L command to the address found in the .HEX file to determine the start address for loading the file.

NAME: Move

SYNTAX: M<range> <address>

FUNCTION: Move the block of memory specified by <range> to the location beginning at the <address> specified.

COMMENTS: Overlapping moves (moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the block's lowest address and working towards the highest. The sequence for moves from lower addresses to higher addresses is to move the data beginning at the block's highest address and working towards the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data there before the move will remain; that is, the M command really copies the data from one area into another, in the sequence described, and writes over the new addresses. This is why the sequence of the move is important.

EXAMPLE: Assume you enter:

MCS:100 110 CS:500

DEBUG first moves address CS:110 to addresses CS:510, then CS:10F to CS:50F, and so on until CS:100 is moved to CS:500. You should enter the D command, using the <address> entered for the M command, to review the results of the move

NAME: Name

SYNTAX: N<filename>[<filename>...]

FUNCTION: Set filenames

COMMENTS: The Name command performs two distinct functions, both having to do with filenames. First, Name is used to assign a filename for a later Load or Write command. Thus, if you invoke DEBUG without naming any file to be debugged, then the N<filename> command must be given before a file can be Loaded. Second, Name is used to assign filename parameters to the file being debugged. In this case, Name accepts a list of parameters that are used by the file being debugged.

These functions overlap. Consider the following set of DEBUG commands:

```
>NFILE1.EXE
>L
>G
```

Because of the two pronged effect of the Name command, the following happens:

1. (N)ame assigns the filename FILE1.EXE to the filename to be used in any later Load or Write commands.

2. (N)ame also assigns the filename FILE.EXE to the first filename parameter to be used by any program that is later debugged.

3. (L)oad loads FILE.EXE into memory.

4. (G)o causes FILE.EXE to be executed with FILE.EXE as the single filename parameter (that is, FILE.EXE is executed as if FILE FILE.EXE had been typed at the command level).

A more useful chain of commands might go like this:

```
>NFILE1.EXE
>L
>NFILE2.DAT FILE3.DAT
>G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory, and then the Name command is used again, this time to specify the parameters to be used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE is executed as if FILE1 FILE2.DAT FILE3.DAT had been typed at the MS-DOS command level. Note that if a Write command were executed at this point, then FILE1.EXE -- the file being debugged -- would be saved with the name FILE2.DAT! To avoid such undesired results, you should always execute a Name command before either a Load or a Write.

There are four distinct regions of memory that can be affected by the Name command:

CS:5C FCB for file 1
CS:6C FCB for file 2
CS:80 Count of characters
CS:81 All characters entered

A File Control Block (FCB) for the first filename parameter given to the Name command is set-up at CS:5C. If a second filename parameter is given, then an FCB is setup for it beginning at CS:6C. The number of characters typed in the Name command (exclusive of the first character, "N") is given at the location CS:80. The actual stream of characters given by the Name command (again, exclusive of the letter "N") begins at CS:81. Note that this stream of characters may contains switches and delimiters that would be legal in any command typed at the MS-DOS command level.

EXAMPLE: A typical use of the Name command would be:

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the Go command executes the file in memory as if the following command line had been entered:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal runtime environment for PROG.COM.

NAME: Output

SYNTAX: O<value> <byte>

FUNCTION: Send the <byte> specified to the output port specified by <value>.

COMMENTS: A 16-bit port address is allowed.

EXAMPLE: Enter:

O2F8 4F

DEBUG outputs the byte value 4F to output port 2F8.

NAME: Quit

SYNTAX: Q

FUNCTION: Terminate the debugger

COMMENTS: The Q command takes no parameters and exits DEBUG without saving the file currently being operated on. You are returned to the MS-DOS command level.

EXAMPLE: To end the debugging session, enter:

Q<Rtn>

DEBUG is terminated, and control returns to the MS-DOS command level.

NAME: Register

SYNTAX: R[<register-name>]

FUNCTION: Display the contents of one or more CPU registers.

COMMENTS: If no <register-name> is entered, the R command dumps the register save area and displays the contents of all registers and flags.

If a register name is entered, the 16-bit value of that register is displayed in hexadecimal, and then a colon appears as a prompt. The user then either enters a <value> to change the register, or simply presses the <Return> key if no change is wanted.

The only valid <register-name>s are:

AX	BP	SS
BX	SI	CS
CX	DI	IP
DX	DS	PC
SP	ES	F

(IP and PC both refer to the instruction pointer.)

Any other entry for <register-name> results in a BR Error message.

If F is entered as the <register-name>, DEBUG displays each flag with a two character alphabetic code. To alter any flag, enter the opposite two letter code. The flags are either set or clear.

The flags with their codes for get and clear are listed below:

FLAG NAME	SET CLEAR
Overflow	OV NV
Direction	DN Decrement UP Increment
Interrupt	EI Enabled DI Disabled
Sign	NG Negative PL Plus
Zero	ZR NZ
Auxiliary Carry	AC NA

Parity	PE Even PO Odd
Carry	CY NC

Whenever the user enters the command RF, the flags are displayed in the order shown above in a row at the beginning of a line. At the end of the list of flags, DEBUG displays a hyphen (-). You may enter new flag values as alphabetic pairs. The new flag values can be entered in any order. You are not required to leave spaces between the flag entries. To exit the R command, press the <Return> key. Flags for which new values were not entered remain unchanged.

If more than one value is entered for a flag, DEBUG returns a DF Error message. If you enter a flag code other than those shown above, DEBUG returns a BF Error message. In both cases, the flags up to the error in the list are changed; flags at and after the error are not.

At start up, the segment registers are set to the bottom of free memory, the Instruction Pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

EXAMPLE: Enter:

R

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, then DEBUG might display:

```
AX=0E00 BX-00FF CS-0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A  CD21          INT     21
```

If you enter:

RF

DEBUG displays the flags:

```
NV UP DI NG NZ AC PE NC - _
```

Now enter any valid flag designation, in any order, with or without spaces.

For example:

```
NV UP DI NG NZ AC PE NC - PLEICY&lt;Rtn&gt;
```

DEBUG responds only with the DEBUG prompt. To see the changes, enter either the R or RF command:

```
RF
NV UP EI PL NZ AC PE CY - _
```

Press <Rtn> to leave the flags this way, or to enter different flag values.

NAME: Search

SYNTAX: S<range> <list>

FUNCTION: Search the range specified for the <list> of bytes specified.

COMMENTS: The <list> may contain one or more bytes, each separated by a space or comma. If the <list> contains more than one byte, only the first address of the byte string is returned. If the <list> contains only one byte, all addresses of the byte in the <range> are displayed.

EXAMPLE: If you enter:

```
SCS:100 110 41
```

DEBUG might return the response:

```
04BA:0104
04BA:010D
>_
```

NAME: Trace

SYNTAX: T[=<address>][<value>]

FUNCTION: Execute one instruction and display the contents of all registers, flags, and the decoded instruction.

COMMENTS: If the original =<address> is entered, tracing occurs at the =<address> specified. The optional <value> causes DEBUG to execute and trace the number of steps specified by <value>.

The T command uses the hardware trace mode of the microprocessor. Consequently, the user may also trace instructions stored in ROM.

EXAMPLE: Enter:

```
T
```

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. Assume that the current position is 04BA:011A; then DEBUG might return the display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A CD21          INT     21
```

Now enter:

```
T=011A 10
```

DEBUG executes sixteen (10 hex) instructions beginning at 011A in the current segment and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that <Ctrl+NumLock> suspends the display at any point, so that you can study the registers and flags for any instruction.

NAME: Unassemble

SYNTAX: U[<range>]

FUNCTION: Disassemble bytes and display the source statements that correspond to them, along with addresses and byte values.

COMMENTS: The display of disassembled code looks like a listing for an assembled file. If you enter the U command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous Unassemble command. If you enter the U command with the <range> parameters, then DEBUG disassembles all bytes in the range. If the <range> is given as an <address> only, then 20H bytes are disassembled, not 80H.

EXAMPLE: Enter:

```
U04BA:100 L10
```

DEBUG disassembles 16 bytes beginning at address 04BA:0100:

```
04BA:0100  206472   AND   [SI+72],AH
04BA:0103  69       DB    69
04BA:0104  7665     JBE   016B
04BA:0106  207370   AND   [BP+DI+70],DH
04BA:0109  65       DB    65
04BA:010A  63       DB    63
04BA:010B  69       DB    69
04BA:010C  66       DB    66
04BA:010D  69       DB    69
04BA:010E  63       DB    63
04BA:010F  61       DB    61
```

If you enter:

```
u04ba:0100 0108
```

the display shows:

```
04BA:0100  206472   AND   [SI+72],AH
04BA:0103  69       DB    69
04BA:0104  7665     JBE   016B
04BA:0106  207370   AND   [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. The U command can be entered for the changed locations, the new instructions viewed, and the disassembled code used to edit the source file.

NAME: Write

SYNTAX: W[<address> [<drive> <record> <record>]]

FUNCTION: Write the file being debugged to a disk file.

COMMENTS: If only the W appears, BX:CX must already be set to the number of bytes to be written; the file is written beginning from CS:100. If the W command is given with just an address, then the file is written beginning at that address. If a G or T command was used, BX:CX must be reset before using the Write command without parameters. (Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file as long as the length has not changed.)

The file must have been named either with the DEBUG invocation command or with the N command (see Name above). Both the invocation and the N commands format a file name properly in the normal format of a file control block at CS:5C.

If the W command is given with parameters, the write begins from the memory address specified; (the drive designation is numeric here -- 0=A:, 1=B:, 2=C:, etc.); DEBUG writes the file beginning at the logical record number specified by the first <record>; and continues until the number of sectors specified in the second <record> have been written.

WARNING

Writing to absolute sectors is EXTREMELY dangerous because the process bypasses the file handler.

EXAMPLE: Enter:

```
W
```

DEBUG writes out the file to disk then displays the DEBUG prompt:

```
W
>_
```

Another example:

```
WCS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100 to the disk in drive B:. The data written out starts in disk logical record number 37H and consists of 2BH records. When the write is complete, DEBUG displays the prompt:

```
WCS:100 1 378 2B
>_
```

3.5 ERROR MESSAGES

During the DEBUG session, you may receive any of the following error messages. Each error terminates the DEBUG command with which it is associated, but does not terminate DEBUG itself.

ERROR CODE	DEFINITION
BF	Bad Flag The user attempted to alter a flag, but the characters entered were not one of the acceptable pairs of flag values. See the Register command for the list of acceptable flag entries.
BP	Too many Breakpoints The user specified more than ten breakpoints as parameters to the G command. Reenter the Go command with ten or fewer breakpoints.
BR	Bad Register The user entered the R command with an invalid register name. See the Register command for the list of valid register names.
DF	Double Flag The user entered two values for one flag. The user may specify a flag value only once per RF command.

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 4 - EDLIN

EDLIN

EDLIN is a text editor used to edit files that are divided into lines. Each line may be up to 255 characters, the last character of each being the end of line character, the carriage return. Line numbers are not actually present in saved text, but when a file is displayed, lines are numbered dynamically. When a file is created or edited, line numbers begin at 1 and are incremented by one through the end of the file. When new lines are inserted between existing lines, all line numbers following the inserted text are automatically incremented by the number of lines inserted. When lines are deleted between existing lines, all line numbers following the deleted text are decremented automatically by the number of lines deleted. Consequently, line numbers always run from 1 through n (the last number).

4.1 INVOCATION

To invoke EDLIN, enter:

```
EDLIN <filespec>
```

If the file specified exists, EDLIN loads the file into memory. If the whole file is loaded, EDLIN returns the message "End of input file" and an asterisk (*) prompt. If the file is "larger than memory", then EDLIN fills 3/4 of available memory with the first part of the file and then returns the asterisk (*) prompt, but not the "End of input file" message. (This is just like the Append command with no parameter. See Section 4.2, "Interline Commands," for more information on Append.)

You may then edit the existing file. When you want to edit the part of a file that is not in memory, you must first write out to disk some of the file that is in memory, and then append lines into memory. (See the A and W commands.)

If the file specified does not exist, EDLIN creates the file and returns the message NEW FILE. and then displays the asterisk (*) prompt, indicating that the editing session may begin.

IMPORTANT

When creating a new file, be sure to specify on which drive the file should be saved. The command to end the editing session and save the file does not allow parameters. Therefore, if the drive is not designated during EDLIN invocation, the file is saved on the default drive.

EDLIN commands belong to two types: intraline and interline. Intraline commands perform editing functions within a single line. The commands used to perform intraline editing are the control character functions and the special editing commands used in DOS. The special editing functions are described in more detail in the following section. Note, however, that these are the same commands that are used at the DOS command level. The only difference between them is that the EDLIN commands operate on the line currently being edited, rather than the MS-DOS command line.

4.2 INTRALINE COMMANDS

Intraline commands include the special editing functions and the control character functions: only the special editing functions are discussed here.

Table 4.1 summarizes the commands, codes, and functions. Descriptions of the special editing functions follow the table.

Table 4.1 Special Editing Commands

Command	Code	Function
Copy one character	<F1> or ->	Copy one character from the template to the new line
Copy up to character	<F2>	Copy all characters from the template to the new line up to the character specified
Copy Template	<F3>	Copy all remaining characters in the template to the new line
Skip one character	<Del>	Do not copy (skip over) a character in the template
Skip up to character	<F4>	Do not copy (skip over) the characters in the template up to the character specified
Quit Input	<Esc>	Void the current input; leaves the template unchanged
Insert mode	<Ins>	Allows you to insert characters within a line; pressing <Ins> again exits insert mode
New Template	<F5>	Make the new line the new template

Key: <F1>

Function: Copy one character from the template to the input buffer.

Comments: Pressing the <F1> key copies one character from the template to the input buffer. When the <F1> key is pressed, one character is inserted in the buffer and insert mode is automatically turned off if it was on. Use the <F1> key to advance the cursor one column across the line.

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <F1> key copies the first character (T) to the second of the two lines displayed:

```
     1:*This is a sample file
<F1> 1:*T_
```

Each time the <F1> key is pressed, one more character appears:

```
<F1> 1:*Th_
<F1> 1:*Thi_
<F1> 1:*This_
```

Key: <F2>

Function: Copy multiple characters up to a given character

Comments: Pressing the <F2> key copies all characters up to a given character from the template to the input buffer. The given character is the next character typed and is not copied or shown on the screen. Pressing the <F2> key causes the cursor to move to the single character that is this command's only parameter. If the template does not contain the specified character, nothing is copied. Pressing <F2> also automatically turns off insert mode if it is on.

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <F2> key copies all characters up to the character pressed immediately after the <F2> key.

```
         1:*This is a sample file.
<F2>p    1:*This is a sam_
```

Key: <F3>

Function: Copy template to input buffer

Comments: Pressing <F3> copies all remaining characters from the template to the input buffer. Regardless of the cursor position at the time the <F3> key is pressed, the rest of the line appears, and the cursor is positioned after the last character on the line.

Example: Assume the screen shows:

```
     1:*This is a sample file.
     1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <F3> key copies all characters from the template (shown in the upper line displayed) to the line with the cursor (the lower line displayed):

```
     1:*This is a sample file.
<F3> 1:*This is a sample file._
```

Also, insert mode is automatically turned off if it was on.

Key: <Del>

Function: Skip over one character in the template

Comments: Pressing the <Del> key skips over one character in the template. Each time you press the <Del> key, one character is deleted (not copied) from the template. The action of the <Del> key is similar to the <F1> key, except that <Del> skips a character in the template rather than copies it to the input buffer.

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <Del> key skips over the first character ("T").

```
      1:*This is a sample file.
<Del> 1:*_
```

The cursor position does not move, only the template is affected. To see how much of the line has been (skipped over), press the <F3> key, which moves the cursor beyond the last character of the line.

```
1:*This is a sample file.
1:*
1:*his is a sample file._
```

Key: <F4>

Function: Skip multiple characters in the template

Comments: Pressing the <F4> key skips over all characters up to a given character in the template. The given character is the next character typed, and is not copied and not shown on the screen. If the template does not contain the specified character, nothing is skipped over. The action of the <F4> key is similar to the <F2> key, except that <F4> skips over characters in the template rather than copies them to the input buffer.

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Pressing the <F4> key skips over (deletes) all the characters in the template up to the character pressed after the <F4> key:

```
      1:*This is a sample file.
<F4>p 1:*_
```

The cursor position does not move. To see how much of the line has been skipped over, press the <F3> key to copy the template. this moves the cursor beyond the last character of the line:

```
      1:*This is a sample file.
<F4>p 1:*
<F2>  1:*ple file._
      1:*_
```

Key: <Esc>

Function: Quit input and flush the input buffer

Comments: Pressing the <Esc> key flushes the input buffer, but it leaves the template unchanged. <Esc> also prints a back slash ( ), carriage return, and line feed, and turns insert mode off if it was on. The cursor is positioned at the beginning of the line. Pressing the <F3> key copies the template to the input buffer just as the line was before <Esc> was pressed.

Example: Assume the screen shows:

```
        1:*This is a sample file.
        1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume you want to replace the line by typing:

```
                1:*This is a sample file.
Sample File     1:*Sample File_
```

Now, to reedit the line, press <Esc>:

```
        1:*This is a sample file.
<Esc>   1:*Sample File
        1:_
```

<Rtn> can now be pressed to keep the original line or to perform any other intraline editing functions. If <F3> is pressed, the original template is copied to the input buffer:

```
<F3> 1:*This is a sample file._
```

Key: **<Ins>**

Function: Enter insert mode or move from insert to replace mode

Comments: The <Ins> key is a toggle switch which moves from replace mode (the default) to insert mode and back to replace mode when the <Ins> key is pressed a second time.

On entry into insert mode the current position in the template is not changed. The cursor does move as each character is inserted. However, when you have finished inserting characters, the cursor is positioned at the same character as it was before the insertion began. Thus, characters are inserted before the character the cursor points to.

Pressing the <Ins> key again causes exit from insert mode and entry into replace mode. All characters entered now overstrike and replace characters in the template. When you start to edit a line, this mode is in effect. Each character typed replaces a character in the template. If the <Rtn> key is pressed, the remainder of the template is truncated.

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume you press the <F2> and "p" keys:

```
            1:*This is a sample file.
<F2>p       1:*This is a sam_
```

Now press the <Ins> key and insert the three characters "s", "o", and "n":

```
               1:*This is a sample file.
<F2>p          1:*This is a sam_
<Ins>son       1:*This is a samson_
```

If you now press the <F3> key, the rest of the template is copied to the line:

```
               1:*This is a samson
<F3>           1:*This is a samsonple file._
```

If you were to press the <Rtn> key, instead, the remainder of the template would be truncated, and the input buffer ended at the end of the insert:

```
<Ins>son<Rtn> 1:*This is a samson
```

Assume you then press the <ins> key and "ite" (thus entering replace mode), and then <F3>:

```
               1:*This is a sample file.
<F2>p          1:*This is a sam_
<Ins>son       1:*This is a samson
<Ins>ite       1:*This is a samsonite_
<F3>           1:*This is a samsonite file._
```

If you type in characters that extend beyond the length of the template, the remaining characters in the template are automatically appended when you type <F3>.

Key: <F5>

Function: Create new template

Comments: Pressing the <F5> key copies the current contents of the input buffer to the template. The contents of the old template are then destroyed. Pressing <F5> outputs an at sign character (@), a carriage return, and a line feed. The input buffer is also emptied and insert mode is turned off.

NOTE

<F5> performs the same functions as the <Esc> key, except that the template is changed and an at sign character (@) is printed instead of a backslash (\).

Example: Assume the screen shows:

```
1:*This is a sample file.
1:*_
```

At the beginning of the intraline edit, the cursor is positioned at the beginning of the line (indicated by the underline). Assume that you enter <F2>p, <Ins>son, <Ins>ite, and then <F3>:

```
             1:*This is a sample file.
<F2>p        1:*This is a sam_
<Ins>son     1:*This is a samson_
<Ins>ite     1:*This is a samsonite_
<F3>         1:*This is a samsonite file._
```

At this point, assume that you want this line as the new template, so you press the <F5> key:

```
<F5> 1:*This is a samsonite file.@
     _
```

Additional editing can now be done using the above new template.

4.3 INTERLINE COMMANDS

Interline commands perform editing functions on whole lines at a time. The interline commands are summarized in the following list and are described in detail with examples following the description of command parameters.

Table 4.2 Interline Commands

Command	Purpose
<line>	Edit Line
A	Append Lines
D	Delete Lines
E	End Editing
I	Insert Text
L	List Text
Q	Quit Editing
R	Replace Text
S	Search Text
W	Write Lines

4.3.1 Parameters

Each interline command accepts some optional parameters. The following list of parameters indicates their form. The effect of a parameter depends on the command it is used with.

PARAMETER	DEFINITION
<line>	<line> indicates a line number to be entered by the user. Line numbers must be separated from other line numbers, other parameters, and the command. Use a comma or space to separate.

<line> may be specified one of four ways:

Number	any integer less than 65534. If a number larger than the largest existing line number is specified, then <line> indicates the line after the last line number.
Period	(.) If a period is specified for <line>, then <line> indicates the current line number. The current line is the last line edited, and not necessarily the last line displayed. The current line is marked on your screen by an asterisk (*) between the line number and the first character.
Octothorpe	(#) The pound sign indicates the line after the last line number. Specifying # for <line> has the same effect as specifying a number larger than the last line number.
<Rtn>	A carriage return entered without any of the <line> specifiers listed above directs EDLIN to use a default value appropriate to the command.

?	The question mark parameter directs EDLIN to ask the user if the correct string has been found. The question mark is used only with the Replace and Search commands. Before continuing, EDLIN waits for either a "Y" or <Rtn> for a yes response, or for any other key for a no response.
<string>	<string> represents text to be found, to be replaced, or to replace other text. The <string> parameter is used only with the Search and Replace commands. Each <string> must be terminated by a <Ctrl+Z> or a <Rtn> (see the Replace command for details). No spaces between a string and its command letter, unless you want spaces as part of a string.

Name: Edit

Syntax: [<line>]

Function: Edit line

Comments: When a line number is entered, EDLIN displays the line number and text, then, on the line below, reprints the line number. The line is then ready for editing. You may use any of the available intraline commands to edit the line. The existing text of the line serves as the template until the <Rtn> key is pressed.

If no line number is entered (that is, only the <Rtn> key is pressed), the line after the current line, marked with an asterisk (*), is edited. If no changes of the current line are needed and the cursor position is at the beginning or end of the line, press the <Rtn> key to accept the line as is.

WARNING

If the <Rtn> key is pressed while the cursor is in the middle of the line, the remainder of the line is truncated.

Example: Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: used to demonstrate
3: the editing of line
4:* four.
```

To edit line 4, enter:

```
4
```

The contents of the line are displayed along with a cursor below the line:

```
4:* four.
4:*_
```

Now type:

```
<Ins>number  4: number_
<F3><Rtn>    4: number four.
             5:*_
```

Name: **Append**

Syntax: [<n>]A

Function: Append lines from input file to editing buffer

Comments: Use this command for extremely large files that will not fit into memory all at one time. By writing out part of the editing buffer to the output file with the Write command, room is made for lines to be appended with the Append command. If A is typed without a parameter, lines are appended to the part of the file currently in memory until available memory is 3/4 full or until there are no more lines to append.

Use the W command to write out lines to the output file. If the parameter <n> is given, then <n> lines are appended to that part of the file that currently is in memory. If <n> is not given, then as much of the input file as possible is read into the editing buffer until the editing buffer is three quarters full.

Name: **Delete**

Syntax: [<line>][,<line>] D

Function: Delete the specified lines and all lines in between.

Comments: If the first <line> is omitted, the first <line> defaults to the current line (the line with the asterisk next to the line number). If the second <line> is omitted, then just the first <line> is deleted. When lines have been deleted, the line immediately after the deleted section becomes the current line and has the same line number as the first <line> had before the deletion occurred.

Example: Assume the following file exists and is ready to edit:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: Delete and Insert
 .
 .
 .
25: (The D and I commands)
26: (Use <Ctrl+C> to exit insert mode)
27:*Line numbers
```

To delete multiple lines, enter <line>,<line> D:

```
5,24 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5:*(The D and I commands)
6: (Use <Ctrl+C> to exit insert mode)
7:*Line numbers
```

To delete a single line, enter:

```
6 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
6:*Line numbers
```

Next, delete range of lines from the following file:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3:*See what happens when you
4: Delete and Insert
5:*(The D and I commands)
6: (Use <Ctrl+C> to exit insert mode)
7: Line numbers
```

To delete beginning with the current line, enter:

```
,6 D
```

The result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3:*Line numbers
```

Name: **End**

Syntax: E

Function: End the editing session

Comments: Save the edited file on disk, rename the original input file "filename.BAK", and then exit EDLIN to the DOS command level. If the file was created during the editing session, no .BAK file is created.

The E command takes no parameters. Therefore, you cannot tell EDLIN on which drive to save the file. The drive must be selected when the editing session is invoked. If the drive is not designated when EDLIN is invoked, the file is saved on the disk in the default drive. (It will still be possible to COPY the file to a different drive. However, this is done automatically if the drive is designated during invocation.)

You must be sure that the disk contains enough free space for the entire file to be written. If the disk does not contain enough free space, the write is aborted and the edited file is lost, although part of the file may be written out.

Example: The only possible command is:

```
E<Rtn>
```

After execution of the E command, control is returned to COMMAND.COM and the DOS prompt is displayed.

Name: **Insert**

Syntax: [<line>] I

Function: Insert line(s) of text immediately before the specified <line>

Comments: If you are creating a new file, the I command must be given before text can be inserted. In this case, the insert begins with line number 1.

EDLIN remains in insert mode until a <Ctrl+Z> or a <Ctrl+C> is entered. Successive line numbers appear automatically each time <Rtn> is pressed. When the insert is finished and insert mode has been exited, the <line>, which now immediately follows the inserted lines, becomes the current line. All line numbers following the inserted section are incremented by the number of lines inserted.

If <line> is not specified, the default is the current line number (the lines are inserted immediately before the current line). If <line> is an integer larger than the last line number, or if # is specified as <line>, the inserted lines are appended to the end of the file. In this case, the last line inserted becomes the current line. (This is the same as when the file is being created.)

Example: Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
6: (Use the <Ctrl+C> to exit insert mode)
7:*Line numbers
```

To insert text before a specific line (not the current line), enter <line> I:

```
4 I The result is 4:_
```

Now, enter the new text for lines 4 and 5:

```
4: fool around with
5: those very useful commands that
```

Then to end the insertion, type:

```
6: <Ctrl+Z>
```

Now type L to list the file; the result is:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: fool around with
5: those very useful commands that
6:*Delete and Insert
7: (The D and I commands)
8: (Use the <Ctrl+C> to exit insert mode)
9: Line numbers
```

To insert lines immediately before the current line, enter:

```
I
```

The result is:

```
6: _
```

Now, insert the following text terminated with a <Ctrl+Z>:

Now to List the file and see the result, type:

```
L
```

The result is:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7:*Delete and Insert
 8: (The D and I commands)
 9: (Use the <Ctrl+C> to exit insert mode)
10: Line numbers
```

To append new lines to the end of the file, enter:

```
11 I
```

This produces the following:

```
11: _
```

Now, enter the following new lines:

```
11: The insert command can place new lines
12: anywhere in the file; there's no space
problems.
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533.
```

End insertion by typing <Ctrl+C>. The new lines will appear at the end of all previous lines in the file. Now enter the list command:

```
L
```

The result is:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7:*Delete and Insert
 8: (The D and I commands)
 9: (Use the <Ctrl+C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space
    problems.
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533.
```

Name: List

Syntax: [<line>][,<line>] L

Function: List the specified range of lines, including the two lines specified.

Comments: If the first <line> is omitted, the first <line> defaults to the current line. If the second <line> is omitted, 23 lines are listed; the eleven lines before <line>, the <line>, and the eleven lines after <line>. The current line remains unchanged. If the current line is one of the lines listed, it contains an asterisk between the line number and the first character.

Example: Assume the following file exists and is ready to edit:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: Delete and Insert
 5: (The D and I commands)
 .
 .
 .
15:*The current line contains an asterisk.
 .
 .
 .
26: (Use the <Ctrl+C> to exit insert mode)
27: Line numbers
```

To list a range of lines without reference to the current line, enter <line>,<line> L:

```
2,5 L
```

The result is:

```
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: (The D and I commands)
```

To list a range of lines beginning with the current line, enter ,<line> L:

    ,26 L

The result is:

```
15:*The current line contains an asterisk.
     .
     .
     .
26: (Use <Ctrl+C> to exit insert mode)
```

To list a range of 23 lines around a specified line, enter <line>, L:

    13, L

The result is:

```
     13: The specified line is listed first in the
         range.
     14: The current line remains unchanged by the L
         command.
     15:*The current line contains an asterisk.
      .
      .
      .
     35: <Ctrl+C> exits interline insert command
mode.
```

To list a range of 23 line centered around the current line, enter only L:

    L

The result is:

```
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: Delete and Insert
 5: (The D and I commands)
 .
 .
 .
13: The specified line is listed first in the
    range.
14: The current line remains unchanged by the L
    command.
15:*The current line contains an asterisk.
 .
 .
 .
24: <Ctrl+C> exits interline insert command
    mode.
```

Name: **Quit**

Syntax: Q

Function: Quit the editing session, do not save any editing changes, and exit to the DOS operating system.

Comments: No .BAK file is created. The Q command takes no parameters. It is simply a fast means of exiting an editing session. As soon as the Q command is given, EDLIN displays the message:

```
Abort edit (Y/N)?_
```

Press "Y" to quit the editing session; press "N" (or any other key except <Ctrl+C>) if you decide to continue the editing session.

Example: Assume the following file exists and is ready to edit:

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: Compare the before and after
4: See what happens when you
5: Delete and Insert
6: Line numbers
```

Now, to delete line 3, enter:

```
3 D
```

To list the file, enter "L":

```
1: This is a sample file.
2: Use: to demonstrate dynamic line numbers
3: See what happens when you
4: Delete and Insert
5: Line numbers
```

Now, to keep the changes and to quit the editing session, enter:

```
q
```

The result is:

```
Abort edit (Y/N)?_
```

Enter "Y" to exit to the operating system command level:

```
Abort edit (Y/N)?Y
A:_
```

Name: **Replace**

Syntax: [<line>][,<line>] [?] R<string1><Ctrl+Z><string2>

Function: Replace all occurrences of <string1> in the specified range with <string2>.

Comments: As each occurrence of <string1> is found, it is replaced by <string2>. Each line in which a replacement occurs is displayed. If a line contains two or more replacements of <string1> with <string2>, then the line is displayed once for each occurrence. When all occurrences of <string1> in the specified range are replaced by <string2>, the R command terminates and the asterisk prompt reappears.

If a second string is to be given as a replacement, then <string1> must be terminated with a <Ctrl+Z>. If the <string2> is to be omitted, the <string1> may be terminated with either a combination <Ctrl+Z><Rtn>, or simply a <Rtn>. <string2> must also be terminated with a <Ctrl+Z><Rtn> combination or with a simple <Rtn>. If <string1> is omitted, then the replacement is terminated immediately. If <string2> is omitted, then <string1> is deleted from all lines in the range. If the first <line> is omitted in the range argument (as in ,<line>) then the first <line> defaults to the line after the current line. If the second <line> is omitted (as in <line> or <line>,) the second <line> defaults to #. Therefore, this is the same as <line>,#. Remember that # indicates the line after the last line of the file.

If the question mark (?) parameter is given, the Replace command stops at each line with a string that matches <string>, displays the line with <string2> in place, and then displays the prompt "O.K.?". If the user presses "Y" or the <Rtn> key, then <string2> replaces <string1>, and the next occurrence of <string1> is found. Again, the "O.K.?" prompt is displayed. This process continues until the end of the range or until the end of the file. After the last occurrence of <string1> is found, EDLIN returns the asterisk prompt.

If you press any key besides "Y" or <Rtn> after the "O.K.?" prompt, the <string1> is left as it was in the line, and the replace goes to the next occurrence of <string1>. If <string1> occurs more than once in a line, each occurrence of <string1> is replaced individually, and the "O.K.?" prompt is displayed after each replacement. In this way, only the desired <string1> is replaced, and you prevent replacement of embedded strings.

Example: Assume the following file exists and is ready to edit:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7: Delete and Insert
 8: (The D and I commands)
 9: (Use <Ctrl+C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space
problems
13: because the line numbers are dynamic;
14: They'll slide all the way to 65533
```

To replace all occurrences of <string1> with <string2> in a specific range, enter:

```
2,12 Rand<Ctrl+Z>or<Rtn>
```

The result is:

```
 5: those very useful commors that
 7: Delete or Insert
 8: (The D or I commands)
 8: (The D or I commors)
11: The insert commor can place new lines
```

Note that in the above replacement, some unwanted substitutions have occurred. To avoid these, and confirm each replacement, the same original file can be used:

```
 .
 5: those very useful commands that
 .
 7: Delete and Insert
 8: (The D and I commands)
 .
11: The insert command can place new lines
 .
 .
```

only with a slightly different command. this time, to replace only certain occurrences of the first <string> with the second <string>, enter:

```
2? Rand<Ctrl+Z>or<Rtn>
```

The result is:

```
 5: those very useful commands that
 O.K.? N
 7: Delete or Insert
 O.K.? Y
 8: (The D or I commands)
 O.K.? Y
 8: (The D or I commors)
 O.K.? N
11: The insert commor can place new lines
 O.K.? N
 *_
```

Now, enter the List command (L) to see the result of all these changes:

```
 .
 .
 5: those very useful commands that
 .
 7: Delete or Insert
 8: (The D or I commands)
 .
11: The insert command can place new lines
 .
 .
```

Name: Search

Syntax: [<line>][,<line>] [?] S<string>

Function: Search the specified range of lines for the specified string.

Comments: The <string> must be terminated with a <Rtn>. The first line that matches <string> is displayed and becomes the current line. The Search command terminates when a match is found. If no line contains a match for <string>, the message "Not found" is displayed.

If the optional parameter, question mark (?), is included in the command, EDLIN displays the first line with a matching string; it then prompts the user with the message "O.K.?". If the user presses either the "Y" or <Rtn> key, the line becomes the current line and the search teminates. If the user presses any other key, the search continues until another match is found, or until all lines have been searched (then the "Not found" message is displayed).

If the first <line> is omitted (as in ,<line> S<string>), the first <line> defaults to the line after the current line. If the second <line> is omitted (as in <line> S<string> or <line>,S<string>), the second <line> defaults to #, which is the same as <line>,# S<string>. If <string> is omitted, no search is made and the command terminates immediately.

Example: Assume the following file exists and is ready to edit:

```
 1: This is a sample file.
 2: Use: to demonstrate dynamic line numbers
 3: See what happens when you
 4: fool around with
 5: those very useful commands that
 6: perform the two major editing functions,
 7: Delete and Insert
 8: (The D and I commands)
 9: (Use <Ctrl+C> to exit insert mode)
10: Line numbers
11: The insert command can place new lines
12: anywhere in the file; there's no space
    problems
```

```
13: because the line numbers are dynamic
14:*They'll slide all the way to 65533
```

To search for the first occurrence of a string, enter:

```
2,12 Sand<Rtn>
```

The result is:

```
5: those very useful commands that
```

To get the "and" in line 7, modify the search command by entering:

```
<Del><F3>,12 Sand<Rtn>
```

The search then continues from the line after the current line (line 5), since no first line is given. The result is:

```
7: Delete and Insert
```

To Search through several occurrences of a string until the correct string is found, enter:

```
1, ? Sand
```

The result is:

```
5: those very useful commands that
O.K.?_
```

Continue:

```
7: Delete and Insert
O.K.?_
```

Now press "Y" to terminate the search:

```
O.K.?Y
*_
```

Name: **Write**

Syntax: [<n>]W

Function: Write lines from the editing buffer to the output file

Comments: The Write command is used when editing files that are larger than available memory. By executing the Write, lines are written out to the output file and room is made in the input buffer for more lines to be appended from the input file. If W is typed with no <n> parameter, then lines are written until memory is 1/4 full.

If the <n> parameter is given, then <n> lines are written out. Note that lines are written out beginning with the start of the file; subsequent lines in the editing buffer are renumbered beginning with one. A later Append command will append lines to any remaining lines in the editing buffer.

4.4 ERROR MESSAGES

EDLIN error messages occur either when you try to invoke EDLIN or during the actual editing session.

4.4.1 Errors When Invoking EDLIN

Cannot edit .BAK file--rename file

Cause: The user attempted to edit a file with the filename extension .BAK. .BAK files cannot be edited because the extension is reserved for backup copies.

Cure: If the user needs the .BAK file for editing purposes, the user must either RENAME the file with a different extension or COPY the .BAK file but with a different filename extension.

No room in directory for file

Cause: When the user attempted to create a new file, either the file directory was full or the user specified an illegal disk drive or an illegal filename.

Cure: Check the EDLIN invocation command line for illegal filename and illegal filename and illegal disk drive entries. If the command is no longer on the screen and if the user has not yet entered a new command, the EDLIN invocation command can be recovered by pressing the <F3> key.

If the invocation command line contains no illegal entries, run the CHKDSK program for the specified disk drive. If the status report shows the disk directory full, remove the disk and insert and format a new disk. If the CHKDSK status report shows the disk directory is not full, check the EDLIN invocation command for an illegal filename or illegal disk drive designation.

4.5 Errors while Editing

Entry Error

Cause: The last command entered contained a syntax error.

Cure: Reenter the command with the correct syntax.

Line too long

Cause: During Replace command mode, the string given as the replacement causes the line to expand beyond the limit of 254 characters. EDLIN aborts the Replace command.

Cure: Divide the long line into two lines, then retry the Replace command.

Disk Full--file write not completed

Cause: The user gave the End command, but the disk did not contain enough free space for the whole file. EDLIN aborts the E command and returns the user to the operating system. Some of the file may have been written to the disk.

Cure: Only a portion (at most) of the file will have been saved. The user should probably delete whatever file was saved and restart the editing session. None of the file not written out will be available after this error. Always be sure that the disk has sufficient free space for the file to be written, before you begin your editing session.

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 5 - CREF

5.1.1 Features and Benefits

The CREF Cross Reference Facility can aid you in debugging your assembly language programs. CREF produces an alphabetical listing of all the symbols in a special file produced by your assembler. With this listing, you can quickly locate all occurrences of any symbol in your source program by line number.

The CREF produced listing is meant to be used with the symbol table produced by your assembler.

The symbol table listing shows the value of each symbol, and its type and length, and its value. This information is needed to correct erroneous symbol definitions or uses.

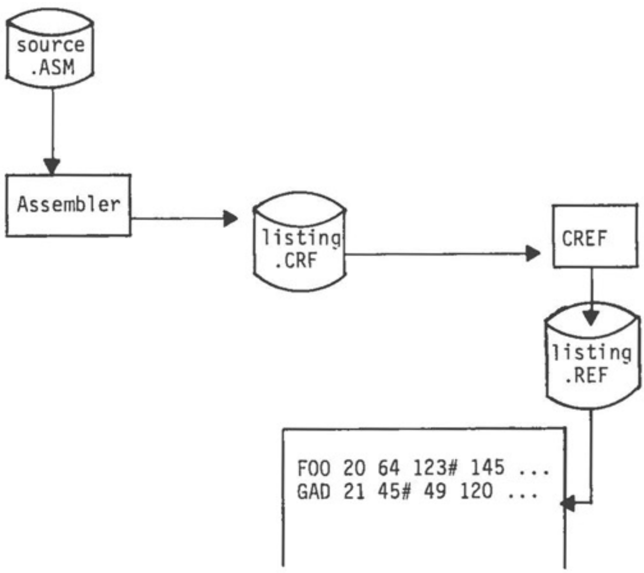
The cross reference listing produced by CREF provides you with the locations, speeding your search and allowing faster debugging.

5.1.2 Overview of CREF Operation

CREF produces a file with cross references for symbolic names in your program.

First, you must create a cross reference file with the assembler. Then, CREF takes this cross reference file, which has the filename extension .CRF, and turns it into an alphabetical listing of the symbols in the file. The cross reference listing file is given the default filename extension .REF.

Beside each symbol in the listing, CREF lists the line numbers in the source program where the symbol occurs in ascending sequence. The line number where the symbol is defined is indicated by a octothorpe sign (#).


source
.ASM
Assembler
listing
.CRF
CREF
listing
.REF
FOO 20 64 123# 145 ...
GAD 21 45# 49 120 ...

5.2 RUNNING CREF

Running CREF requires two types of commands: a command to invoke CREF and answers to command prompts. You will enter all the commands to CREF on the keyboard. Some special command characters exist to assist you while entering CREF commands.

Before you can use CREF to create the cross reference listing, you must first have created a cross reference file using your assembler. This step is reviewed in Section 5.2.3.

5.2.1 Creating a Cross Reference File

A cross reference file is created during an assembly session.

To create a cross reference file, answer the fourth assembler command prompt with the name of the file you want to receive the cross reference file.

The fourth assembler prompt is:

```
Cross reference [NUL.CRF]:
```

If you do not enter a filename in response to this prompt, or if you in any other way use the default response to the prompt, the assembler will not create a cross reference file. Therefore, you must enter a filename. You may also specify which drive or device you want to receive the file and what filename extension you want the file to have, if different from .CRF. If you change the filename extension from .CRF to anything else, you must remember to specify the filename extension when naming the file in response to the first CREF prompt (see Section 5.2.3).

When you have given a filename in response to the fourth assembler prompt, the cross reference file will be generated during the assembly session.

You are now ready to convert the cross reference file produced by the assembler into a cross reference listing using CREF.

5.2.2 Invoking CREF

CREF may be invoked two ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke CREF.

Summary of Methods to invoke CREF

Method 1	CREF
Method 2	CREF <crffile>,<listing>

5.2.3 Method 1: CREF

Enter:

CREF

CREF will be loaded into memory. Then, CREF returns a series of two text prompts that appear one at a time. You answer the prompts to command CREF to convert a cross reference file into a cross reference listing.

Command Prompts

Cross reference [.CRF]:

Enter the name of the cross reference file you want CREF to convert into a cross reference listing. The name of the file is the name you gave your assembler when you directed it to produce the cross reference file.

CREF assumes that the filename extension is .CRF. If you do not specify a filename extension when you enter the cross reference filename, CREF will look for a file with the name you specify and the filename extension .CRF. If your cross reference file has a different extension, specify the extension when entering the filename.

See Section 5.4, Format of CREF Compatible Files, for a description of what CREF expects to see in the cross reference file. You will need this information only if your cross reference file was not produced by a Microsoft assembler.

Listing [crffile.REF]:

Enter the name you want the cross reference listing file to have. CREF will automatically give the cross reference listing the filename extension .REF.

If you want your cross reference listing to have the same filename as the cross reference file but with the filename extension .REF, simply press the carriage return key when the Listing prompt appears. If you want your cross reference listing file to be named anything else and/or to have any other filename extension, you must enter a response following the Listing prompt.

If you want the listing file placed on a drive or device other than the defualt drive, specify the drive or device when entering your response to the Listing prompt.

Special Command Characters

; Use a single semicolon (;) followed immediately by a carriage return at any time after responding to the cross reference prompt to select the default response to the Listing prompt. This feature saves time and overrides the need to answer the Listing prompt.

If you use the semicolon, CREF gives the listing file the filename of the cross reference file and the default filename extension .REF.

Example:

```
Cross reference [.CRF]: FUN;
```

CREF will process the cross reference file named FUN.CRF and output a listing file name FUN.REF.

<Ctrl+C> Use <Ctrl+C> at any time to abort the CREF session. If you enter an erroneous response, (the wrong filename), or an incorrectly spelled filename, you must press <Ctrl+C> to exit CREF then reinvoke CREF and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

5.2.4 Method 2: CREF <crffile>,<listing>

Enter:

```
CREF <crffile>,<listing>
```

CREF will be loaded into memory. Then, CREF immediately procedes to convert your cross reference file into a cross reference listing.

The entries following CREF are responses to the command prompts. The crffile and listing fields must be separated by a comma.

where: crffile is the name of a cross reference file produced by your assembler. CREF assumes that the filename extension is .CRF, which you may override by specifying a different extension. If the file named for the crffile does not exist, CREF will display the message:

```
Fatal I/O Error 110
in File:  <crffile>.CRF
```

Control then returns to your operating system.

listing is the name of the file you want to receive the cross reference listing of symbols in your program.

To select the default filename and extension for the listing file, enter a semicolon after you enter the crffile name.

Example:

```
CREF FUN;<Rtn>
```

This example causes CREF to process the cross reference file FUN.CRF and to produce a listing file named FUN.REF.

To give the listing file a different name, extension, or destination, simply specify these differences when entering the command line.

```
CREF FUN,B:WORK.ARG
```

this example causes CREF to process the cross reference file named RUN.CRF and to produce a listing file named WORK.ARG, which will be placed on the diskette in drive B:.

5.2.5 Format of Cross Reference Listings

The cross reference listing is an alphabetical list of all the symbols in your program.

Each page is headed with the title of the program or program module.

Then comes the list of symbols. Following each symbol name is a list of the line numbers where the symbol occurs in your program. The line number for the definition has a octothorpe sign (#) appended to it.

On the next page is a cross reference listing as an example:

```
CREF     (vers no.)     (date)

ENTX   PASCAL entry for initializing programs      comes from
                                                   TITLE directive

 Symbol Cross Reference        (# is definition)    Cref-1

AAAX00 . . . . . . . .       37#    38

BEGH00 . . . . . . . .       83     84#   154    176
BEGO00 . . . . . . . .       33    162
BEGX00 . . . . . . . .      113    126#   164    223

CESX00 . . . . . . . .       97     99#
CLNE00 . . . . . . . .       67     68#
CODE . . . . . . . . .       37    182
CONST. . . . . . . . .      104    104    105    110
CRCX00 . . . . . . . .       93     94#   210    215
CRDX00 . . . . . . . .       95     96#   216
CSXE00 . . . . . . . .       65     66#   149
CURH00 . . . . . . . .       85     86#   155

DATA . . . . . . . . .       64#    64    100    110
DGROUP . . . . . . . .      110#   111    111    111    127    153    171    172
DOSOFF . . . . . . . .       98#   198    199
DOSX00 . . . . . . . .      184    204#   219

ENDH00 . . . . . . . .       87     88#   158
ENDO00 . . . . . . . .       33#   195
ENDU00 . . . . . . . .       31#   197
ENDX00 . . . . . . . .      184    194#
ENDY00 . . . . . . . .       32#   196
ENTG00 . . . . . . . .       30#   187
ENTYCM . . . . . . . .      182#   183    221

FREX00 . . . . . . . .      169    170#   178

HDRF00 . . . . . . . .       71     72#   151
HDRV00 . . . . . . . .       73     74#   152
HEAP . . . . . . . . .       42     44    110
HEAPBEG. . . . . . . .       54#   153    172
HEAPLOW. . . . . . . .       43    171

INIU00 . . . . . . . .       31    161

MAIN_STARTUP. . . . .       109#   111    180
MEMORY . . . . . . . .       42     48#    48     49    109    110
```

PMUX00	69	70	150	
RECE00	81	82#		
REFE00	77	78#		
REPE00	79	80#		
RESE00	75	76#	148	
SKTOP.	59#			
SMLSTK	135	137#		
STACK.	53#	53	60	110
STARTMAIN.	163#	186#	200	
STKB00	89	90#	146	
STKH00	91	92#	160	

5.3 ERROR MESSAGES

All errors cause CREF to abort. Control is returned to your operating system.

All error messages are displayed in the format:

```
Fatal I/O Error <error number>
in File:  <filename>
```

Where: filename is the name of the file where the error occurs.

error number is one of the numbers in the following list of errors.

Number	Error
101	Hard data error Unrecoverable disk I/O error
102	Device name error Illegal device specification (for example X:FOO.CRF)
103	Internal error Report to Dynalogic Info-Tech Corporation
104	Internal error Report to Dynalogic Info-Tech Corporation
105	Device offline Disk drive door open, no printer attached, and so on.
106	Internal error Report to Dynalogic Info-Tech Corporation
108	Disk full
110	File not found
111	Disk is write protected
112	Internal error Report to Dynalogic Info-Tech Corporation
113	Internal error Report to Dynalogic Info-Tech Corporation

114 Internal error
Report to Dynalogic Info-Tech Corporation

115 Internal error
Report to Dynalogic Info-Tech Corporation

5.4 FORMAT OF CREF COMPATIBLE FILES

CREF will process files other than those generated by Macro Assembler as long as the file conforms to the format that CREF expects.

5.4.1 General Description of CREF File Processing

In essence, CREF reads a stream of bytes from the corss reference file (or source file), sorts them, then emits them as a printable listing file (the .REF file). The symbols are held in memory as a sorted tree. References to the symbols are held in a linked list.

CREF keeps track of line numbers in the source file by the number of end-of-line characters it encounters. Therefore, every line in the source file must contain at least an end-of-line character (see chart below).

CREF attempts to place a heading at the top of every page of the listing. The name it uses as a title is the text passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol (see chart below). If CREF encounters more than one title symbol in the source file, it uses the last title read for all page headings. If CREF does not encounter a title symbol in the file, the title line on the listing is left blank.

5.4.2 Format of Source Files

CREF uses the first three bytes of the source file as format specification data. The rest of the ifle is processed as a series of records that either begin or end with a byte that identifies the type of record.

First Three Bytes

(The PAGE directive in your assembler, which takes arguments for the page length and line length, will pass this information to the cross reference file.)

First Byte

The number of lines to be printed per page (page length range is from 1 to 255 lines).

Second Byte

The number of characters per line (line length range is from 1 to 132 characters).

Third Byte

The Page Symbol (07) that tells CREF that the two preceding bytes define listing page size.

If CREF does not see these first three bytes in the file, it uses default values for page size (page length: 58 lines; line length: 80 characters).

Control Symbols

The two charts show the types of records that CREF recognizes and the byte values and placement it uses to recognize record types.

Records have a Control Symbol (which identifies the record type) either as the first byte of the record or as the last byte.

Records That Begin with a Control Symbol

Byte Value	Control Symbol	Subsequent Bytes
01	Reference symbol	Record is a reference to a symbol name (1 to 80 characters)
02	Define symbol	Record is a definition of a symbol name (1 to 80 characters)
04	End of line	(none)
05	End of file	1AH

Records That End with a Control Symbol

Byte Value	Control Symbol	Preceding Bytes
06	Title defined	Record is titled text (1 to 80 characters)
07	Page length/ line length	One byte for page length followed by one byte for line length

For all record types, the byte value represents a control character. as follows:

```
01   Ctrl+A
02   Ctrl+B
04   Ctrl+D
05   Ctrl+E
06   Ctrl+F
07   Ctrl+G
```

The Control Symbols are defined as follows:

Reference symbol

Record contains the name of a symbol that is referenced. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

Define symbol

Record contains the name of a symbol that is defined. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

End of line

Record is an end of line symbol character only (04H or Ctrl+D).

End of file

Record is the end of file character (1AH).

Title defined

ASCII characters of the title to be printed at the top of each listing page. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used for th title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the record is not encountered, the title line on the listing is left blank.

Page length/line length

The first byte of the record contains the number of lines to be printed per page (range is from 1 to 255 lines). The second byte contains the number of characters to be printed per line (range is from 1 to 132 characters). The default page length is 58 lines. The default line length is 80 characters.

Summary of CRF File Record Contents

byte contents	length of record
01 symbol name	2-81 bytes
02 symbol name	2-81 bytes
04	1 byte
05 1A	2 bytes
title text 06	2-81 bytes
PL LL 07	3 bytes

PART II - ASSEMBLY LANGUAGE TOOLS

SECTION 6 - EXE2BIN

6.1 EXE2BIN

Format: EXE2BIN filespec [d:][filename][.ext]

Purpose: Convert files from .EXE format to binary format

Remarks: The first parameter is the input file; if no extension is given, it defaults to .EXE. The second parameter is the output file. If no drive is given, the drive of the input file is used; if no filename is given, the filename of the input file is used; if no extension is given, .BIN is used.

The input file must be in valid .EXE format produced by the linker. The "resident", or actual code and data part of the file, must be less than 64K. There must be no STACK segment. Two kinds of conversion are possible depending on the specified initial CS:IP:

1. If CS:IP is not specified, a pure binary conversion is assumed. If segment fix-ups are necessary, the following prompt appears:

   ```
   Fix-up needed - base segment (hex):
   ```

   By typing a legal hexadecimal number and then <Rtn>, execution will continue.

2. If CS:IP is specified as 100H, then it is assumed the file is to be run as a .COM file ORGed at 100H, and the first 100H of the file is to be deleted. No segment fix-ups are allowed, as .COM files must be segment relocatable.

If CS:IP does not meet one of these criteria or meets the .COM file criterion, but has segment fix-ups, the following error message is displayed:

```
File cannot be converted
```

Note that to produce standard .COM files with the Macro Assembler, one must both ORG the file at 100H and specify the first location as the start address (this is done in the END statement).

For example:

```
        ORG     100H
    START:
        .
        .
        .
        END     START
```

PART III - APPENDICES

APPENDIX A - ASCII CHARACTER CODES

APPENDIX A

ASCII Character Codes

ASCII Code		Character
000		NUL
001	☺	SOH
002	☻	STX
003	♥	ETX
004	♦	EOT
005	♣	ENQ
006	♠	ACK
007		BEL
008		BS
009		HT
010		LF
011		VT
012		FF
013		CR
014	♫	SO
015	☼	SI
016	►	DLE
017	◄	DC1
018	↕	DC2
019	‼	DC3
020	¶	DC4

ASCII Code		Character
021	§	NAK
022	▬	SYN
023	↨	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESCAPE
028		FS
029		GS
030		RS
031		US
032		SPACE
033		!
034		"
035		# (octothorpe)
036		$ (dollar sign)
037		% (percent)
038		& (ampersand)
039		' (apostrophe)
040		((right parenthesis)
041		) (left parenthesis)
042		* (asterisk)
043		+ (plus sign)

ASCII Code	Character
044	, (comma)
045	- (hyphen)
046	. (period)
047	/ (slash)
048	0
049	1
050	2
051	3
052	4
053	5
054	6
055	7
056	8
057	9
058	: (colon)
059	; (semi-colon)
060	< (less than)
061	= (equal sign)
062	> (greater than)
063	? (question mark)
064	@ (at sign)
065	A
066	B

ASCII Code	Character
067	C
068	D
069	E
070	F
071	G
072	H
073	I
074	J
075	K
076	L
077	M
078	N
079	O
080	P
081	Q
082	R
083	S
084	T
085	U
086	V
087	W
088	X
089	Y
090	Z
091	[(left bracket)

ASCII Code	Character
092*	(backslash)
093	] (right bracket)
094	^ (caret)
095	< (less than)
096	' (apostrophe)
097	a
098	b
099	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s

ASCII Code	Character
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{ (left curlicue)
124	¦ (line)
125	} (right curlicue)
126	~ (non-trivial blank)
127	DEL
128	Ç
129	ü
130	é
131	â
132	ä
133	à
134	å
135	ç
136	ê
137	ë
138	è

ASCII Code	Character
139	ï
140	î
141	ì
142	Ä
143	Å
144	E
145	æ
146	Æ
147	ô
148	ö
149	o
150	û
151	u
152	ÿ
153	Ö
154	Ü
155	¢
156	£
157	¥
158	Pt
159	ƒ
160	á
161	í

ASCII Code	Character
162	ó
163	ú
164	ñ
165	Ñ
166	ª
167	º
168	¿
169	⌐
170	¬
171	1/2
172	1/4
173	¡
174	<<
175	>>
176	░
177	▒
178	▓
179	│
180	┤
181	╡
182	╢
183	╖
184	╕

ASCII Code	Character
185	╣
186	║
187	╗
188	╝
189	╜
190	╛
191	┐
192	└
193	┴
194	┬
195	├
196	─
197	┼
198	╞
199	╟
200	╚
201	╔
202	╩
203	╦
204	╠
205	═
206	╬
207	╧

ASCII Code	Character
208	╨
209	╤
210	╥
211	╙
212	╘
213	╒
214	╓
215	╫
216	╪
217	┘
218	┌
219	█
220	▄
221	▌
222	▐
223	▀
224	α
225	β
226	Γ
227	π
228	Σ
229	σ
230	µ

ASCII Code	Character
231	τ
232	Φ
233	Θ
234	Ω
235	δ
236	∞
237	ø
238	ε
239	∩
240	≡
241	±
242	≥
243	≤
244	⌠
245	⌡
246	÷
247	≈
248	°
249	•
250	·
251	√
252	^{n}
253	^{2}
254	■
255	(blank 'FF')

PART III - APPENDICES

APPENDIX B - BASIC DISK I/O

APPENDIX B

BASIC DISK I/O

Disk I/O procedures for the beginning BASIC user are examined in this appendix. If you are new to bASIC or if you're getting errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. (Will our system appends a default .BAS to filenames?)

B.1 - PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE <filename>[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD <filename>[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.
RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename> AS <new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

B.2 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

B.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

B.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
OPEN
PRINT#
INPUT#
WRITE#
PRINT# USING
LINE INPUT#
CLOSE
EOF
LOC
```

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.

   ```
   OPEN "O",#1,"DATA"
   ```

2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)

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

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.

   ```
   CLOSE #1
   OPEN "I",#1,"DATA"
   ```

4. Use the INPUT# statement to read data from the sequential file into the program.

INPUT#1,X$,Y$,Z$

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement:

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

B.3.1.1 Adding Data to a Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.

2. OPEN a second file called "COPY" in "O" mode.

3. Read in the data in "NAMES" and write it to "COPY".

4. CLOSE "NAMES" and KILL it.

5. Write the new information to "COPY".

6. Rename "COPY" as "NAMES" and CLOSE.

7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

B.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN
FIELD
LSET/RSET
GET
PUT
CLOSE
LOC
MKI$
CVI
MKS$
CVS
MKD$
CVD

B.3.2.1 Creating a Random File

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random fileg.

```
FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$-MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Note

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS AS$, 8 AS P$
30 INPUT "2-DIGIT CODE",CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

B.3.2.2 Access a Random File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
```

   Note:

   In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions; CVI for integers, CVS for single precision values, and CVD for double precision values.

Program B-5 accesses the random file "FILE" that was created in Program B-4. BY inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4AS A$, 8AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number". The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different pqrt numbers. Lines 900-960 initialize the data file by writing chr$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired fucntion number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE:
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6)
 THEN PRINT "BAD FUNCTION NUMBER": GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
 IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(R%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$-MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##>##:;CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F%)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
```

```
620 IF (O%-S%)<0 THEN PRINT "ONLY";O%;" IN STOCK":GOTO 600
630 O%=O%-S%
640 IF O%=<CVI(R%) THEN PRINT "QUANTITY NOW";O%;
 " REORDER LEVEL" CVI(R$)
650 LAST O$=MKI$(O%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I-1 TO 100
710 GET#1,I
720 IF CVI(O$)<CVI(R$) THEN PRINT D$;" QUANTITY";
 CVI(O$) TAB(50) "REORDER LEVEL"
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
 GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B%:IF B$<>"Y" THEN RETURN
920 LSET F$-CHR$(255)
930 FOR I+1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM B-6 - INVENTORY

PART III - APPENDICES

APPENDIX C - SUMMARY OF BASIC ERROR CODES AND ERROR MESSAGES

APPENDIX C

Summary of Error Codes and Error Messages

CODE	NUMBER	MESSAGE
NF	1	NEXT without For A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parentheses, misspelled command or statement, incorrect punctuation, etc.).
RG	3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1) a negative or unreasonably large subscript; 2) a negative or zero argument with LOG 3) a negative argument to SQR 4) a negative mantissa with a non-integer exponent

CODE	NUMBER	MESSAGE
		5) a call to a USR function for which the starting address has not yet been given 6) an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
OV	6	Overflow The result of a calculation is too large to be represented in BASIC's number format. If underflow occurs, the result is zero and execution occurs without an error.
OM	7	Out of memory A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
UL	8	Undefined line A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.
BS	9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
DD	10	Redimensioned array Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

CODE	NUMBER	MESSAGE
/0	11	Division by zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
ID	12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
TM	13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.
OS	14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.
LS	15	String too long An attempts is made to create a string more than 255 characters long.
ST	16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.

CODE	NUMBER	MESSAGE
CN	17	Can't continue An attempt is made to continue a program that: 1) has halted due to an error 2) has been modified during a break in execution, or 3) does not exist.
UF	18	Undefined user function A USR function is called before the function definition (DEF statement) is given.
	19	No RESUME An error trapping routine is entered but contains no RESUME statement.
	20	RESUME without error A RESUME statement is encountered before an error trapping routine is entered.
	21	Unprintable error An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.
	22	Missing operand An expression contains an operator with no operand following it.
	23	Line buffer overflow An attempt is made to input a line that has too many characters.
	26	FOR without NEXT A FOR was encountered without a matching NEXT.
	29	WHILE without WEND A WHILE statement does not have a matching WEND.

CODE	NUMBER	MESSAGE
	30	WEND without WHILE A WEND was encountered without a matching WHILE.
	50	Field overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
	51	Internal error An internal malfunction has occurred in BASIC. Report to Dynalogic the conditions under which the message appeared.
	52	Bad file number A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
	53	File not found A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.
	54	Bad file mode An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.
	55	File already open A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
	57	Disk I/O error An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

CODE	NUMBER	MESSAGE
	58	File already exists The filename specified in a NAME statement is identical to a filename already in use on the disk.
	61	Disk full All disk storage space is in use.
	62	Input past end An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.
	63	Bad record number In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
	64	Bad file name An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
	66	Direct statement in file A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.
	67	Too many files An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

PART III - APPENDICES

APPENDIX D - MATHEMATICAL FUNCTIONS IN BASIC

APPENDIX D

Mathematical Functions

Derived Functions

Functions that are not intrinsic to BASIC may be calculated as follows.

FUNCTION	BASIC EQUIVALENT
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(-X*X+1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)=1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))*2+1
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	CSCH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))

FUNCTION	BASIC EQUIVALENT
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1)/2

PART III - APPENDICES

APPENDIX E - ALPHABETICAL SUMMARY OF BASIC COMMANDS, FUNCTIONS AND STATEMENTS

APPENDIX E

ALPHABETICAL SUMMARY OF BASIC COMMANDS, FUNCTIONS & STATEMENTS