# utility software package reference manual

# utility software package reference manual

## for 8086 microprocessors

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

## LIMITED WARRANTY

**MICROSOFT, Inc.** shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

**THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.**

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.



8407-100
Part Number 14F06A

## Package Contents

1 diskette, with the following files:
   M86.EXE
   LINK.EXE
   LIB.EXE
   CREF.EXE

1 binder with 4 manuals
   MACRO-86 Macro Assembler Manual
   MS-LINK Linker Utility Manual
   MS-LIB Library Manager Manual
   MS-CREF Cross Reference Facility Manual

1 Utility Software Reference Card

## System Requirements

Each utility requires different amounts of memory.

MACRO-86 - 96K bytes of memory minimum:
   64K bytes for code and static data
   32K bytes for run space

MS-LINK - 54K bytes of memory minimum:
   44K bytes for code
   10K bytes for run space

MS-LIB - 38K bytes of memory minimum:
   28K bytes for code
   10K bytes for run space

MS-CREF - 24K bytes of memory minimum:
   14K bytes for code
   10K bytes for run space

1 disk drive

1 disk drive if and only if output is sent to the same physical diskette from which the input was taken. None of the utility programs in this package allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Microsoft

Welcome to the Microsoft family of products.

Microsoft, Inc. is recognized as the leader in microcomputer software. Microsoft BASIC interpreter, in its several versions, has become the standard high-level programming language used in microcomputers. Microsoft, Inc. continues to supply consistently high-quality software which sets the standard for software quality for all types of users.

In addition to the Utility Software Package and Microsoft BASIC interpreter, Microsoft sells other full-feature language compilers, language subsets, and operating system products. Microsoft offers a "family" of software products that both look alike from one product to the next, and can be used together for effective program development.

For more information about other Microsoft products, contact:

Microsoft, Inc.
10700 Northup Way
Bellevue, WA 98004
(206) 828-8080

# Contents

# GENERAL INTRODUCTION

The Microsoft Utility Software Package includes four utility programs used for developing assembly language programs. In addition, the MS-LINK Linker Utility is used with all of Microsoft's 16-bit language compilers.

## Major Features

### MACRO-86 Macro Assembler

Microsoft's MACRO-86 Macro Assembler is a very rich and powerful assembler for 8086 based computers. MACRO-86 is more complex than any other microcomputer assembler.

MACRO-86 supports most of the directives found in Microsoft's MACRO-80 Macro Assembler. Macros and conditionals are Intel 8080 standard.

MACRO-86 is upward compatible with Intel's ASM-86, except Intel codemacros, macros, and a few $ directives.

Some relaxed typing, so that if the user enters a typeless operand for an instruction that accepts one one type of operand, MACRO-86 assembles the statement correctly instead of returning an error message.

MS-LINK Linker Utility

MS-LINK is a virtual linker, which can link programs that are larger than available memory

MS-LINK produces relocatable executable object code.

MS-LINK knows how to handle user-defined overlays.

MS-LINK can perform multiple library searches, using a dictionary library search method.

MS-LINK prompts the user for input and output modules and other link session parameters.

MS-LINK can be run with an automatic response file to answer the linker prompts.

MS-LIB Library Manager

MS-LIB can add, delete, and extract modules in the user's library of program files.

MS-LIB prompts the user for input and output file and module names.

MS-LIB can be run with an automatic response file to answer the library prompts.

MS-LIB produces a cross reference of symbols in the library modules.

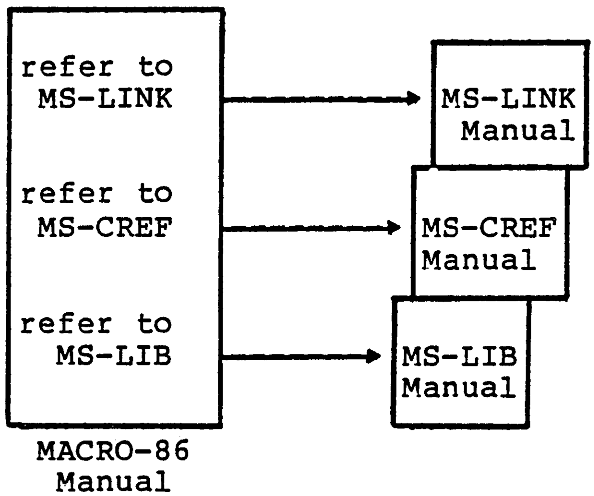
MS-CREF Cross Reference Facility

MS-CREF produces a cross reference listing of all symbolic names in the source program, giving both the source line number of the definition and the source line numbers of all other references to them.

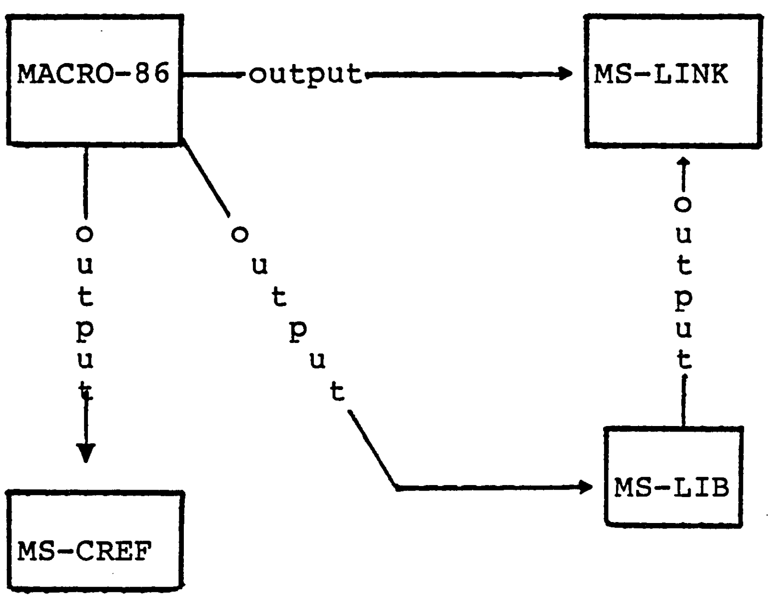
## Using These Manuals

These manuals are designed to be used as a set and individually. Each manual is mostly self-contained and refers to the other manuals only at junctures in the software. The Overview given below describes generally the flow of program development from creating a source file through program execution. The processes described in this overview are echoed and expanded in overviews in each of the four manuals.

Also, note that each manual has its own index. The four individual indexes are compiled into a general index at the back of this documentation set.




Each of the four manuals is used independently. References between manuals reflect junctures in the software.



## Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

- [ ]   Square brackets indicate that the enclosed entry is optional.

- < >   Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.

- { }   Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

- ...   Ellipses indicate that an entry may be repeated as many times as needed or desired.

- CAPS   Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

User has an option;
user may stop here,
or may enter more

User enters a value
here to replace the
"dummy" entry and
the angle brackets

Enter as many more
parameters as you
want, up to end of line

```
CALL  (<parameter>  [,<parameter>...])  <RETURN>
```

upper case
inside angle
brackets, press
this key

Enter CAPS
exactly as
shown

Enter punctuation as shown

## Learning More About Assembly Language Programming

These manuals explain how to use Microsoft's Utility Software Package, but they do not teach users how to program in assembly language.

We assume that the user of The Utility Software Package will have had some experience programming in assembly language. If you do not have any experience, we suggest two courses:

1. Gain some experience on a less sophisticated assembler.

2. Refer to any or all of the following books for assistance:


Morse, Stephen P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.

Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA: Osbourne/McGraw-Hill, 1980.

The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1979.

8086/8087/8088 Macro Assembly Language Reference Manual. Santa Clara, CA: Intel Corporation, 1980.


NOTE

Some of the information in these books was based on preliminary data and may not reflect the final functional state. Information in your Microsoft manuals was based on Microsoft's development of its 16-bit software for the 8086 and 8088.

## Overview of Program Development

This overview describes generally the steps of program development. Each step is described fully in the individual product manuals. The numbers in the descriptions match the numbers in the facing diagram.

1. Use EDLIN (the editor in Microsoft's MS-DOS), or other 8086 editor compatible with your operating system, to create an 8086 assembly language source file. Give the source file the filename extension .ASM (ASMRO-86 recognizes .ASM as default).

2. Assemble the source file with MACRO-86, which outputs an assembled object file with the default filename extension .OBJ (2a). Assembled files, the user's program files (2b), can be linked together in step 3.

   MACRO-86 (optionally) creates two types of listing file:

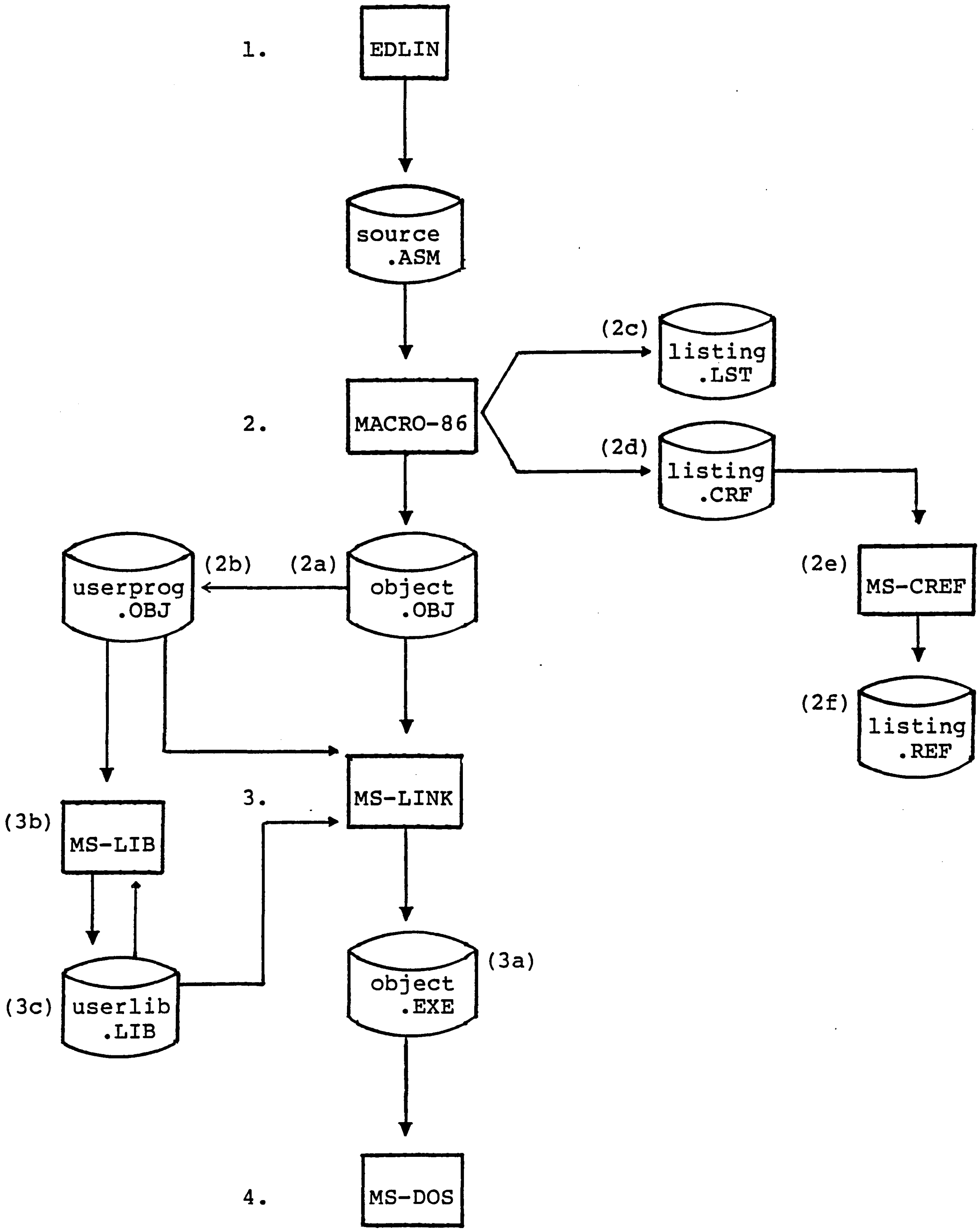
   (2c) a normal listing file which shows assembled code with relative addresses, source statements, and full symbol table;

   (2d) a cross-reference file, a special file with special control characters that allow MS-CREF (2e) to create a list showing the source line number of every symbol's definition and all references to it (2f). When a cross reference file is created, the normal listing file (with the .LST extension) has line number placed into it as references for line numbers following symbols in the cross reference listing.

3. Link one or more .OBJ modules together, using MS-LINK, to produce an executable object file with the default filename extension .EXE (3a).

   While developing your program, you may want to create a library file for MS-LINK to search to resolve external references. Use MS-LIB (3b) to create user library file(s) (3c) from existing library files (3c) and/or user program object files (2b).

4. Run your assembled and linked program ,the .EXE file (3a), under MS-DOS, or your operating system.


1.
EDLIN
source .ASM
2.
MACRO-86
(2c)
listing .LST
(2d)
listing .CRF
(2e)
MS-CREF
(2f)
listing .REF
(2a)
object .OBJ
(2b)
userprog .OBJ
3.
MS-LINK
(3b)
MS-LIB
(3c)
userlib .LIB
(3a)
object .EXE
4.
MS-DOS

# Microsoft
# Software Problem Report

Use this form to report errors or problems in: ☐ MS-FORTRAN

☐ OTHER - Please indicate name of manual.

____________________

Release (or version) number: ____________________

Date ____________________

Report only one problem per form.

Describe your hardware and operating system: ____________________

Please supply a concise description of the problem and the circumstances surrounding its occurrence. If possible, reduce the problem to a simple test case. Otherwise, include all programs and data in <u>machine readable form</u> (preferably on a diskette). If a patch or interim solution is being used, please describe it.

This form may also be used to describe suggested enhancements to Microsoft software.

Problem Description:

-over-

Did you find errors in the documentation supplied with the software?
If so, please include page numbers and describe:

Fill in the following information before returning this form: ____________________

Name ______________________________ Phone ____________________

Organization ______________________________________________

Address ____________________ City ____________ State ______ ZIP __________

RETURN FORM TO: MICROSOFT, INC.
10700 Northup Way
C 97200
Bellevue, Wa. 98004

# microsoft MACRO-86 macro assembler manual

## 8086 and 8088 implementations

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

## LIMITED WARRANTY

**MICROSOFT, Inc.** shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

**THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.**

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.



8407A-100-01

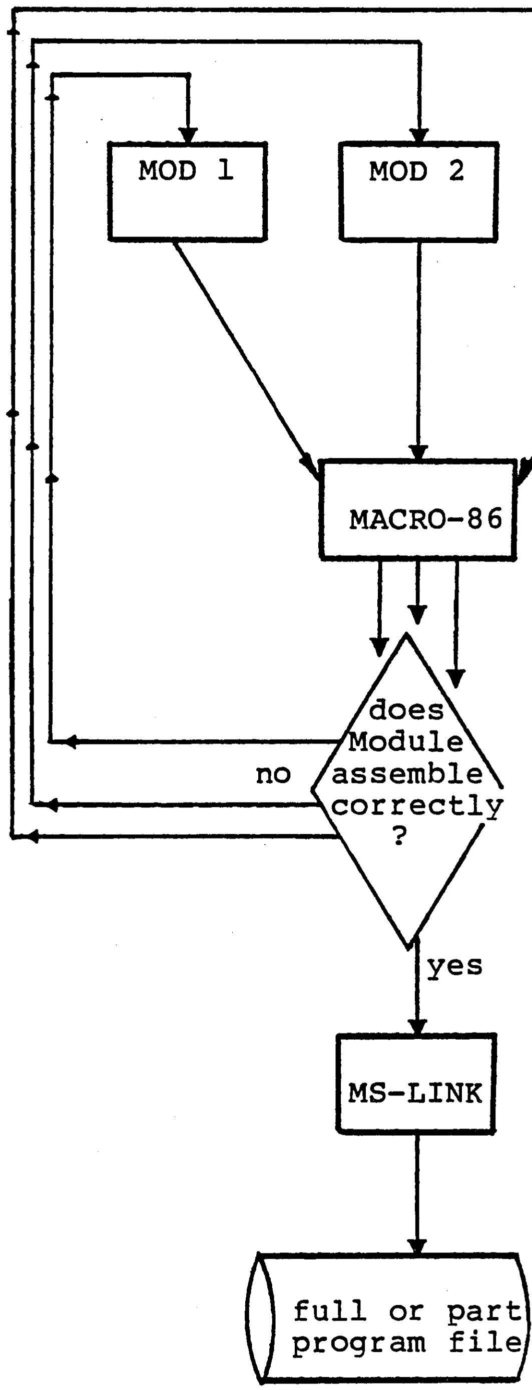
# INTRODUCTION

## Features and Benefits of MACRO-86

Microsoft's MACRO-86 Macro Assembler is a very rich and powerful assembler for 8086 based computers. MACRO-86 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 an 8086 or 8088 microprocessor. Even though MACRO-86 is more complex than any other microcomputer assembler, it is easy to use.

MACRO-86 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 Microsoft's MS-LINK Linker Utility 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 some 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

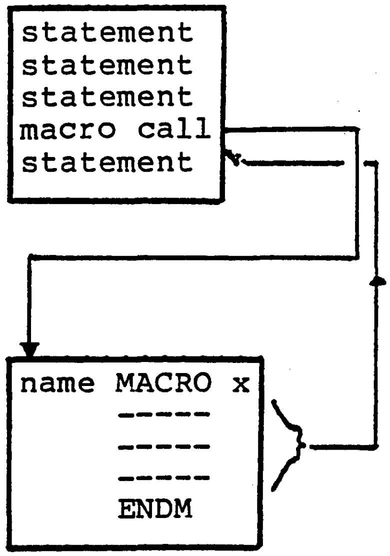
When the individual modules are ready, they can be linked singly or into one or more larger modules

MACRO-86 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.

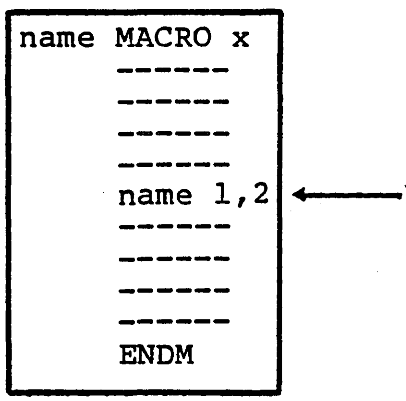
This block of code is given a name, called a macro. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, a simple "call" to the macro is placed in the source file. MACRO-80 expands the macro call by assembling the block of 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, a macro can 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-86 checks first the symbol table it created to see if the instrucion or directive is a macro name. If it is, MACRO-86 "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-86 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-86 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-86 can test for blank or nonblank arguments, for defined or not-defined symbols, for equivalence, for first assembly pass or second, and MACRO-86 can compare strings for identity or difference. The conditional directives simplify the evaluation of assembly results, and make programming the testing code for conditions easier as well as more powerful.

MACRO-86'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 assembles the whole conditional block

```
statement
statement
statement
IF <exp true>
    ---------
    ---------
    ---------
ELSE
    ---------
    ---------
    ---------
ENDIF
statement
statement
    .
    .
    .
```

If the condition in the expression is false, MACRO-86 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-86 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-86. Processor instructions and some directives (e.g., .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-86 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-86 is upward compatible, with MACRO-80 and with Intel's ASM86, except Intel codemacros and macros.

MACRO-86 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-86 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 meant one of three instructions:

2 MOV AL,BYTE PTR WORDLBL

3 MOV AL,<other>

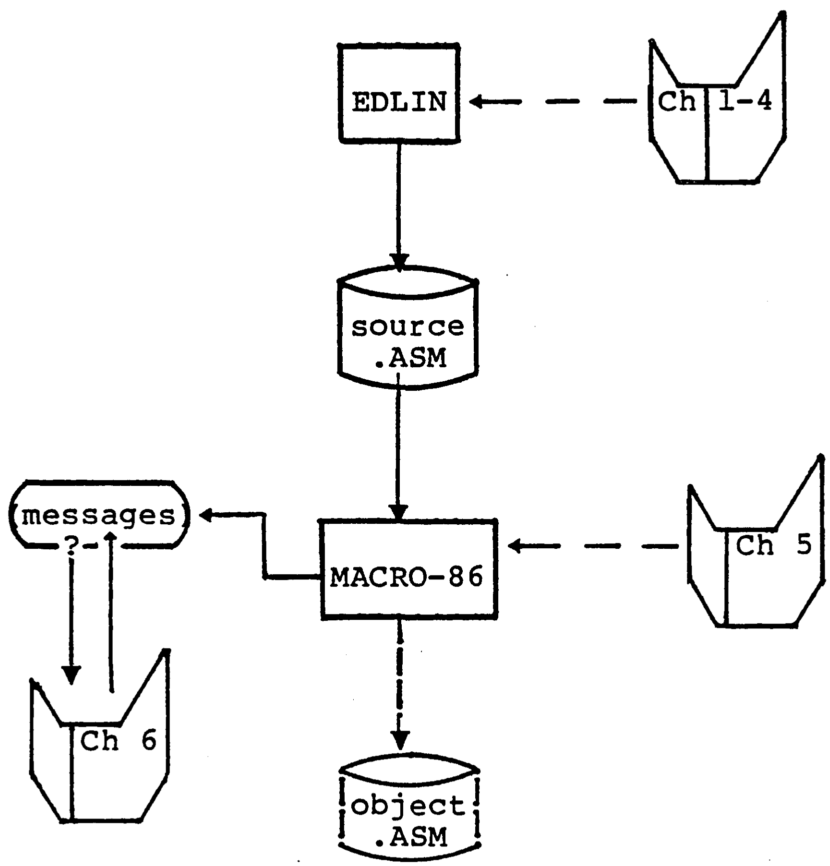
1 MOV AX,WORDLBL

MACRO-86 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-86 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-86 assumes is "correct."

Overview of MACRO-86 Operation

The first task is to create a source file. Use EDLIN (the resident editor in Microsoft's MS-DOS operating system), or other 8086 editor compatible with your operating system, to create the MACRO-86 source file. MACRO-86 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 contraints described in Chapters 1-4 in this manual.

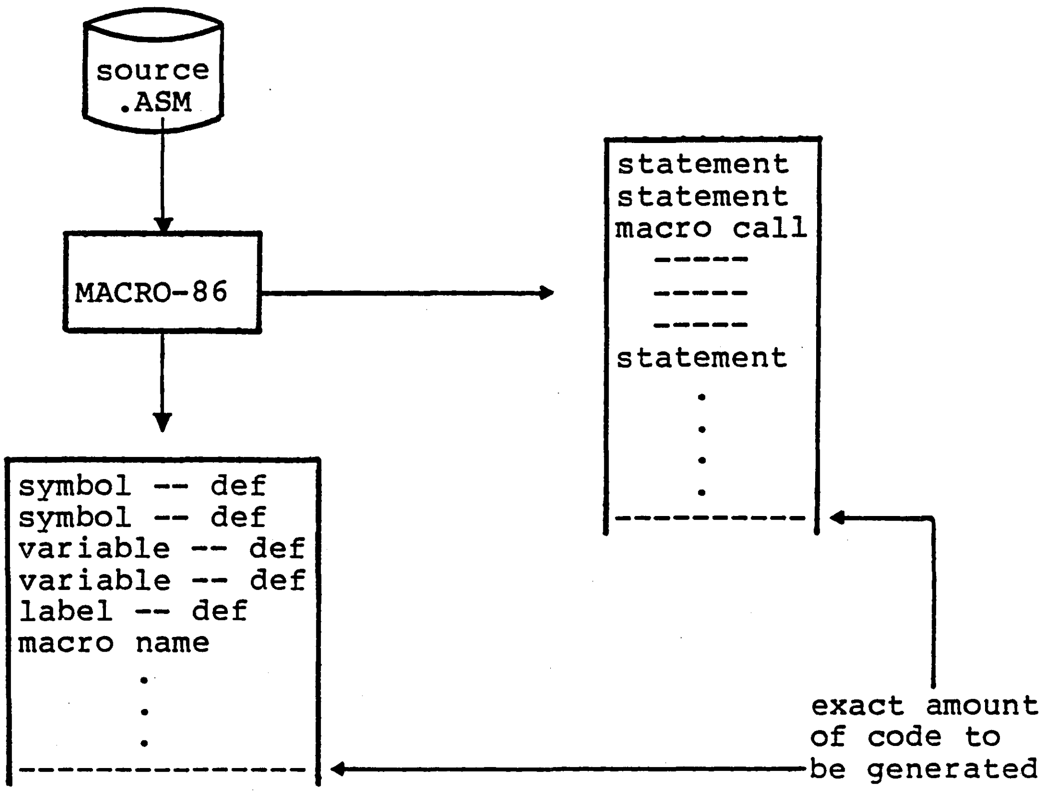
When the source file is ready, run MACRO-86 as described in Chapter 5. Refer to Chapter 6 for explanations of any messages displayed during or immediately after assembly.



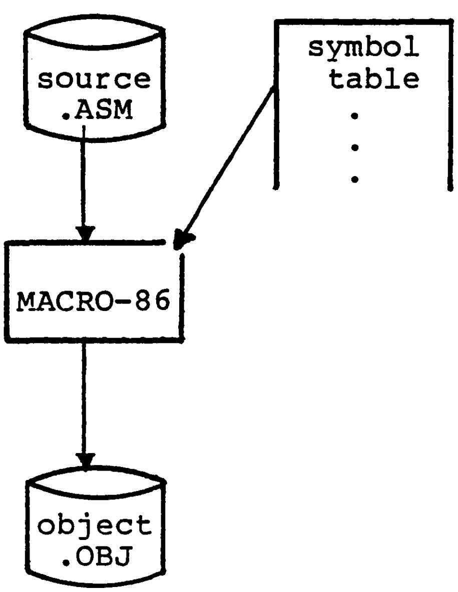
MACRO-86 is a two-pass assembler. This means that the source file is assembled twice. But slightly different actions occur during each pass. During the first pass, the assembler evaluates the statements and expands macro call statements, calculates the amount of code it will generate, and builds a symbol table where all symbols, variables, labels, and macros are assigned values. During the second pass, the 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 Microsoft's MS-LINK Linker Utility. 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 .OBJ modules by MS-LINK (refer to the MS-LINK Linker Utility Manual for further explanation and instructions). The .OBJ modules can also be processed with Microsoft's MS-LIB Library Manager (refer to the MS-LIB Library Manager Manual for further explanation and instructions).

The source file can also be assembled without creating an .OBJ file. All the 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.

PASS 1

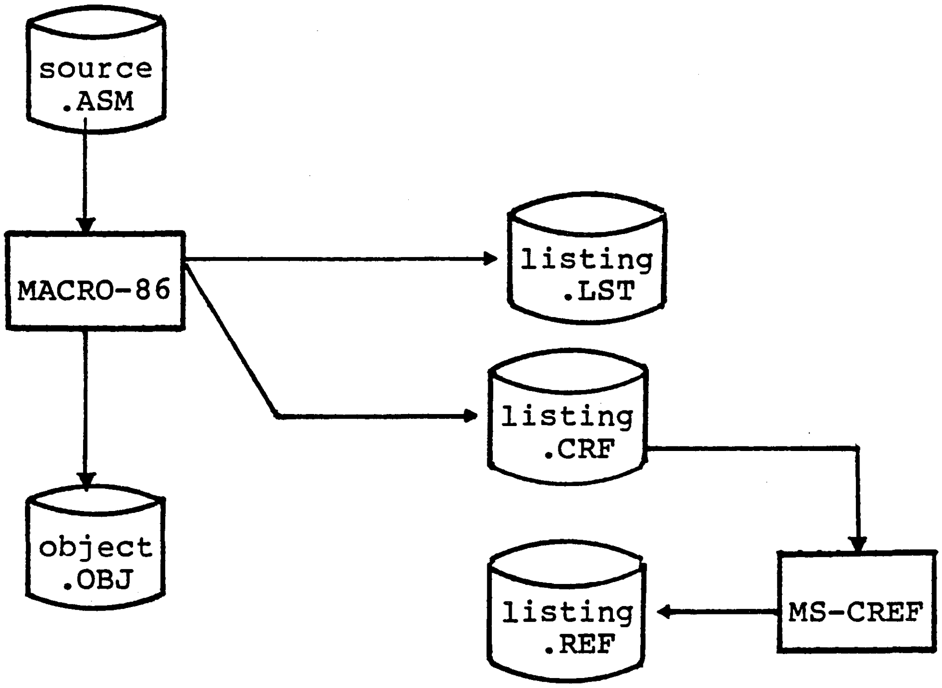

source
.ASM
MACRO-86
statement
statement
macro call
statement
symbol -- def
symbol -- def
variable -- def
variable -- def
label -- def
macro name
exact amount
of code to
be generated

PASS 2


source
.ASM
symbol
table
MACRO-86
object
.OBJ

MACRO-86 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 MS-CREF, the file is converted into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical 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 MS-CREF Cross Reference Facility Manual for further explanation and instructions.)


source
.ASM
MACRO-86
listing
.LST
listing
.CRF
object
.OBJ
listing
.REF
MS-CREF

## System Requirements

The MACRO-86 Macro Assembler requires 96K bytes of memory minimum:

64K bytes for code and static data
32K bytes for run space

1 disk drive

1 disk drive if and only if output is sent to the same physical diskette from which the input was taken. MACRO-86 does not allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

Contents

# CHAPTER 1

# CREATING A MACRO-86 SOURCE FILE

To create a source file for MACRO-86, you need to use an editor program, such as EDLIN in Microsoft's MS-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 chapter and the three following chapters when creating the file.

In this chapter, you will find discussions of the statement format and introductory descriptions of its components. In Chapter 2, you will find full descriptions of names: variables, labels, and symbols. In Chapter 3, you will find full descriptions of expressions and their components, operands and operators. In Chapter 4, you will find full descriptions of the assembler directives.

## 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-86 expects a specific three character filename extension, .ASM. Whenever you run MACRO-86 to assemble your source file, MACRO-86 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-86, 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-86, it is impossible to omit the filename extension. When you assemble a source file without a filename extension, MACRO-86 will assume that the source has a .ASM extension because you would not be specifying an extension. When MACRO-86 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-86 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 dicussions of the assembler directives (Section 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-86 is run (see Section 5.3, Command Switches). The input radix may be changed two ways:

1. The .RADIX directive (see Section 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<br>621O |
| Decimal | 0-9 | (none) or D | 9384 (default)<br>8149D<br>(when .RADIX directive changes default radix to not decimal.) |
| Hexadecimal | 0-9, A-F | H | 0FFH<br>80H<br>(first character must be numeral in range 0-9) |

What's in a Source File?

A source file for MACRO-86 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-86 to prepare data for use in and by instructions.

Statement format is described in Section 1.2. The parts of a statement are described in Sections 1.3-1.6 and in Chapters 2-4. Statements are usually placed in blocks 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-86 where program execution should begin.

Section 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 Chapter 3 as a prelude to the discussion of operands and operators.

## 1.2 STATEMENT LINE FORMAT

Statements in source files follow a strict format, which allows some variations.

MACRO-86 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-86 Instruction statements usually 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 normally names are started in column one.

Names may be any length you choose. However, MACRO-86 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 Chapter 2 apply the same to MACRO names, the discussion of macro names is better left to the section described the macro facility.

MACRO-86 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 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.

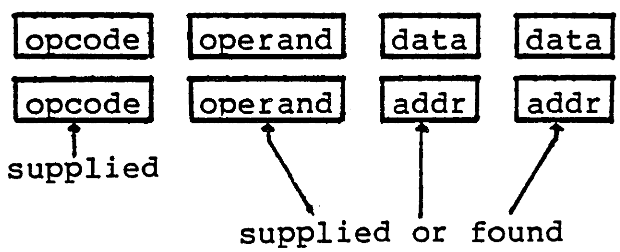
Comments are ignored by MACRO-86. Comments do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code. Comments are not required for anything but human understanding.

Action

The action field contains either an 8086 instruction mnemonic or a MACRO-86 assembler directive. Refer to Section 4.1 for some general discussion and to Appendix C for a list of 8086 instruction mnemonics. The MACRO-86 directives are described in detail in Section 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 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 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:

$\boxed{\text{opcode}}$ $\boxed{\text{dest-operand}}$ , $\boxed{\text{source-operand}}$

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:

$\boxed{\text{directive}}$ $\boxed{\text{operand}}$

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 ffrom 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-86 supports the following operands and operators in the expression field (shown in order of precedence):

**Operands**

- Immediate
  - (incl. symbols)
- Register
- Memory
  - label
  - variables
    - 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, FIELD
  - [ ], ( ), < >
- segment override(:)
- PTR, OFFSET, SEG, TYPE, THIS,
- 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. Refer to Sections 3.2, Operands, and 3.3, Operators, for details of operands and operators.

Contents

# CHAPTER 2

# NAMES: LABELS, VARIABLES, AND SYMBOLS

Names are used in several capacities throughout MACRO-86, 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. However, MACRO-86 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 chapter introduces you to the basic ways to define and use names. You will discover additional uses as you study the chapters on Expressions and Action, and as you use MACRO-86.

MACRO-86 supports three types of names in statement lines: labels, variables, and symbols. This chapter covers how to define and use these three types of names.

## 2.1 LABELS

Labels are names used as targets for JMP, CALL, and LOOP instructions. MACRO-86 assigns an address to each label as it is defined. When you use a label as an operand for JMP, CALL, or LOOP, MACRO-86 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 4.2.1, Memory Directives.

   NEAR and FAR are discussed under the 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 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.

Examples:

```
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 4.2.1, Memory Directives.

   Examples:

```
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 CS.) (MACRO-86 requires that a label 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 of 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 is more than 64K bytes between the label reference and the label definition.

NEAR and FAR cause MACRO-86 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).

## 2.2 VARIABLES

Variables are names used in expressions (as operands to instructions and directives).

A variable represents an address where a specified value may be found.

Variables 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 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 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 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 defined. 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 |

## 2.3 SYMBOLS

Symbols are names defined without reference to a Define directive or to code. Like variables, symbols are also used in expressions as operands to instructions and directives.

Symbols are defined three ways:

1. <name> EQU <expression>

   Use the EQU directive. See EQU directive in Section 4.2.1, Memory Directives.

   <expression> may be another symbol, an 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 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 4.2.1, Memory Directives.

   Example:

   ```
   EXTRN BAZ:ABS
   ```

   BAZ must be defined by an EQU or = directive to a valid expression.

Contents

# CHAPTER 3

## EXPRESSIONS: OPERANDS AND OPERATORS

Chapter 1 provided a brief introduction to expressions. Basically, expression is the term used to indicate values on which an instruction or directive performs 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-86 supports. The discussion of memory organization in a MACRO-86 program acts as a preface to the descriptions of operands and operators, and as a link to topics discussed in Chapter 2.

## 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-86 leaves everything to do with segments to MS-LINK. MS-LINK resolves all references. For that reason, MACRO-86 does not check (because it cannot) 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-86 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 makes values within a particular segment less immediately 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 completed, 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-86 to generate an immediate instruction; that is, to use the address of the value instead of the value itself.

There are two kinds of references 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-86 first looks for the segment of the reference. MACRO-86 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 its 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-86 compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they are different, MACRO-86 returns an error (you must use a FAR JMP or CALL).

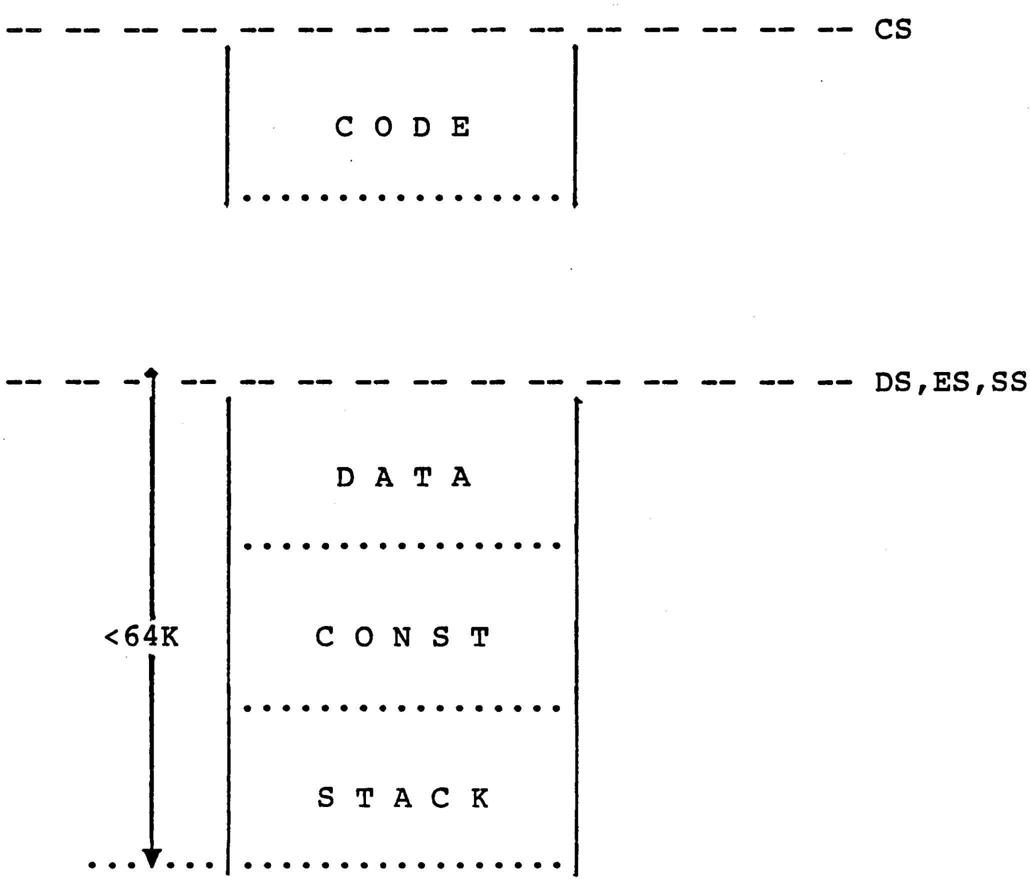
When you enter a FAR JMP or FAR CALL, you are changing both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

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;
        MOV     DS,AX     ;you initialize DS, do this
                          ;as soon as possible, especially
                          ;before any DS relative references
          .
          .
          .
```

As a diagram, this arrangement could be represented as follows:



Given this arrangement, a statement like:

```
MOV AX,<variable>
```

causes MACRO-86 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-86 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 DGROUP, you need a statement like:

```
MOV AX,OFFSET DGROUP:<variable>
```

<or>

MOV AX, OFFSET DS:<variable>

MACRO-86 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), assumptions 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 is given 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-86 returns a phase error.

Because pass 1 must keep correct track of the relative offset, some references must be know 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-86 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-86 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 when it still has not seen the definition? This is one of the main reasons for two passes. And, unless MACRO-86 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-86 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-86 includes a number of operators to help the assembler. These operators tell MACRO-86 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
```

i.e., MASM handles forward symbolic constants — GOOD!

This statement causes MACRO-86 to generate a move from memory instruction on pass 1. By using the OFFSET operator, we can cause MACRO-86 to generate an immediate operand instruction.

maybe could use "PTR" operator . . . nope!

```
MOV AX,OFFSET FOO ;OFFSET says use the address of
FOO
```

FOO doesn't have an address!

Because OFFSET tells MACRO-86 to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code. But doesn't get the value of FOO! unless symbolic value is equivalent to offset on 2nd pass . .

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:

use of "ptr" op with "far"/"near"

```
CALL FAR PTR <forward-label>
```

Think this works equally as "FAR", without "PTR"

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-86 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
```

MOV [BX], LOW FOO ?

These statements tell MACRO-86 that FOO is a byte immediate. A smaller instruction is generated.

## 3.2 OPERANDS

An operand may be any one of three types: Immediate, Registers, or Memory operands. There is no restriction on combining 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
```

### 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 without numeric notation appended will be treated as a decimal value. MACRO-86 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<br>xxxQ | 735O (letter O)<br>412Q |
| Decimal | xxxxx<br>xxxxxD | 65535 (default)<br>1000D (when .RADIX changes input radix to nondecimal) |
| Hexadecimal | xxxxH | 0FFFFH (first digit must be 0-9) |
| ASCII | 'xx'<br>"xx" | 'OM' (more than two with DB only;<br>"OM" 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 (see Section 2.3, Symbols) 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
```

assuming FOO was defined as a constant symbol. For example:

```
FOO EQU 9
```

### 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 bits 0-7.

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-86 that an operand is in a different segment than specified in an ASSUME statement. (See the segment override operator in Section 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 |
| | PF | parity flag | IF interrupt-enable |
| 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).

### 3.2.3 Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct MACRO-86 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]<br><br>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.<br>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 combinations:

```
[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.

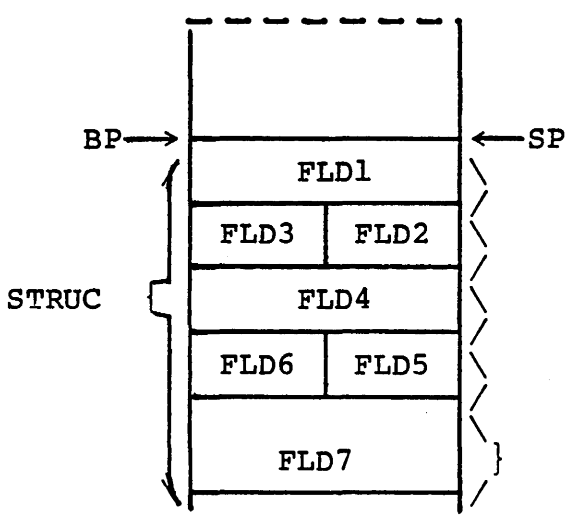
The period (.) must be included.

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 are 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.

## 3.3 OPERATORS

An operator may be one of four types: attribute, arithmetic, relational, or logical.

Attribute operators are used with operands to override their attributes, return the value of the attributes, or to isolate fields of Records.

Arithmetic, relational, and logical operators are used to combine or compare operands.

### 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 returning 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.

<u>Pointer</u> (PTR)

<attribute> PTR <expression>

The PTR operator overrides the type (BYTE, WORD, DWORD) <u>or</u> the distance (NEAR, FAR) of an operand.
Think "PTR" is optional when NEAR/FAR specified

<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-86 understands what attribute the expression is supposed to have. This is especially 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 structures. If the structure is defined as WORD but 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 WORD PTR [BX][SI]   superfluous in near call-indirect
MOV BYTE PTR ARRAY,(something)

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 label, a variable, or other memory operand).

The colon operator helps with forward references by telling the assembler to what a reference is relative (segment, group, or segment register).

MACRO-86 assumes that labels are addressable through the current CS register. MACRO-86 assumes that variables 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-86 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-86 can reach an operand through a nondefault segment register, it will use it, but the 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 labels used as targets for the JMP instruction. SHORT tells MACRO-86 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 location 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 LABEL 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 absolute 16-bit value or address expression.

LOW isolates the low 8 bits of an absolute 16-bit value or address expression.

Examples:

```
MOV AH,HIGH WORD_VALUE ;get byte with sign bit

MOV 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-86 have three attributes, you need to use 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.

Example:

```
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 segment 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 MS-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:

```
MOV BX,OFFSET FOO
```

If you use an ASSUME to GROUP, OFFSET will not automatically return the offset of a variable from the base address of the group. Rather, OFFSET will return the segment offset, unless you use the segment override operator (group-name version). If the variable GOB is defined in a segment placed in DGROUP, and you want the offset of GOB in the group, you need to enter a statment like:

```
MOV BX,OFFSET DGROUP:GOB
```

You must be sure that the GROUP directive precedes any reference to a group name, including it 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 variable 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]
```

## .TYPE

.TYPE <variable>

The .TYPE operator returns a byte that describes two characteristics of the <variable>: 1) the mode, and 2) whether it is External or not. The argument to .TYPE may be any expression (string, numeric, logical). If the expression is invalid, .TYPE returns zero.

The byte that is returned is configured as follows:

The lower two bits are the mode. If the lower two bits are:

| | |
|---|---|
| 0 | the mode is Absolute (constant) |
| 1 | the mode is Program Related |
| 2 | the mode is Data Related |

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is not External.

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

.TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow; for example, when conditional assembly is involved.

Example:

```
FOO     MACRO   X
        LOCAL   Z
Z       =   .TYPE X
IF      Z...
```

.TYPE tests the mode and type of X. Depending on the evaluation of X, the block of code beginning with IF Z... may be assembled or omitted.

## 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)

MOV 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 4.2.1, Memory Directives). A record may be up to 16 bits long. The record is defined by fields, which may be from one to 16 bits long. To isolate one of the three characteristics of a record field, 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 an 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.

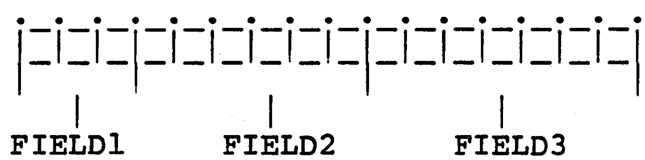
Shift-count - (Record fieldname)

<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.

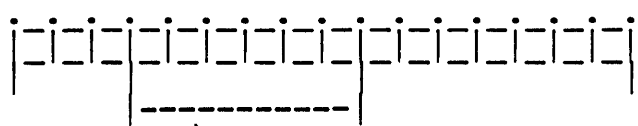
## WIDTH

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:

```
MOV CL,WIDTH FIELD2
```

The number of bits in FIELD2 is now in the count register.

### 3.3.2 Arithmetic Operators

Eight arithmetic operators provide the common mathematical functions (add, subtract, divide, multiply, modulo, negation), plus two shift operators.

The arithmetic operators are used to combine operands to form 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 absolute.<br><br>Example:<br><br>`MOV AX,100 MOD 17`<br><br>The value moved into AX will be 0FH (decimal 15). |
| SHR | Shift Right. SHR is followed by an integer which specifies the number of bit positions the value is to be right shifted.<br><br>Example:<br><br>`MOV AX,1100000B SHR 5`<br><br>The value moved into AX will be 11B (03). |
| SHL | Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be left shifted.<br><br>Example:<br><br>`MOV AX,0110B SHL 5`<br><br>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 if the left is also a nonconstant and in the same segment. |

### 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. |

### 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 anything 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 or if both are true. Returns false if 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 operators are true or if both operators are false. Both operands must be absolute values. |

### 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. *, /, MOD, 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

Contents

# CHAPTER 4

## ACTION: INSTRUCTIONS AND DIRECTIVES

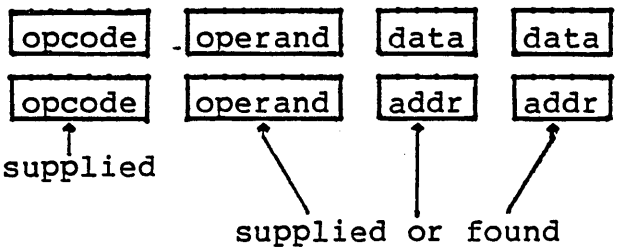
The action field contains either an 8086 instruction mnemonic or a MACRO-86 assembler directive.

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 improved 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.

## 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 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 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 texts. For now, the following texts 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.

Appendix C contains both an alphabetical listing and a grouped listing of the instruction mnemonics. The alphabetical listing shows the full name of the instruction. Following the alphabetical list is a list that groups the instruction mnemonics by the number and type of arguments they take. Within each group, the instruction mnemonics are arranged alphabetically.

## 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.

Appendix B contains a table of assembler directives, also grouped by function. Here below is an alphabetical list of all the directives that MACRO-86 supports:

| | | | |
|---|---|---|---|
| ASSUME | EVEN | IRPC | .RADIX |
| | EXITM | | RECORD |
| COMMENT | EXTERN | LABEL | REPT |
| .CREF | | .LALL | |
| | 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 | |

### 4.2.1 Memory Directives

<u>ASSUME</u>

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 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 be prefixed by a segment register. For example, DS:FOO instead of simply FOO.

Example:

```
ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHING
```

<u>COMMENT</u>

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 your 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 BYTE**
**DEFINE 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 colon, it becomes a NEAR label instead of a variable. (See also, Section 2.1, Labels, and Section 2.2, Variables.)

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 data block. 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  ?                   ;initialize with
                                    ;indeterminate 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  100 DUP(5 DUP(4),7)
            ;100 copies of bytes with values 4,4,4,4,4,7
NEW_PAGE    DB  0CH                 ;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(?,?)         ;initalized 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 MS-LINK for that program or module.

Example:

```
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 alias 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.

Example:

```
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.

Example:

```
FOO  =    5          ;the same as FOO EQU 5
FOO  EQU  6;         ;error, FOO cannot be
                     ;redefined 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 directive 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.

Example:

```
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 within 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 then use either:

ASSUME <seg-reg>:SEG <name>

or an explicit segment prefix.

NOTE

If a mistake is made and the symbol is not in the segment, MS-LINK will take the offset relative to the given segment, if possible. If the real segment is more than 64K bytes away from the reference, MS-LINK may find the definition. If the real segment is more than 64K bytes away, MS-LINK will fail to make the link between the reference and the definition and will not return an error message. less than?

Example:

<table>
<tr><th>In Same Segment:</th><th>In Another Segment:</th></tr>
<tr><td>

In Module 1:

```
CSEG    SEGMENT
        PUBLIC TAGN
        .
        .
        .
TAGN:
        .
        .
        .
CSEG    ENDS
```

In Module 2:

```
CSEG    SEGMENT
        EXTRN TAGN:NEAR
        .
        .
        .
        JMP TAGN
CSEG    ENDS
```

</td><td>

In Module 1:

```
CSEGA   SEGMENT
        PUBLIC TAGF
        .
        .
        .
TAGF:
        .
        .
        .
CSEGA   ENDS
```

In Module 2:

```
        EXTRN TAGF:FAR
CSEGB   SEGMENT
        .
        .
        .
        JMP TAGF
CSEGB   ENDS
```

</td></tr>
</table>

## 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 MS-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 MS-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 MS-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 statements:

   ```
   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
```

<u>INCLUDE</u>

multiple EOFs used by WordStar cause "extra character on line" warning. Solution

INCLUDE <filename> is to "trim" or "scrub" the include-file.

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 with small systems because of the amount of memory that 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-86 listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See Section 5.4, Formats of Listings and Symbol Tables, for a description of listing file formats.

Example:

```
INCLUDE ENTRY
INCLUDE B:RECORD.TST
```

## LABEL

<name> LABEL <type>

By using LABEL to define a <name>, you cause the assembler 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 instuctions 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,ARRAY[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. It 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 way. It is easier to give the array a second name than to remember to use PTR.

NAME

NAME <module-name>

<module-name> must not be a reserved word. The module name may be any length, but MACRO-86 uses only the first six characters and truncates the rest.

The module name is passed to MS-LINK, but otherwise has no significance for the assembler. MACRO-86 does check if more than one module name has been declared.

Every module has a name. MACRO-86 derives the module name from:

1. a valid NAME directive statement

2. If the module does not contain a NAME statement, MACRO-86 uses the first six characters of a TITLE directive statement. The first six characters must be legal as a name.

3. First 6 of file-name?

Example:

```
NAME CURSOR
```

## ORG

ORG <exp>

The location counter is set to the value of <exp>, and the assembler assigns generated 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 absolute 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)
        ENDIF
```

See Section 4.2.2, Conditional Directives, for an explanation of conditional assembly.

Note $ = (IP)

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.

Each 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.

Example:

```
             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 segment 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 want 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 passed to MS-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.

Example:

```
         PUBLIC  GETINFO
GETINFO  PROC    FAR
         PUSH    BP           ;save caller's register
         MOV     BP,SP        ;get address parameters
                              ;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
```

## .RADIX

.RADIX <exp>

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. Succesively 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>|
```

Total 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
 . . . . . . . . . . . . . . . . .
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|     not    |<HIGH->|<MID>|<LOW>|
   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='Q'
```

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 value applies (the value given in the original record defintion), 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.

Example:

```
FOO   RECORD  HIGH:5,MID:3,LOW:3
      .
      .
      .
BAX   FOO     <>  ;leave undeterminate Hhere
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, an MS-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 MS-LINK manual).

<class> name is used to group segments at link time.

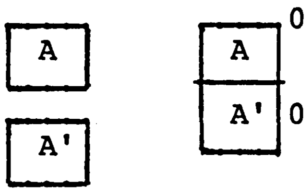
All three operands are passed to MS-LINK.

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 alignment 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)
```

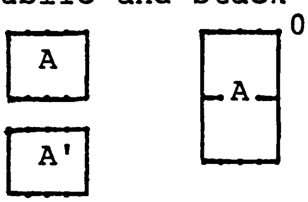
The combine type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or Private)



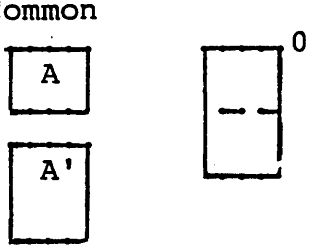

Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

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. MS-LINK requires at least one 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.

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 MS-LINK. MS-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 fixed offsets within fixed areas of storage, such as ROM or the vector space in low memory.

NOTE

This restriction is imposed by MS-LINK and MS-DOS.

Class names must be enclosed in quotation marks. Class names may be any legal name. Refer to Microsoft's MS-LINK Linker Utility manual 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  |      |A   SEGMENT
          .      |      |      .
          .      |      |      .
          .      |      |      .
    B   SEGMENT  |      |A   ENDS
          .      |      |B   SEGMENT
          .      |----> |      .
          .      |      |      .
    B   ENDS     |      |      .
          .      |      |B   ENDS
          .      |      |A   SEGMENT
          .      |      |      .
    A   ENDS     |      |      .
                        |      .
                        |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
       .        ;MS-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 RECORDs.

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 overridden 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.

Example:

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
S       ENDS
```

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
```

### 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. 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>

and

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.

IFNB (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 4.2.3, Macro Directives, for a full explanation.

IFIDN <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

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 macro call specifies two identical parameters to replace the dummys, the block will be assembled. (IFDIF is the opposite case.) Refer to Section 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 call. If the macro call specifies two different parameters to replace the dummys, the block will be assembled. (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.

### 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-86 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 macro'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-86 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-86 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-86 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-86 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 ignored. 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 block 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.

EXAMPLE:

```
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
```

Notice that MACRO-86 has substituted LABEL names in the form ??nnnn for the instances of the dummy symbols.

## 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 orignal 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 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.

Example:

```
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 passed 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 performed 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.

<u>Repeat</u>

```
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.

EXAMPLE:

```
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-86, unlike MACRO-80, 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>            IRP     Z,<1,2,3>
X&&Z    DB      Z            X&Z     DB      Z
        ENDM                         ENDM
        ENDM                         ENDM
```

When called, for example, by FOO BAZ, the expansion would be (correcly 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-86 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 4.2.4, Listing Directives, below). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments inside 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 *
```

## 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 page 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-86 encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until MACRO-86 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 unless 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.

Example:

```
.XCREF CURSOR,FOO,GOO,BAZ,ZOO
        ;these variables will not be
        ;in the listing or cross reference file
```

## Contents

# CHAPTER 5

# ASSEMBLING A MACRO-86 SOURCE FILE

Assembling with MACRO-86 requires two types of commands: a command to invoke MACRO-86 and answers to command prompts. In addition, four switches control alternate MACRO-86 features. Usually, the user will enter all the commands to MACRO-86 on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in a Batch File (see the MS-DOS manual for Batch File instructions). Some Command Characters are provided to assist the user while entering assembler commands.

## 5.1 INVOKING MACRO-86

MACRO-86 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-86.

Summary of Methods to invoke MACRO-86

| | |
|---|---|
| Method 1 | MASM |
| Method 2 | MASM <source>,<object>,<listing>,<cross-ref>[/switch...] |

### 5.1.1 Method 1: MASM

Enter:

```
MASM
```

MACRO-86 will be loaded into memory. Then, MACRO-86 returns a series of four text prompts that appear one at a time. The user answers the prompts as commands to MACRO-86 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-86 defaults to not performing the function described for the switches in the chart below.

The command prompts are summarized here and described in more detail in Section 2.2, Command Prompts. Following the summary of prompts is a summary of switches, which are described in more detail in Section 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 MS-CREF to create a cross reference listing). (default: no cross reference file) |

| SWITCH | ACTION |
|---|---|
| /D | Produce 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-86 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<CR>
Object filename [FUN.OBJ]:  ;<CR>
```

The remaining prompts will not appear, and MACRO-86 will use the default values (including no listing file and no cross reference file).

To achieve exactly the same result, you could alternatively enter:

```
Source filename [.ASM]:  FUN;<CR>
```

This response produces the same files as the previous example.

Control-C Use Control-C 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 Control-C to exit MACRO-86 then reinvoke MACRO-86 and start over. If the error has been typed and not entered, you may delete the erroneous characters, but for that line only.

### 5.1.2 Method 2: MASM <filenames>[/switches]

Enter:

```
MASM <source>,<object>,<listing>,<cross-ref>[/switch...]
```

MACRO-86 will be loaded into memory. Then MACRO-86 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 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-86 to be loaded, then causes the source file FUN.ASM to be assembled. MACRO-86 then outputs the relocatable object code 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 swithces are given but no filename, the switches are ignored.

## 5.2 MACRO-86 COMMAND PROMPTS

MACRO-86 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, MACRO-86 begins assembly automatically without further command. When assembly is finished, MACRO-86 exits to the operating system. When the operating system prompt is displayed, MACRO-86 has finished successfully. If the assembly is unsuccessful, MACRO-86 returns the appropriate error message.

MACRO-86 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 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-86 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-86 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-86 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 MS-CREF Cross Reference Facility. MS-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. MS-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.

## 5.3 MACRO-86 COMMAND SWITCHES

The three switches control alternate assembler functions. Switches must be entered at the end of a prompt response, regardless of which method is used to invoke MACRO-86. 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 (/). You may not enter only a switch as a response 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-86 to create a source listing (enter a filename in response to the Source listing command prompt). |
| /O | Output the listing file 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 you 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 generated code, use the /X switch to suppress the blocks that evaluate false from your listing.<br><br>The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives. |
| /I | 4.Ø recognizes extended search path for INCLUDEs |

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-86 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 Chapter 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 | ON |
| . | . | . |
| . | . | . |
| . | . | . |
| .TFCOND | OFF | ON |
| . | . | . |
| . | . | . |
| . | . | . |
| .TFCOND | ON | OFF |
| . | . | . |
| . | . | . |
| . | . | . |
| .SFCOND | OFF | OFF |
| . | . | . |
| . | . | . |
| . | . | . |
| .TFCOND | OFF | ON |
| .TFCOND | ON | OFF |
| . | . | . |
| . | . | . |
| . | . | . |
| .TFCOND | OFF | ON |

## 5.4 FORMATS OF LISTINGS AND SYMBOL TABLES

The source listing produced by MACRO-86 (created when you specify a filename in response to the Source listing prompt) is divided into two parts.

The first part of the listing shows:

> the line number for each line of the source file, if a cross reference file is also being created.
> the offset of each source line that generates code.
> the code generated by each source line.
> a plus sign (+), if the code came from a macro or a letter C, if the code came from an INCLUDE file.
> the source statement line.

The second part of the listing shows:

> Macros - name and length in bytes
> Structures and records - name, width and fields
> Segments and groups - name, size, align, combine, and class
> Symbols - name, type, value, and attributes
> the number of warning errors and severe errors

### 5.4.1 Program Listing

The program portion of the listing is essentially your source program fle with the line numbers, offsets, generated code, and (where applicable) a plus sign to indicate that the source statements are part of a macro block a letter C to indicate that the source statements are from a file input by the INCLUDE directive.

If any errors occur during assembly, the error message will be printed directly below the statement where the error occurred.

On the next page is part of a listing file, with notes explaining what the various entries represent.

The comments have been moved down one line because of format restrictions. If you print your listing on 132 column paper, the comments shown here would easily fit on the same line as the rest of the statement.

Explanatory notes are spliced into the listing at points of special interest.

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:DI until CX=0
```

| | |
|---|---|
| [ xx ] | = DUP expression;xx is the value in parentheses 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 |

```
Microsoft MACRO-86 MACRO Assembler    1-Dec-81              PAGE 1-3

ENTX     PASCAL entry for initializing programs


                                                       ;
0000                              STACK      SEGMENT WORD STACK   'STACK'
= 0000                            HEAPbeg    EQU     THIS BYTE
      Indicates EQU or = directive

                                                        ;Base of heap before init
done
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  BEGXQQ ;Main entry

                                  ;
0000                              BEGXQQ     PROC    FAR
0000     B8   ---- R                         MOV     AX,DGROUP
                                                     ;get assumed data segment
value
0003     8E D8                               MOV     DS,AX ;Set DS seg

0005     8C 06 0022 R                        MOV     CESXQQ,ES
          generated                name      action          comment
            code                                    expression
offset
                                                      .
                                                      .
                                                      .
000C     26: 8B 1E 0002                      MOV     BX,ES:2   ;Highest paragraph
                          segment override
```

```
Microsoft MACRO-86 MACRO Assembler    1-Dec-81              PAGE 1-4

ENTX    PASCAL entry for initializing programs


0011  2B D8                                  SUB     BX,AX  ;Get # paras for DS
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  D1 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     directive      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
                     examples in this listing.)

006E                         BEGXQQ          ENDP

                                                     .
                                                     .
                                                     .
007E                         MAINSTARTUP             ENDS

0000                         ENTXCM          SEGMENT WORD 'CODE'
                                             ASSUME  CS:ENTXCM
                                             PUBLIC  ENDXQQ,DOSXQQ
```

```
Microsoft MACRO-86 MACRO Assembler    1-Dec-81              PAGE 1-5

ENTX     PASCAL entry for initializing programs

0000                          STARTmain  PROC     FAR ;This code remains
0000   9A 0000 ---- E                    CALL     ENTGQQ
                                                  ;call main program
                              ;
0005                          ENDXQQ     LABEL    FAR
                                                    ;termination entry point
0005   9A 0000 ---- E                    CALL     ENDOQQ
                                                    ;user system termination
000A   9A 0000 ---- E                    CALL     ENDYQQ
                                                    ;close all open files
000F   9A 0000 ---- E                    CALL     ENDUQQ
                                                    ;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

00 2E 0020 R                             JMP      DWORD PTR DOSOFF
                                                              ;return to DOS
001E                          STARTmain  ENDP

                                                  .
                                                  .
                                                  .
0037                          ENTXCM     ENDS
                                         END      BEGXQQ
```

Differences Between Pass 1 Listing and Pass 2 Listing

If you give the /D switch when you run MACRO-86 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 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 produces:

```
0017  7E 00                      JLE     SMLSTK  ;No, use what we have
 E r r o r   ---     9:Symbol 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
001C                 SMLSTK:     REPT    4
```

Notice that the JLE instructions code now contain 03 instead of 00; a jump of 3 bytes.

The same amount of code was produced during both passes, so there was no phase error. The only difference is one of content instead of size, in this case.

### 5.4.2 Symbol Table Format

The symbol table portion of 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 MS-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.

```
Microsoft MACRO-86 MACRO
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   0004
  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 blocks
                                        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 blocks long or (5 X 32 bytes =) 160 bytes long.

Structures and records:

<u>Example for Structures</u>

```
                                       This line applies to Structure Names
                                                  (begin in column 1)

               Name      Width  # fields
                         Shift  Width  Mask  Initial   This line
                                                       for fields
PARMLIST . . . . . .     001C   0004                   of Records
  BUFSIZE. . . . . .     0000                          (indented).
  NAMESIZE . . . . .     0001
  NAMETEXT . . . . .     0002
  TERMINATOR . . . .     001B
                                                     Number of fields in
  field names of        Offset of field                Structure
  PARMLIST Structure     into structure
                                                 The number of bytes
                                              wide of Structure
```

<u>Example for Records</u>

```
               Name      Width  # fields
                         Shift  Width  Mask  Initial    This line is
                                                           for fields
                                                           of records.

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 in Record   count   bits in 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:

<u>Width</u> (upper line) shows the number of bytes your Structure occupies in memory.
<u># fields</u> 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 fields 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:

```
 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0  <---MASK = 07F8
.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
15       11|10             4|3   0
           |----------------|---->|
              |               shift count = 0003
              |
          WIDTH = 0008
```

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:

```
 0 0 0 0 0|1 0 0 0|0 0 0 0|0 0 0  <---Initial = 0400
.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
          |               |
          | initial = 80H |
               80H = 128 decimal
```

Segments and groups:

```
                   Name         Size   align   combine class
                                                 ----called Private
                                               / in MS-LINK manual
AAAXQQ . . . . . . . .          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
                                segment
```

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 a part of a group or not:

<u>Size</u> is the number of bytes the Segment occupies.

<u>Align</u> 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)
```

<u>Combine</u> describes how MS-LINK Linker Utility will combine the various segments. (See MS-LINK Linker Utility Manual for a full description.)

<u>Class</u> is the class name under which MS-LINK will combine segments in memory. (See MS-LINK

Linker Utility Manual for a full description.)

```
Symbols:

             Name             Type     Value   Attr

FOO. . . . . . . . . .        Number   0005
FOO1 . . . . . . . . .        Text     1.234
FOO2 . . . . . . . . .        Number   0008          all formed by
FOO3 . . . . . . . . .        Alias    FOO           EQU or =
FOO4 . . . . . . . . .        Text     5[BP][DI]     directive
FOO5 . . . . . . . . .        Opcode
```

```
Symbols:

             Name             Type     Value   Attr

BEGHQQ . . . . . . . .        L WORD   0012    DATA    Global
BEGOQQ . . . . . . . .        L FAR    0000      External
BEGXQQ . . . . . . . .        F PROC   0000    MAIN_STARTUP  Global Length =006E
CESXQQ . . . . . . . .        L WORD   0022    DATA    Global
CLNEQQ . . . . . . . .        L WORD   0002    DATA    Global              length
CRCXQQ . . . . . . . .        L WORD   001C    DATA    Global              of PROC
CRDXQQ . . . . . . . .        L WORD   001E    DATA    Global
CSXEQQ . . . . . . . .        L WORD   0000    DATA    Global
CURHQQ . . . . . . . .        L WORD   0014    DATA    Global
DOSOFF . . . . . . . .        L WORD   0020    DATA
DOSXQQ . . . . . . . .        F PROC   001E    ENTXCM  Global Length =0019
ENDHQQ . . . . . . . .        L WORD   0016    DATA    Global
ENDOQQ . . . . . . . .        L FAR    0000            External
ENDUQQ . . . . . . . .        L FAR    0000            External
ENDXQQ . . . . . . . .        L FAR    0005    ENTXCM  Global
ENDYQQ . . . . . . . .        L FAR    0000            External
ENTGQQ . . . . . . . .        L FAR    0000            External
FREXQQ . . . . . . . .        F PROC   006E    MAIN_STARTUP  Global Length =0010
HDRFQQ . . . . . . . .        L WORD   0006    DATA    Global
HDRVQQ . . . . . . . .        L WORD   0008    DATA    Global
HEAPBEG. . . . . . . .        BYTE     0000    STACK            EQU statements
HEAPLOW. . . . . . . .        BYTE     0000    HEAP             showing segment
INIUQQ . . . . . . . .        L FAR    0000            External
PNUXQQ . . . . . . . .        L WORD   0004    DATA    Global
RECEQQ . . . . . . . .        L WORD   0010    DATA    Global
REFEQQ . . . . . . . .        L WORD   000C    DATA    Global
REPEQQ . . . . . . . .        L WORD   000E    DATA    Global
RESEQQ . . . . . . . .        L WORD   000A    DATA    Global
SKTOP. . . . . . . . .        BYTE     0014    STACK
SMLSTK . . . . . . . .        L NEAR   001C    MAIN_STARTUP
STARTMAIN. . . . . . .        F PROC   0000    ENTXCM  Length =001E
STKBQQ . . . . . . . .        L WORD   0018    DATA    Global
STKHQQ . . . . . . . .        L WORD   001A    DATA    Global

                              If MACRO-86 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 entry 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 <u>Number</u>, <u>Opcode</u>, <u>Alias</u>, or <u>Text</u>, 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 simple text.

Each of the four types shows a value as follows:

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 represents. "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
```

Contents

# CHAPTER 6

# MACRO-86 MESSAGES

Most of the messages output by MACRO-86 are error messages. The nonerror messages output by MACRO-86 are the banner MACRO-86 displays when first invoked, the command prompt 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.

## 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)
```

(your disk operating system's prompt)

## 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 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 to define 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 DUP's 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 example, 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.

Improper use of segment reg (Code 61)

Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

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)

Used 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 45)

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 exected. 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 sepcified.

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, QW, 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 instrucion 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 two 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 (see Section x.x, Switches).

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 12)

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.

I/O Handler Errors

These error messages are generated by the I/O handlers. These messages appear in a different format from the Assembler Errors:

```
MASM Error -- error-message-text
  in:  filename
```

The filename is the name of the file being handled when the error occurred.

The error-message-text is one of the following messages:

Data format (Code 114)

Device full (Code 108)

Device name (Code 102)

Device offline (Code 105)

File in use (Code 112)

File name (Code 107)

File not found (Code 110)

File not open (Code 113)

File system (Code 104)

Hard data (Code 101)

Line too long (Code 115)

Lost file (Code 106)

Operation (Code 103)

Protected file (Code 111)

Unknown device (Code 109)

Runtime Errors

These messages may be displayed as your assembled program is being executed.

Internal Error

Usually caused by an arithmetic check. If it occurs, notify Microsoft, Inc.

Out of Memory

This message has no corresponding number. Either the source was too big or too many labels are in the symbol table.

Numerical Order List of Error Messages

| Code | Message |
|---|---|
| 0 | Block nesting error |
| 1 | Extra characters on line |
| 2 | Register already defined |
| 3 | Unknown symbol type |
| 4 | Redefinition of symbol |
| 5 | Symbol is multi-defined |
| 6 | Phase error between passes |
| 7 | Already had ELSE clause |
| 8 | Not in conditional block |
| 9 | Symbol not defined |
| 10 | Syntax error |
| 11 | Type illegal in context |
| 12 | Should have been group name |
| 13 | Must be declared in pass 1 |
| 14 | Symbol type usage illegal |
| 15 | Symbol already different kind |
| 16 | Symbol is reserved word |
| 17 | Forward reference is illegal |
| 18 | Must be register |
| 19 | Wrong type of register |
| 20 | Must be segment or group |
| 21 | Symbol has no segment |
| 22 | Must be symbol type |
| 23 | Already defined locally |
| 24 | Segment parameters are changed |
| 25 | Not proper align/combine type |
| 26 | Reference to mult defined |
| 27 | Operand was expected |
| 28 | Operator was expected |
| 29 | Division by 0 or overflow |
| 30 | Shift count is negative |
| 31 | Operand types must match |
| 32 | Illegal use of external |
| 33 | Must be record field name |
| 34 | Must be record or field name |
| 35 | Operand must have size |
| 36 | Must be var, label or constant |
| 37 | Must be structure field name |
| 38 | Left operand must have segment |
| 39 | One operand must be const |
| 40 | Operands must be same or 1 abs |
| 41 | Normal type operand expected |
| 42 | Constant was expected |
| 43 | Operand must have segment |
| 44 | Must be associated with data |
| 45 | Must be associated with code |
| 46 | Already have base register |
| 47 | Already have index register |
| 48 | Must be index or base register |
| 49 | Illegal use of register |
| 50 | Value is out of range |

| | |
|---|---|
| 51 | Operand not in IP segment |
| 52 | Improper operand type |
| 53 | Relative jump out of range |
| 54 | Index displ. must be constant |
| 55 | Illegal register value |
| 56 | No immediate mode |
| 57 | Illegal size for item |
| 58 | Byte register is illegal |
| 59 | CS register illegal usage |
| 60 | Must be AX or AL |
| 61 | Improper use of segment reg |
| 62 | No or unreachable CS |
| 63 | Operand combination illegal |
| 64 | Near JMP/CALL to different CS |
| 65 | Label can't have seg. override |
| 66 | Must have opcode after prefix |
| 67 | Can't override ES segment |
| 68 | Can't reach with segment reg |
| 69 | Must be in segment block |
| 70 | Can't use EVEN on BYTE segment |
| 71 | Forward needs override |
| 72 | Illegal value for DUP count |
| 73 | Symbol already external |
| 74 | DUP is too large for linker |
| 75 | Usage of ? inate) bad (Code 75) |
| 76 | More values than defined with |
| 77 | Only initialize list legal |
| 78 | Directive illegal in STRUC |
| 79 | Override with DUP is illegal |
| 80 | Field cannot be overridden |
| 81 | Override is of wrong type |
| 82 | Register can't be forward ref |
| 83 | Circular chain of EQU aliases |
| 101 | Hard data |
| 102 | Device name |
| 103 | Operation |
| 104 | File system |
| 105 | Device offline |
| 106 | Lost file |
| 107 | File name |
| 108 | Device full |
| 109 | Unknown device |
| 110 | File not found |
| 111 | Protected file |
| 112 | File in use |
| 113 | File not open |
| 114 | Data format |
| 115 | Line too long |

INDEX

# MS-LINK linker utility

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

## LIMITED WARRANTY

**MICROSOFT, Inc.** shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

**THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.**

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.



8407B-100-01

System Requirements

The MS-LINK Linker Utility requires:

49K bytes of memory minimum:
- 40K bytes for code and data
- 10K bytes for run space

1 disk drive

1 disk drive if and only if output is sent to the same physical diskette from which the input was taken. MS-LINK does not allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

# INTRODUCTION

## Features and Benefits of MS-LINK

MS-LINK is a relocatable linker designed to link together separately produced modules of 8086 object code. The object modules must be 8086 files only.

MS-LINK is user-friendly. For all the necessary and optional commands, MS-LINK prompts the user. The user's answers to the prompts are the commands for MS-LINK.

The output file from MS-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.

MS-LINK uses a dictionary-indexed library search method, which substantially reduces link time for sessions involving library searches.

MS-LINK is capable of linking files totaling 384K bytes.

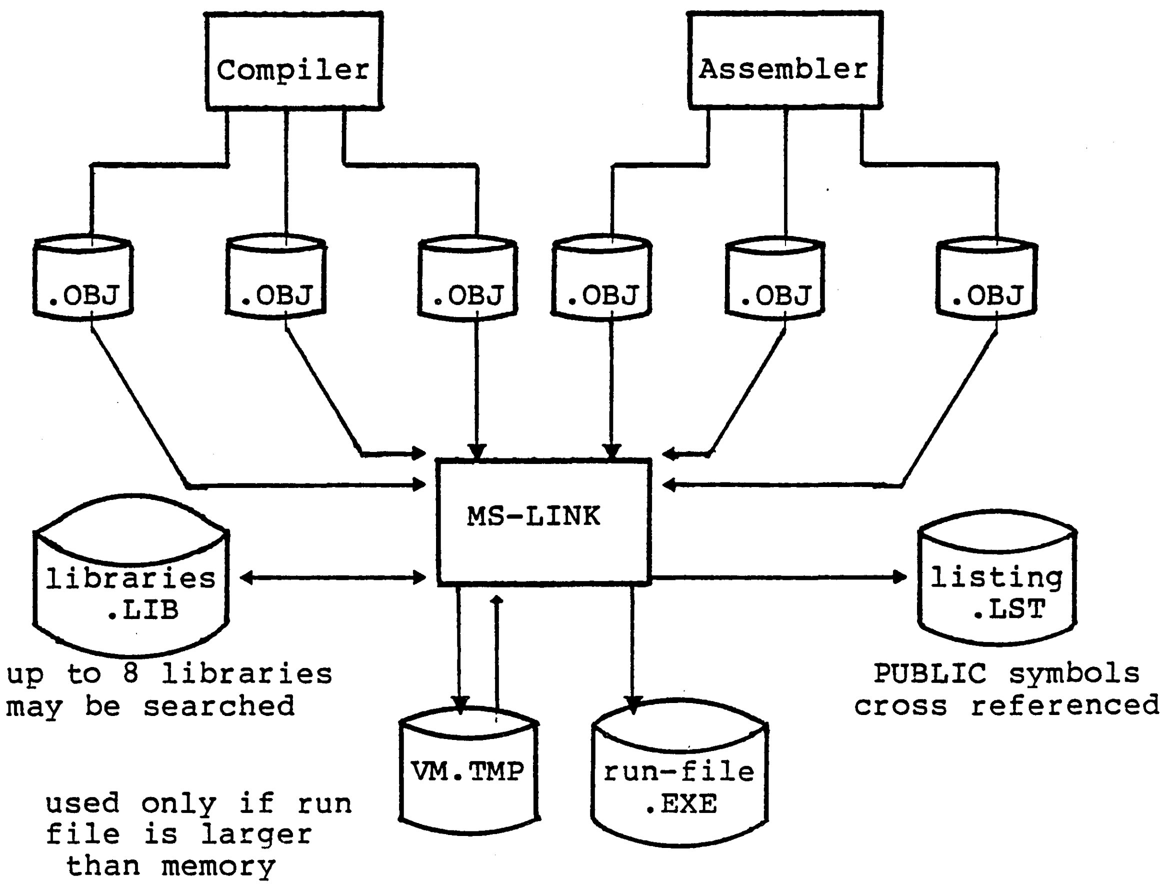
Overview of MS-LINK Operation

MS-LINK combines several object modules into one relocatable load module, or Run file.

As it combines modules, MS-LINK resolves external references between object modules and can search multiple library files for definitions for any external references left unresolved.

MS-LINK also produces a list file that shows external references resolved and any error messages.

MS-LINK uses available memory as much as possible. When available memory is exhausted, MS-LINK then creates a disk file and becomes a virtual linker.



Definitions

Three terms will appear in some of the error messages listed in Chapter 2. These terms describe the underlying functioning of MS-LINK. An understanding of the concepts that define these terms provides a basic understanding of the way MS-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. MS-LINK checks to see that the object modules of a Group meet the 64K byte constraint.

3. Class

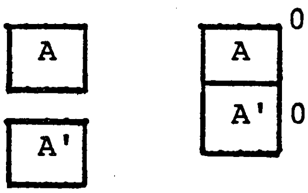
   A Class is a collection of Segments. The naming of Segments to a Class controls the order and relative placement of Segments in memory. The Class name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler. 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 MS-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 MS-LINK. Classes may be loaded across 64K byte boundaries. The Classes will be divided into Groups for addressing.

## How MS-LINK Combines and Arranges Segments

MS-LINK works with four combine types, which are declared in the source module for the assembler or compiler: private, public, stack, and common. (The memory combine type available in Microsoft's MACRO-86 is treated the same as public. MS-LINK does not automatically place memory combine type as the highest segments.)

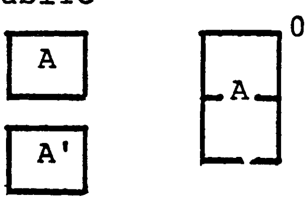
MS-LINK combines segments for these combine types as follows:

Private



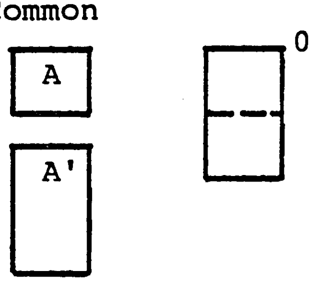

Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

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.)

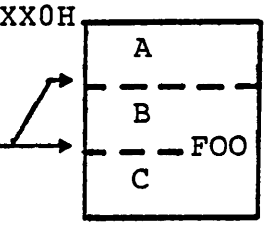
Common




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.

DS:DGROUP → XXXX0H ... 0 -- relative offset




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. MS-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 MS-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  '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.

## Files That MS-LINK Uses

MS-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 MS-LINK file specifications is:

drv:filename.ext

where: drv: is the drive designation. Permissible drive designations for MS-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 an 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, MS-LINK recognizes by default:

| File | Default Extension |
|---|---|
| Object | .OBJ |
| Library | .LIB |

## Output Files

MS-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) |

VM.TMP File

MS-LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, MS-LINK creates a temporary file and names it VM.TMP. If MS-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 MS-LINK is unpredictable, and MS-LINK might return the error message:

    Unexpected end of file on VM.TMP

MS-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 MS-LINK requires the VM.TMP file, MS-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. |

# CHAPTER 1

# RUNNING MS-LINK

Running MS-LINK requires two types of commands: a command to invoke MS-LINK and answers to command prompts. In addition, six switches control alternate MS-LINK features. Usually, the user will enter all the commands to MS-LINK on the terminal keyboard. As an option, answers to the 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.

## 1.1 INVOKING MS-LINK

MS-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 MS-LINK. By the third method, the user creates a Response File that contains all the necessary commands.

Summary of Methods to invoke MS-LINK

| | |
|---|---|
| Method 1 | LINK |
| Method 2 | LINK <filenames>[/switches] |
| Method 3 | LINK @<filespec> |

### 1.1.1 Method 1: LINK

Enter:

```
LINK
```

MS-LINK will be loaded into memory. Then, MS-LINK returns a series of four text prompts that appear one at a time. The user answers the prompts as commands to MS-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, MS-LINK defaults to not performing the function described for the switches in the chart below.

The command prompts are summarized here and described in more detail in Section 2.2, COMMAND PROMPTS. Following the summary of prompts is a summary of switches, which are described in more detail in Section 2.3, 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 carriage return key. |
| /STACK:<number> | Set fixed stack size in Run file. |

Command Characters

MS-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 which may also be very long), enter an 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, MS-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+<CR>
Object Modules [.OBJ]:
FOO+FLIPFLOP+JUNQUE+<CR>
Object Modules [.OBJ]: CORSAIR<CR>
```

; 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 carriage return.

Example:

```
Object Modules [.OBJ]:  FUN TEXT TABLE CARE<CR>
Run Module [FUN.EXE]:  ;<CR>
```

The remaining prompts will not appear, and MS-LINK will use the default values (including FUN.MAP for the List File).

Control-C Use Control-C 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 Control-C to exit MS-LINK then reinvoke MS-LINK and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

### 1.1.2 Method 2: LINK <filenames>[/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: <u>object list</u> is a list of object modules, separated by plus signs

<u>runfile</u> is the name of the file to receive the executable output

<u>listfile</u> is the name of the file to receive the listing

<u>lib list</u> is a list of library modules to be searched

<u>/switch</u> 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 comma without spaces in between (see the example below).

Example

```
LINK FUN+TEXT+TABLE+CARE/P/M,,FUNLIST,COBLIB.LIB
```

This example causes MS-LINK to be loaded, then causes the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ to be loaded. MS-LINK then pauses (caused by the /P switch). When the user presses any key, MS-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.

### 1.1.3 Method 3: LINK @<filespec>

Enter:

LINK @<filespec>

where: filespec is the name of a Response File. A Response File contains answers to the MS-LINK prompts (shown under method 1 for invoking), and may also contain any of the switches. Method 3 permits the user to conduct the MS-LINK session without interactive (direct) user responses to the MS-LINK prompts.

IMPORTANT

> Before using method 3 to invoke MS-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 MS-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, MS-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, MS-LINK continues the link session.

Example:

```
FUN TEXT TABLE CARE
/PAUSE/MAP
FUNLIST
COBLIB.LIB
```

This Response File will cause MS-LINK to load the four Object modules. MS-LINK will pause before creating and producing a public symbol map to permit the user to swap diskettes (see discussion under /PAUSE in Section 2.3, Switches, before using this feature). When the user presses any key, the output files will be named FUN.EXE and FUNLIST.MAP, MS-LINK will search the library file COBLIB.LIB, and will use the default settings for the flags.

## 1.2 COMMAND PROMPTS

MS-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, MS-LINK begins linking automatically without further command. When the link session is finished, MS-LINK exits to the operating system. When the operating system prompt is displayed, MS-LINK has finished successfully. If the link session is unsuccessful, MS-LINK returns the appropriate error message.

MS-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.

<u>Object Modules [.OBJ]:</u>

Enter a list of the object modules to be linked. MS-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 omitted.

Modules must be separated by plus signs (+).

Remember that MS-LINK loads Segments into Classes in the order encountered (see Section 1.2, 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, MS-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 MS-LINK to create the Run file PAYROLL.EXE on drive B:. Also, MS-LINK will pause, which allows the user to insert a new diskette to receive the Run file.

List File [Run-Filename.MAP]:
The List file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in 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. MS-LINK assumes by default that the filename extension is .LIB for library files.

Library filenames must be separated by blank spaces or plus signs (+).

MS-LINK searches the library files in the order listed to resolve external references. When it finds the module that defines the external symbol, MS-LINK processes the module as another object module.

If MS-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).

MS-LINK does not search within each library file sequentially. MS-LINK uses a method called dictionary indexed library search. This means that MS-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.

## 1.3 SWITCHES

The six switches control alternate linker functions. Switches must be entered at the end of a prompt response, regardless of which method is used to invoke MS-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 transpositions are allowed. For example:

| Legal | Illegal |
|---|---|
| /D | /DSL |
| /DS | /DAL |
| /DSA | /DLC |
| /DSALLOCA | /DSALLOCT |

/DSALLOCATE

Use of the /DSALLOCATE switch directs MS-LINK to load all data (DGroup) at the high end of the Data Segment. Otherwise, MS-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 MS-LINK to place the Run image as high as possible in memory. Otherwise, MS-LINK places the Run file as low as possible.

IMPORTANT

> Do not use the /HIGH switch with Pascal or FORTRAN programs.

/LINENUMBERS

Use of the /LINENUMBERS switch directs MS-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, MS-LINK cannot include line numbers.

/MAP

/MAP directs MS-LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, MS-LINK will list only errors (which includes undefined globals).

The symbols are listed alphabetically. For each symbol, MS-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 MS-LINK to pause in the link session when the switch is encountered. Normally, MS-LINK performs the linking session without stop from beginning to end. This allows the user to swap the diskettes before MS-LINK outputs the Run (.EXE) file.

When MS-LINK encounters the /PAUSE switch, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

MS-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, MS-LINK calculates the necessary stack size automatically.

If a value from 1 to 511 is entered, MS-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, MS-LINK will return a WARNING: NO STACK STATEMENT error message.

# CHAPTER 2

## ERROR MESSAGES

All errors cause the link session to abort. Therefore, after the cause is found and corrected, MS-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: MS-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: MS-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 $\leq$ 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 MS-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 MS-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

INDEX

# MS-LIB library manager

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

## LIMITED WARRANTY

**MICROSOFT, Inc.** shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

**THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.**

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.



8407C-100-00

## System Requirements

The MS-LIB Library Manager requires:

38K bytes of memory minimum:
- 28K bytes for code
- 10K bytes for run space

1 disk drive

1 disk drive if and only if output is sent to the same physical diskette from which the input was taken. None of the utility programs in this package allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

# INTRODUCTION

## Features and Benefits

MS-LIB creates and modifies library files that are used with Microsoft's MS-LINK Linker Utility. MS-LIB can add object files to a library, delete modules from a library, or extract modules from a library and place the extracted modules into separate object files.

MS-LIB provides a means of creating either general or special libraries for a variety of programs or for specific programs only. With MS-LIB you can create a library for a language compiler, or you can create a library for one program only, which would permit very fast linking and possibly more efficient execution.

You can modify individual modules within a library by extracting the modules, making changes, then adding the modules to the library again. You can also replace an existing module with a different module or with a new version of an existing module.

The command scanner in MS-LIB is the same as the one used in Microsoft's MS-LINK, MS-Pascal, MS-FORTRAN, and other 16-bit Microsoft products. If you have used any of these products, using MS-LIB is familiar to you. Command syntax is straightforward, and MS-LIB prompts you for any of the commands it needs that you have not supplied. There are no surprises in the user interface.

## Overview of MS-LIB Operation

MS-LIB performs two basic actions: it deletes modules from a library file, and it changes object files into modules and appends them to a library file. These two actions underlie five library manager functions:

delete a module

extract a module and place it in a separate object file

append an object file as a module of a library

replace a module in the library file with a new module

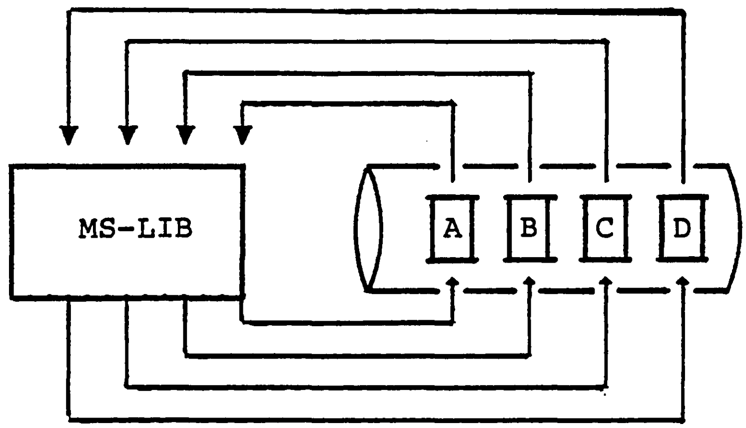
create a library file

During each library session, MS-LIB first deletes or extracts modules, then appends new ones. In a single operation, MS-LIB reads each module into memory, checks it for consistency, and writes it back to the file. If you delete a module, MS-LIB reads in that module but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places the module at the end of the last module written. This procedure effectively "closes up" the disk space to keep the library file from growing larger than necessary. When MS-LIB has read through the whole library file, it apends any new modules to the end of the file. Finally, MS-LIB creates the index, which MS-LINK uses to find modules and symbols in the library file, and outputs a cross reference listing of the PUBLIC symbols in the library, if you request such a listing. (Building the library index may take some extra time, up to 20 second in some cases.)

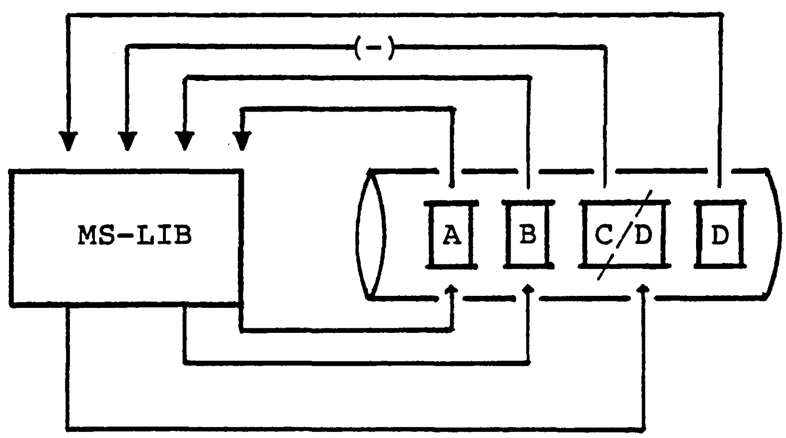
For example:

```
LIB PASCAL+HEAP-HEAP;
```

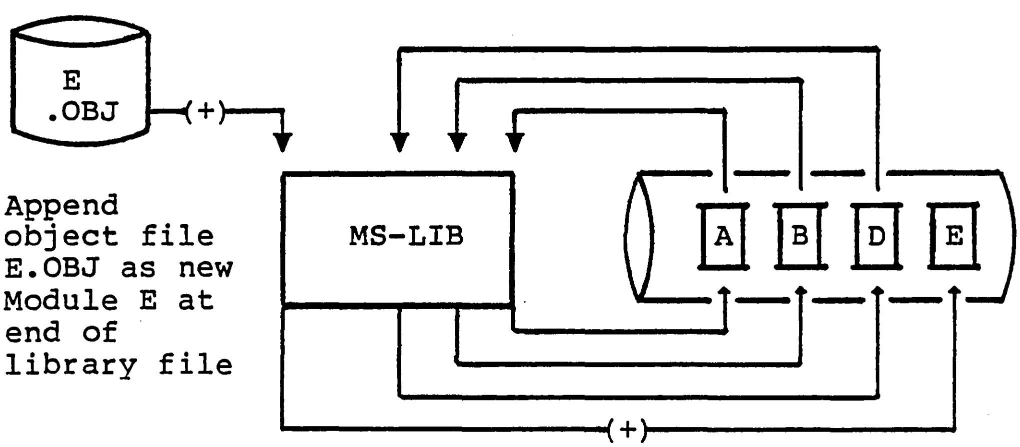
first deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. This order of execution prevents confusion in MS-LIB when a new version of a module replaces a version in the library file. Note that the replace function is simply the delete-append functions in succession. Also note that you can specify delete, append, or extract functions in any order; the order is insignificant to the MS-LIB command scanner.




Consistency
Check only

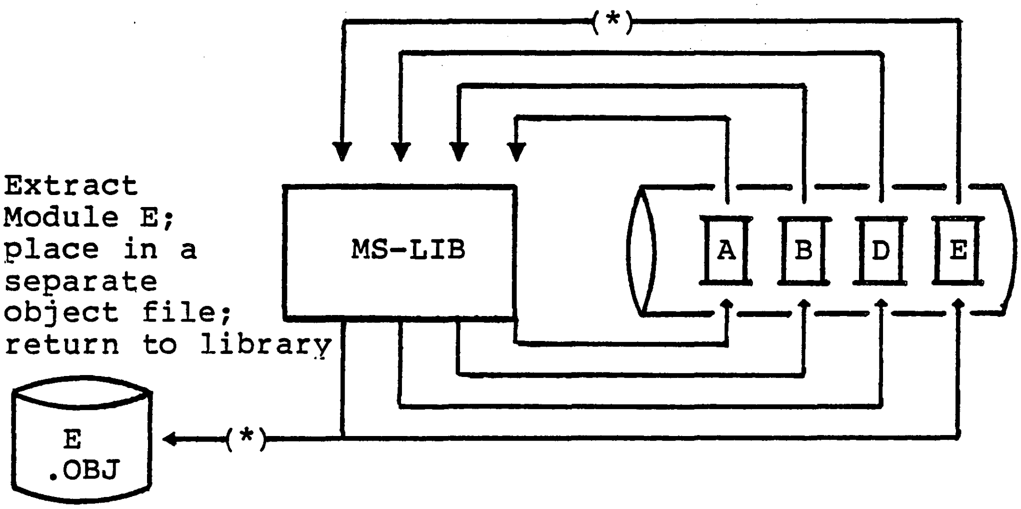



Delete
Module C;
Module D
written to
space of
Module C

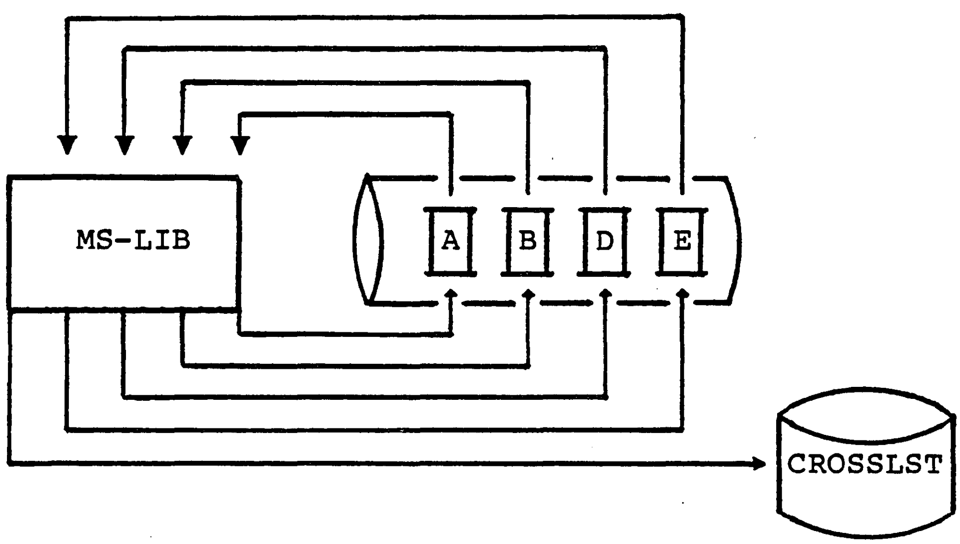



Append
object file
E.OBJ as new
Module E at
end of
library file

Extract
Module E;
place in a
separate
object file;
return to library




Consistency
Check, then
output a
cross
reference
listing of
PUBLIC
symbols



# CHAPTER 1

# RUNNING MS-LIB

Running MS-LIB requires two types of commands: a command to invoke MS-LIB and answers to command prompts. Usually you will enter all the commands to MS-LIB on the terminal keyboard. As an option, answers to the command prompts may be contained in a Response File. Some special command characters exist. Some are used as a required part of MS-LIB commands. Others assist you while entering MS-LIB commands.

## 1.1 INVOKING MS-LIB

MS-LIB may be invoked three 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 MS-LIB. By the third method, you create a Response File that contains all the necessary commands.

Summary of Methods to invoke MS-LIB

| | |
|---|---|
| Method 1 | LIB |
| Method 2 | LIB <library><operations>,<listing> |
| Method 3 | LIB @<filespec> |

### 1.1.1 Method 1: LIB

Enter:

```
LIB
```

MS-LIB will be loaded into memory. Then, MS-LIB returns a series of three text prompts that appear one at a time. You answer the prompts as commands to MS-LIB to perform specific tasks.

The Command Prompts and Command Characters are summarized here. The Command Prompts and Command Characters are described fully in Sections 1.2 and 1.3.

Summary of Command Prompts

| PROMPT | RESPONSES |
|---|---|
| Library file: | List filename of library to be manipulated (default: filename extension .LIB) |
| Operation: | List command character(s) followed by module name(s) or object filename(s) (default action: no changes. default object filename extension: .OBJ) |
| List file: | List filename for a cross reference listing file (default: NUL; no file) |

Summary of Command Characters

| Character | Action |
|---|---|
| + | Append an object file as the last module |
| - | Delete a module from the library |
| * | Extract a module and place in an object file |
| ; | Use default responses to remaining prompts |
| & | Extend current physical line; repeat command prompt |
| Control-C | Abort library session. |

### 1.1.2 Method 2: LIB <library><operations>,<listing>

Enter:

LIB <library><operations>,<listing>

The entries following LIB are responses to the command prompts. The library and operations fields and all operations entries must be separated by one of the command characters plus, minus, and asterisk (+, -, *). If a cross reference listing is wanted, the name of the file must be separated from the last operations entry by a comma.

where: library is the name of a library file. MS-LIB assumes that the filename extension is .OBJ, which you may override by specifying a different extension. If the filename given for the library field does not exist, MS-LIB will prompt you:

Library file does not exist. Create?

Enter Yes (or any response beginning with Y) to create a new library file. Enter No (or any other response not beginning with Y) to abort the library session.

operations is deleting a module, appending an object file as a module, or extracting a module as an object file from the library file. Use the three command characters plus (+), minus (-), and asterisk (*) to direct MS-LIB what to do with each module or object file.

listing is the name of the file you want to receive the cross reference listing of PUBLIC symbols in the modules in the library. The list is compiled after all module manipulation has taken place.

To select the default for remaining field(s), you may enter the semicolon command character.

If you enter a Library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you enter a Library filename followed immediately by a comma and a List filename, MS-LIB will perform its consistency check of the library file, then produce the cross reference listing file.

Example

LIB PASCAL-HEAP+HEAP;

This example causes MS-LIB to delete the module HEAP from the library file PASCAL.LIB, then append the object file HEAP.OBJ as the last module of PASCAL.LIB (the module will be named HEAP).

If you have many operations to perform during a library session, use the ampersand (&) command character to extend the line so that you can enter additional object filenames and module names. Be sure to always include one of the command characters for operations (+, -, *) before the name of each module or object filename.

Example

    LIB PASCAL<CR>

causes MS-LIB to perform a consistency check of the library file PASCAL.LIB. No other action is performed.

Example

    LIB PASCAL,PASCROSS.PUB

causes MS-LIB to perform a consistency check of the library file PASCAL.LIB, then output a cross reference listing file named PASCROSS.PUB.

### 1.1.3 Method 3: LIB @<filespec>

Enter:

LIB @<filespec>

where: filespec is the name of a Response File. A Response File contains answers to the MS-LIB prompts (summarized under method 1 for invoking and described fully in Section 1.2). Method 3 permits you to conduct the MS-LIB session without interactive (direct) user responses to the MS-LIB prompts.

IMPORTANT

> Before using method 3 to invoke MS-LIB, you 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 Command Characters in the Response File the same way as they are used for responses entered on the terminal keyboard.

When the library 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, MS-LIB will use the default responses (no changes to the modules currently in the library file for Operation, and no cross reference listing file created).

If you enter a Library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you enter a Library filename then only a carriage return of Operations then a comma and a List filename, MS-LIB will perform its consistency check of the library file, then produce the cross reference listing file.

Example:

```
PASCAL<CR>
+CURSOR+HEAP-HEAP*FOIBLES<CR>
CROSSLST<CR>
```

This Response File will cause MS-LIB to delete the module HEAP from the PASCAL.LIB library file, extract the module FOIBLES and place in an object file named FOIBLES.OBJ, then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Then, MS-LIB will create a cross reference file named CROSSLST.

## 1.2 COMMAND PROMPTS

MS-LIB is commanded by entering responses to three text prompts. When you have entered your response to the current prompt, the next appears. When the last prompt has been answered, MS-LIB performs its library management functions without further command. When the library session is finished, MS-LIB exits to the operating system. When the operating system prompt is displayed, MS-LIB has finished the library session successfully. If the library session is unsuccessful, MS-LIB returns the appropriate error message.

MS-LIB prompts you for the name of the library file, the operation(s) you want to perform, and the name you want to give to a cross reference listing file, if any.

Library file:

Enter the name of the library file that you want to manipulate. MS-LIB assumes that the filename extension is .LIB. You can override this assumption by giving a filename extension when you enter the library filename. Because MS-LIB can manage only one library file at a time, only one filename is allowed in response to this prompt. Additional responses, except the semicolon command character, are ignored.

If you enter a library filename and follow it immediately with a semicolon command character, MS-LIB will perform a consistency check only, then return to the operating system. Any errors in the file will be reported.

If the filename you enter does not exist, MS-LIB returns the prompt:

```
Library file does not exist. Create?
```

You must enter either Yes or No, in either upper or lower (or mixed) case. Actually, MS-LIB checks the response for the letter Y as the first charcter. If any other character is entered first, MS-LIB terminates and returns to the operating system.

Operation:

Enter one of the three command characters for manipulating modules (+, -, *), followed immediately (no space) by the module name or the object filename. Plus sign appends an object file as the last module in the library file (see further discussion under the description of plus sign below). Minus sign deletes a module from the library file. Asterisk extracts a module from the library and places it in a separate object file with the filename taken from the module name and a filename extension .OBJ.

When you have a large number of modules to manipulate (more than can be typed on one line), enter an ampersand (&) as the last character on the line. MS-LIB will repeat the Operation prompt, which permits you to enter additional module names and object filenames.

MS-LIB allows you to enter operations on modules and object files in any order you want.

More information about order of execution and what MS-LIB does with each module is given in the descriptions of each Command Character.

List file:

If you want a cross reference list of the PUBLIC symbols in the modules in the library file after your manipulations, enter a filename in which you want MS-LIB to place the cross reference listing. If you do not enter a filename, no cross reference listing is generated (a NUL file).

The response to the List file prompt is a file specification. Therefore, you can specify, along with the filename, a drive (or device) designation and a filename extension. The List file is not given a default filename extension. If you want the file to have a filename extension, you must specify it when entering the filename.

The cross reference listing file contains two lists. The first list is an alphabetical listing of all PUBLIC symbols. Each symbol name is followed by the name of its module. The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the PUBLIC symbols in that module.

## 1.3 COMMAND CHARACTERS

MS-LIB provides six command characters: three of the command characters are required in responses to the Operation prompt; the other three command characters provide you additional helpful commands to MS-LIB.

+ The plus sign followed by an object filename appends the object file as the last module in the library named in reponse to the Library file prompt. When MS-LIB sees the plus sign, it assumes that the filename extension is .OBJ. You may override this assumption by specifying a different filename extension.

  MS-LIB strips the drive designation and the extension from the object file specification, leaving only the filename. For example, if the object file to be appended as a module to a library is:

  ```
  B:CURSOR.OBJ
  ```

  a response to the Operation prompt of:

  ```
  +B:CURSOR.OBJ
  ```

  causes MS-LIB to strip off the B: and the .OBJ, leaving only CURSOR, which becomes a module named CURSOR in the library.

  NOTE

  The distinction between an object file and a module (or object module) is that the file possesses a drive designation (even if it is default drive) and a filename extension. Object modules possess neither of these.

\- The minus sign followed by a module name deletes that module from the library file. MS-LIB then "closes up" the file space left empty by the deletion. This cleanup action keeps the library file from growing larger than necessary with empty space. Remember that new modules, even replacement modules are added to the end of the file, not stuffed into space vacated by deleting modules.

* The asterisk followed by a module name extracts that module from the library file and places it into a separate object file. The module will still exist in the library (extract means, essentially, copy the module to a separate object file). The module name is used as the filename. MS-LIB adds the default drive designation and the filename extension .OBJ. For example, if the module to be extracted is:

    CURSOR

and the current default disk drive is A:, a reponse to the Operation prompt of:

    *CURSOR

causes MS-LIB to extract the module named CURSOR from the library file and to set it up as an object file with the file specification of:

    default drive:CURSOR.OBJ

(The drive designation and filename extension cannot be overridden. You can, however, rename the file, giving a new filename extension, and/or copy the file to a new disk drive, giving a new filename and/or filename extension.)

; Use a single semicolon (;) followed immediately by a carriage return at any time after responding to the first prompt (from Library file on) to select default responses to the remaining prompts. This feature saves time and overrides the need to answer additional prompts.

NOTE

Once the semicolon has been entered, you can no longer respond to any of the prompts for that library session. Therefore, do not use the semicolon to skip over some prompts. For this, use carriage return.

Example:

```
Library file:  FUN <CR>
Operation:  +CURSOR;<CR>
```

The remaining prompt will not appear, and MS-LIB will use the default value (no cross reference file).

& Use the ampersand to extend the current physical line. This command character will only be needed for the Operation prompt. MS-LIB can perform many functions during a single library session. The number of modules you can append it limited only be disk space. The number of module you can replace or extract is also limited only by disk space. The number of modules you can delete is limited only by the number of modules in the library file. However, the line length for a response to any prompt is limited to the line length of your system. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. MS-LIB will display the Operation prompt again, then enter more responses. You may use the ampersand character as many times as you need. For example:

```
Library file:  FUN<CR>
Operation:  +CURSOR-HEAP+HEAP*FOIBLES&
Operation:  *INIT+ASSUME+RIDE;<CR>
```

MS-LIB will delete the module HEAP, extract the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then append the object files CURSOR, HEAP, ASSUME, and RIDE. Note, however, that MS-LIB allows you to enter your Operation reponses in any order.

Control-C Use Control-C at any time to abort the library session. If you enter an erroneous response, such as the wrong filename or module name, or an incorrectly spelled filename or module name, you must press CTRL-C to exit MS-LIB then reinvoke MS-LIB and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

# CHAPTER 2

## ERROR MESSAGES

<symbol> is a multiply defined PUBLIC. Proceed?
Cause: two modules define the same public symbol. The user is asked to confirm the removal of the definition of the old symbol. A No response leaves the library in an undetermined state.
Cure: Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

Allocate error on VM.TMP
Cause: out of space

Cannot create extract file
Cause: no room in directory for extract file

Cannot create list file
Cause: No room in directory for library file

Cannot nest response file
Cause: '@filespec' in response (or indirect) file

Cannot open VM.TMP
Cause: no room for VM.TMP in disk directory

Cannot write library file
Cause: Out of space

Close error on extract file
Cause: out of space

Error: An internal error has occurred.
Contact Microsoft, Inc.

Fatal Error: Cannot open input file
Cause: Mistyped object file name

Fatal Error: Module is not in the library
Cause: trying to delete a module that is not in the library

Input file read error
Cause: bad object module or faulty disk

Invalid object module/library
Cause: bad object and/or library

Library Disk is full
Cause: no more room on diskette

Listing file write error
Cause: out of space

No library file specified
Cause: no response to Library File prompt

Read error on VM.TMP
Cause: disk not ready for read

Symbol table capacity exceeded
Cause: too many public symbols (about 30K chars in symbols)

Too many object modules
Cause: more than 500 object modules

Too many public symbols
Cause: 1024 public symbols maximum

Write error on library/extract file
Cause: Out of space

Write error on VM.TMP
Cause: out of space

INDEX

# microsoft
# MS-CREF
# cross reference facility
# manual

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

## LIMITED WARRANTY

**MICROSOFT, Inc.** shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

**THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.**

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.



8407D-100-00

# INTRODUCTION

## Features and Benefits

The MS-CREF Cross Reference Facility can aid you in debugging your assembly language programs. MS-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 MS-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.

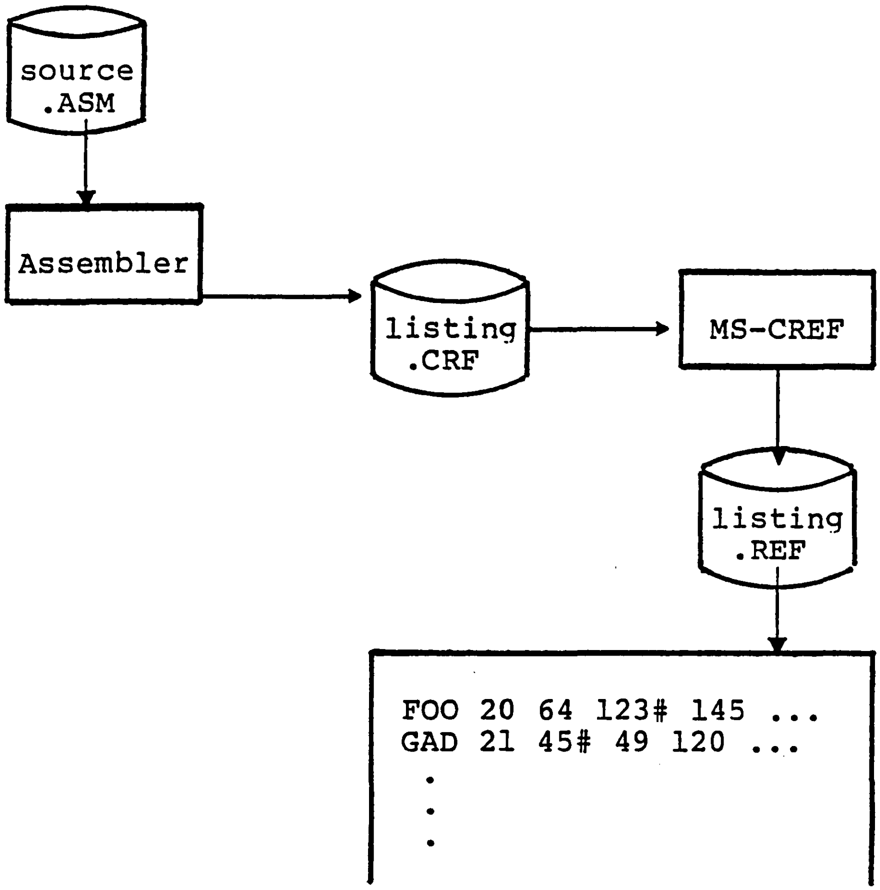
The cross reference listing produced by MS-CREF provides you the locations, speeding your search and allowing faster debugging.

## Overview of MS-CREF Operation

MS-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, MS-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, MS-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 pound sign (#).


source
.ASM
Assembler
listing
.CRF
MS-CREF
listing
.REF
FOO 20 64 123# 145 ...
GAD 21 45# 49 120 ...
.
.
.

## System Requirements

The MS-CREF Cross Reference Facility requires:

24K bytes of memory minimum:
- 14K bytes for code
- 10K bytes for run space

1 disk drive
- 1 disk drive if and only if output is sent to the same physical diskette from which the input was taken. None of the utility programs in this package allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

# CHAPTER 1

# RUNNING MS-CREF

Running MS-CREF requires two types of commands: a command to invoke MS-CREF and answers to command prompts. You will enter all the commands to MS-CREF on the terminal keyboard. Some special command characters exist to assist you while entering MS-CREF commands.

Before you can use MS-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 1.1.

## 1.1 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 this 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 MS-CREF prompt (see Section 1.2.1).

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 MS-CREF.

## 1.2 INVOKING MS-CREF

MS-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 MS-CREF.

Summary of Methods to invoke MS-CREF

| | |
|---|---|
| Method 1 | CREF |
| Method 2 | CREF <crffile>,<listing> |

### 1.2.1 Method 1: CREF

Enter:

CREF

MS-CREF will be loaded into memory. Then, MS-CREF returns a series of two text prompts that appear one at a time. You answer the prompts to command MS-CREF to convert a cross reference file into a cross reference listing.

Command Prompts

<u>Cross reference [.CRF]:</u>

Enter the name of the cross reference file you want MS-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.

MS-CREF assumes that the filename extension is .CRF. If you do not specify a filename extension when you enter the cross reference filename, MS-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 Chapter 3, Format of MS-CREF Compatible Files, for a description of what MS-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.

<u>Listing [crffile.REF]:</u>

Enter the name you want the cross reference listing file to have. MS-CREF will automatically give the cross reference listing the filename extension .REF.

If you want you 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 default 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, MS-CREF gives the listing file the filename of the cross reference file and the default filename extension .REF.

Example:

```
Cross reference [.CRF]:  FUN;
```

MS-CREF will process the cross reference file named FUN.CRF and output a listing file named FUN.REF.

Control-C Use Control-C at any time to abort the MS-CREF session. If you enter an erroneous response, (the wrong filename), or an incorrectly spelled filename, you must press Control-C to exit MS-CREF then reinvoke MS-CREF and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

### 1.2.2 Method 2: CREF <crffile>,<listing>

Enter:

    CREF <crffile>,<listing>

MS-CREF will be loaded into memory. Then, MS-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 separate by a comma.

where: crffile is the name of a cross reference file produced by your assembler. MS-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, MS-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;<CR>

This example causes MS-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 MS-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:.

## 1.3 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 pound sign (#) appended to it.

On the next page is a cross reference listing as an example:

```
MS-CREF     (vers no.)     (date)

ENTX    PASCAL entry for initializing programs  <--comes from
                                                   TITLE directive

 Symbol Cross Reference      (# is definition)      Cref-1

AAAXQQ . . . . . . . .      37#   38

BEGHQQ . . . . . . . .      83    84#   154   176
BEGOQQ . . . . . . . .      33   162
BEGXQQ . . . . . . . .     113   126#   164   223

CESXQQ . . . . . . . .      97    99#   129
CLNEQQ . . . . . . . .      67    68#
CODE . . . . . . . . .      37   182
CONST. . . . . . . . .     104   104    105   110
CRCXQQ . . . . . . . .      93    94#   210   215
CRDXQQ . . . . . . . .      95    96#   216
CSXEQQ . . . . . . . .      65    66#   149
CURHQQ . . . . . . . .      85    86#   155

DATA . . . . . . . . .      64#   64    100   110
DGROUP . . . . . . . .     110#  111    111   111   127   153   171   172
DOSOFF . . . . . . . .      98#  198    199
DOSXQQ . . . . . . . .     184   204#   219

ENDHQQ . . . . . . . .      87    88#   158
ENDOQQ . . . . . . . .      33#  195
ENDUQQ . . . . . . . .      31#  197
ENDXQQ . . . . . . . .     184   194#
ENDYQQ . . . . . . . .      32#  196
ENTGQQ . . . . . . . .      30#  187
ENTXCM . . . . . . . .     182#  183    221

FREXQQ . . . . . . . .     169   170#   178

HDRFQQ . . . . . . . .      71    72#   151
HDRVQQ . . . . . . . .      73    74#   152
HEAP . . . . . . . . .      42    44    110
HEAPBEG. . . . . . . .      54#  153    172
HEAPLOW. . . . . . . .      43   171

INIUQQ . . . . . . . .      31   161

MAIN_STARTUP . . . . .     109#  111    180
MEMORY . . . . . . . .      42    48#    48    49   109   110

PNUXQQ . . . . . . . .      69    70    150

RECEQQ . . . . . . . .      81    82#
REFEQQ . . . . . . . .      77    78#
REPEQQ . . . . . . . .      79    80#
RESEQQ . . . . . . . .      75    76#   148
```

```
SKTOP. . . . . . . . .        59#
SMLSTK . . . . . . . .       135   137#
STACK. . . . . . . . .        53#   53     60    110
STARTMAIN. . . . . . .       163   186#   200
STKBQQ . . . . . . . .        89    90#   146
STKHQQ . . . . . . . .        91    92#   160
```

# CHAPTER 2

## ERROR MESSAGES

All errors cause MS-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<br>Unrecoverable disk I/O error |
| 101 | Device name error<br>Illegal device specification (for example, X:FOO.CRF) |
| 103 | Internal error<br>Report to Microsoft, Inc. |
| 104 | Internal error<br>Report to Microsoft, Inc. |
| 105 | Device offline<br>disk drive door open, no printer attached, and so on. |
| 106 | Internal error<br>Report to Microsoft, Inc. |
| 108 | Disk full |
| 110 | File not found |
| 111 | Disk is write protected |
| 112 | Internal error<br>Report to Microsoft, Inc. |
| 113 | Internal error<br>Report to Microsoft, Inc. |
| 114 | Internal error<br>Report to Microsoft, Inc. |
| 115 | Internal error<br>Report to Microsoft, Inc. |

# CHAPTER 3

## FORMAT OF MS-CREF COMPATIBLE FILES

MS-CREF will process files other than those generated by Microsoft's assembler as long as the file conforms to the format that MS-CREF expects.

### 3.1 GENERAL DESCRIPTION OF MS-CREF FILE PROCESSING

In essence, MS-CREF reads a stream of bytes from the cross 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.

MS-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).

MS-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 MS-CREF encounters more than one title symbol in the source file, it uses the last title read for all page headings. If MS-CREF does not encounter a title symbol in the file, the title line on the listing is left blank.

## 3.2 FORMAT OF SOURCE FILES

MS-CREF uses the first three bytes of the source file as format specification data. The rest of the file 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 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 MS-CREF that the two preceding bytes define listing page size.

If MS-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 MS-CREF recognizes and the byte values and placement it uses to recognize record types.

Record 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 title 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   Control-A
02   Control-B
04   Control-D
05   Control-E
06   Control-F
07   Control-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 Control-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 the title placed at the top of all pages of the listing. If a title definition 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 page (range is from 1 to 132 characters). The default page length is 58 lines. The default line length is 80 characters.

<u>Summary of CRF File Record Contents</u>

| 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 |

## INDEX

Contents

# APPENDIX A

## ASCII CHARACTER CODES

| Dec | Hex | CHR | Dec | Hex | CHR | Dec | Hex | CHR |
|---|---|---|---|---|---|---|---|---|
| 000 | 00H | NUL | 043 | 2BH | + | 086 | 56H | V |
| 001 | 01H | SOH | 044 | 2CH | , | 087 | 57H | W |
| 002 | 02H | STX | 045 | 2DH | - | 088 | 58H | X |
| 003 | 03H | ETX | 046 | 2EH | . | 089 | 59H | Y |
| 004 | 04H | EOT | 047 | 2FH | / | 090 | 5AH | Z |
| 005 | 05H | ENQ | 048 | 30H | 0 | 091 | 5BH | [ |
| 006 | 06H | ACK | 049 | 31H | 1 | 092 | 5CH | \ |
| 007 | 07H | BEL | 050 | 32H | 2 | 093 | 5DH | ] |
| 008 | 08H | BS | 051 | 33H | 3 | 094 | 5EH | ^ |
| 009 | 09H | HT | 052 | 34H | 4 | 095 | 5FH | _ |
| 010 | 0AH | LF | 053 | 35H | 5 | 096 | 60H | ` |
| 011 | 0BH | VT | 054 | 36H | 6 | 097 | 61H | a |
| 012 | 0CH | FF | 055 | 37H | 7 | 098 | 62H | b |
| 013 | 0DH | CR | 056 | 38H | 8 | 099 | 63H | c |
| 014 | 0EH | SO | 057 | 39H | 9 | 100 | 64H | d |
| 015 | 0FH | SI | 058 | 3AH | : | 101 | 65H | e |
| 016 | 10H | DLE | 059 | 3BH | ; | 102 | 66H | f |
| 017 | 11H | DC1 | 060 | 3CH | < | 103 | 67H | g |
| 018 | 12H | DC2 | 061 | 3DH | = | 104 | 68H | h |
| 019 | 13H | DC3 | 062 | 3EH | > | 105 | 69H | i |
| 020 | 14H | DC4 | 063 | 3FH | ? | 106 | 6AH | j |
| 021 | 15H | NAK | 064 | 40H | @ | 107 | 6BH | k |
| 022 | 16H | SYN | 065 | 41H | A | 108 | 6CH | l |
| 023 | 17H | ETB | 066 | 42H | B | 109 | 6DH | m |
| 024 | 18H | CAN | 067 | 43H | C | 110 | 6EH | n |
| 025 | 19H | EM | 068 | 44H | D | 111 | 6FH | o |
| 026 | 1AH | SUB | 069 | 45H | E | 112 | 70H | p |
| 027 | 1BH | ESCAPE | 070 | 46H | F | 113 | 71H | q |
| 028 | 1CH | FS | 071 | 47H | G | 114 | 72H | r |
| 029 | 1DH | GS | 072 | 48H | H | 115 | 73H | s |
| 030 | 1EH | RS | 073 | 49H | I | 116 | 74H | t |
| 031 | 1FH | US | 074 | 4AH | J | 117 | 75H | u |
| 032 | 20H | SPACE | 075 | 4BH | K | 118 | 76H | v |
| 033 | 21H | ! | 076 | 4CH | L | 119 | 77H | w |
| 034 | 22H | " | 077 | 4DH | M | 120 | 78H | x |
| 035 | 23H | # | 078 | 4EH | N | 121 | 79H | y |
| 036 | 24H | $ | 079 | 4FH | O | 122 | 7AH | z |
| 037 | 25H | % | 080 | 50H | P | 123 | 7BH | { |
| 038 | 26H | & | 081 | 51H | Q | 124 | 7CH | \| |
| 039 | 27H | ' | 082 | 52H | R | 125 | 7DH | } |
| 040 | 28H | ( | 083 | 53H | S | 126 | 7EH | ~ |
| 041 | 29H | ) | 084 | 54H | T | 127 | 7FH | DEL |
| 042 | 2AH | * | 085 | 55H | U | | | |

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX B

Table of MACRO-86 Directives

## B.1 MEMORY DIRECTIVES

```
               ASSUME <seg-reg>:<seg-name>[,<seg-reg>:<seg-name>...]
               ASSUME NOTHING
               COMMENT <delim><text><delim>

        <name> DB <exp>
        <name> DD <exp>
        <name> DQ <exp>
        <name> DT <exp>
        <name> DW <exp>

               END [<exp>]
        <name> EQU <exp>
        <name> = <exp>
               EXTRN <name>:<type>[,<name>:<type>...]
               PUBLIC <name>[,<name>...]
        <name> LABEL <type>
               NAME <module-name>

        <name> PROC [NEAR]
        <name> PROC [FAR]
                 |
   <proc-name> ENDP

               .RADIX <exp>
        <name> RECORD <field>:<width>[=<exp>][,...]

        <name> GROUP <segment-name>[,...]
        <name> SEGMENT [<align>][<combine>][<class>]
                 |
    <seg-name> ENDS
               EVEN
               ORG <exp>

        <name> STRUC
                 |
  <struc-name> ENDS
```

## B.2 MACRO DIRECTIVES

```
        ENDM
        EXITM
        IRP <dummy>,<parameters in angle brackets>
        IRPC <dummy>,string
        LOCAL <parameter>[,<parameter>...]
 <name> MACRO <parameter>[,<parameter>...]
        PURGE <macro-name>[,...]
        REPT <exp>
```

Special Macro Operators

- & (ampersand) - concantenation
- <text> (angle brackets - single literal)
- ;; (double semicolons) - suppress comment
- ! (exclamation point) - next character literal
- % (percent sign) - convert expression to number

## B.3 CONDITIONAL DIRECTIVES

```
ELSE
ENDIF
IF <exp>
IFB <arg>
IFDEF <symbol>
IFDIF <arg1>,<arg2>
IFE <exp>
IFIDN <arg1>,<arg2>
IFNB <arg>
IFNDEF <symbol>
IF1
IF2
```

## B.4 LISTING DIRECTIVES

```
.CREF
.LALL
.LFCOND
.LIST
%OUT <text>
PAGE <exp>
.SALL
.SFCOND
SUBTTL <text>
.TFCOND
TITLE <text>
.XALL
.XCREF
.XLIST
```

## B.5 ATTRIBUTE OPERATORS

### Override operators

```
Pointer (PTR)
    <attribute>   PTR   <expression>
Segment Override (:) (colon)
    <segment-register>:<address-expression>
    <segment-name>:<address-expression>
    <group-name>:<address-expression>
SHORT
    SHORT <label>
THIS
    THIS <distance>
    THIS <type>
```

### Value Returning Operators

```
SEG
    SEG <label>
    SEG <variable>
OFFSET
    OFFSET <label>
    OFFSET <variable>
TYPE
    TYPE <label>
    TYPE <variable>
.TYPE
    .TYPE <variable>
LENGTH
    LENGTH <variable>
SIZE
    SIZE <variable>
```

### Record Specific operators

```
Shift-count  - (Record fieldname)
    <record-fieldname>
MASK
    MASK <record-fieldname>
WIDTH
    WIDTH <record-fieldname>
    WIDTH <record>
```

## B.6 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. *, /, MOD, 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

# APPENDIX C

## Table of 8086 Instructions

The mnemonics are listed alphabetically with their full names. The 8086 instructions are also listed in groups based on the type of arguments the instruction takes.

### C.1 8086 INSTRUCTION MNEMONICS, ALPHABETICAL

| Mnemonic | Full Name |
|---|---|
| AAA | ASCII adjust for addition |
| AAD | ASCII adjust for division |
| AAM | ASCII adjust for multiplication |
| AAS | ASCII adjust for subtraction |
| ADC | Add with carry |
| ADD | Add |
| AND | AND |
| CALL | CALL |
| CBW | Convert byte to word |
| CLC | Clear carry flag |
| CLD | Clear direction flag |
| CLI | Clear interrupt flag |
| CMC | Complement carry flag |
| CMP | Compare |
| CMPS | Compare byte or word (of string) |
| CMPSB | Compare byte string |
| CMPSW | Compare word string |
| CWD | Convert word to double word |
| DAA | Decimal adjust for addition |
| DAS | Decimal adjust for subtraction |
| DEC | Decrement |
| DIV | Divide |
| ESC | Escape |
| HLT | Halt |
| IDIV | Integer divide |
| IMUL | Integer multiply |
| IN | Input byte or word |
| INC | Increment |
| INT | Interrupt |
| INTO | Interrupt on overflow |

| Mnemonic | Full Name |
|---|---|
| IRET | Interrupt return |
| JA | Jump on above |
| JAE | Jump on above or equal |
| JB | Jump on below |
| JBE | Jump on below or equal |
| JC | Jump on carry |
| JCXZ | Jump on CX zero |
| JE | Jump on equal |
| JG | Jump on greater |
| JGE | Jump on greater or equal |
| JL | Jump on less than |
| JLE | Jump on less than or equal |
| JMP | Jump |
| JNA | Jump on not above |
| JNAE | Jump on not above or equal |
| JNB | Jump on not below |
| JNBE | Jump on not below or equal |
| JNC | Jump on no carry |
| JNE | Jump on not equal |
| JNG | Jump on not greater |
| JNGE | Jump on not greater or equal |
| JNL | Jump on not less than |
| JNLE | Jump on not less than or equal |
| JNO | Jump on not overflow |
| JNP | Jump on not parity |
| JNS | Jump on not sign |
| JNZ | Jump on not zero |
| JO | Jump on overflow |
| JP | Jump on parity |
| JPE | Jump on parity even |
| JPO | Jump on parity odd |
| JS | Jump on sign |
| JZ | Jump on zero |
| LAHF | Load AH with flags |
| LDS | Load pointer into DS |
| LEA | Load effective address |
| LES | Load pointer into ES |
| LOCK | LOCK bus |
| LODS | Load byte or word (of string) |
| LODSB | Load byte (string) |
| LODSW | Load word (string) |
| LOOP | LOOP |
| LOOPE | LOOP while equal |
| LOOPNE | LOOP while not equal |
| LOOPNZ | LOOP while not zero |
| LOOPZ | LOOP while zero |
| MOV | Move |
| MOVS | Move byte or word (of string) |
| MOVBS | Move byte (string) |
| MOVSW | Move word (string) |
| MUL | Multiply |
| NEG | Negate |
| NOP | No operation |
| NOT | NOT |

| Mnemonic | Full Name |
|---|---|
| OR | OR |
| OUT | Output byte or word |
| POP | POP |
| POPF | POP flags |
| PUSH | PUSH |
| PUSHF | PUSH flags |
| RCL | Rotate through carry left |
| RCR | Rotate through carry right |
| REP | Repeat |
| RET | Return |
| ROL | Rotate left |
| ROR | Rotate right |
| SAHF | Store AH into flags |
| SAL | Shift arithmetic left |
| SAR | Shift arithmetic right |
| SBB | Subtract with borrow |
| SCAS | Scan byte or word (of string) |
| SCASB | Scan byte (string) |
| SCASW | Scan word (string) |
| SHL | Shift left |
| SHR | Shift right |
| STC | Set carry flag |
| STD | Set direction flag |
| STI | Set interrupt flag |
| STOS | Store byte or word (of string) |
| STOSB | Store byte (string) |
| STOSW | Store word (string) |
| SUB | Subtract |
| TEST | TEST |
| WAIT | WAIT |
| XCHG | Exchange |
| XLAT | Translate |
| XOR | Exclusive OR |

## C.2 8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

In this section, the instructions are grouped according to the type of argument(s) they take. In each group the instructions are listed alphabetically in the first column. The formats of the instructions with the valid argument types are shown in the second column. If a format shows OP, that format is legal for all the instructions shown in that group. If a format is specific to one mnemonic, the mnemonic is shown in the format instead of OP.

The following abbreviations are used in these lists:

```
    OP = opcode;  instruction mnemonic

   reg = byte register (AL,AH,BL,BH,CL,CH,DL,DH)
      or word register (AX,BX,CX,DX,SI,DI,BP,SP)

   r/m = register or memory address or indexed and/or based

 accum = AX or AL register

 immed = immediate

   mem = memory operand

segreg = segment register (CS,DS,SS,ES)
```

### General 2 operand instructions

| Mnemonics | Argument Types |
|---|---|
| ADC | OP reg,r/m |
| ADD | OP r/m,reg |
| AND | OP accum,immed |
| CMP | OP r/m,immed |
| OR | |
| SBB | |
| SUB | |
| TEST | |
| XOR | |

### CALL and JUMP type instructions

| Mnemonics | Argument Types |
|---|---|
| CALL | OP mem {NEAR}{FAR} direction |
| JMP | OP r/m (indirect data -- DWORD, WORD) |

8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

## Relative jumps

Argument Type

OP addr (+129 or -126 of IP at start, or ±127 at end of jump instruction)

Mnemonics

| | | | | |
|---|---|---|---|---|
| JA | JC | JZ | JNGE | JNP |
| JNBE | JNAE | JG | JLE | JPO |
| JAE | JBE | JNLE | JNG | JNS |
| JNB | JNA | JGE | JNE | JO |
| JNC | JCXZ | JNL | JNZ | JP |
| JB | JE | JL | JNO | JPE |
| | | | | JS |

## Loop instructions : same as Relative jumps

LOOP LOOPE LOOPZ LOOPNE LOOPNZ

## Return instruction

| Mnemonic | Argument Type |
|---|---|
| RET | [immed] (optional, number of words to POP) |
| IRET | stack + 6 |

## No operand instructions

Mnemonics

| | | | | | |
|---|---|---|---|---|---|
| AAA | CLD | DAA | LODSB | PUSHF | STI |
| AAD | CLI | DAS | LODSW | SAHF | STOSB |
| AAM | CMC | HLT | MOVSB | SCASB | STOSW |
| AAS | CMPSB | INTO | MOVSW | SCASW | WAIT |
| CBW | CMPSW | IRET | NOP | STC | XLATB |
| CLC | CWD | LAHF | POPF | STD | |

## Load instructions

| Mnemonics | Argument Type |
|---|---|
| LDS | OP r/m (except that OP reg is illegal) |
| LEA | |
| LES | |

8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

## Move instructions

| Mnemonic | Argument Types |
|---|---|
| MOV | OP mem,accum |
| | OP accum,mem |
| | OP segreg,r/m (except CS is illegal) |
| | OP r/m,segreg |
| | OP r/m,reg |
| | OP reg,r/m |
| | OP reg,immed |
| | OP r/m,immed |

## Push and pop instructions

| Mnemonics | Argument Types |
|---|---|
| PUSH | OP word-reg |
| POP | OP segreg (POP CS is illegal) |
| PUSHF | OP r/m |
| POPF | |

## Shift/rotate type instructions

| Mnemonics | Argument Types |
|---|---|
| RCL | OP r/m,1 |
| RCR | OP r/m,CL |
| ROL | |
| ROR | |
| SAL | |
| SHL | |
| SAR | |
| SHR | |

## Input/output instructions

| Mnemonics | Argument Types |
|---|---|
| IN | IN accum,byte-immed (immed = port 0-255) |
| | IN accum,DX |
| OUT | OUT immed,accum |
| | OUT DX,accum |

## Increment/decrement instructions

| Mnemonics | Argument Types |
|---|---|
| INC | OP word-reg |
| DEC | OP r/m |

## Arith. multiply/division/negate/not

| Mnemonics | Argument Type |
|---|---|
| DIV | OP r/m (implies AX OP r/m, except NEG) |
| IDIV | |
| MUL | |
| IMUL | |
| NEG | (NEG implies AX OP NOP) |
| NOT | |

## Interrupt instruction

| Mnemonic | Argument Types |
|---|---|
| INT | INT 3 (value 3 is one byte instruction)<br>INT byte-immed |

## Exchange instruction

| Mnemonic | Argument Types |
|---|---|
| XCHG | XCHG accum,reg<br>XCHG reg,accum<br>XCHG reg,r/m<br>XCHG r/m,reg |

## Miscellaneous instructions

| Mnemonics | Argument Types |
|---|---|
| XLAT | XLAT byte-mem (only checks argument, not in opcode) |
| ESC | ESC 6-bit-number,r/m |

## String primitives

These instructions have bits to record only their operand(s), if they are byte or word, and if a segment override is involved.

| | Mnemonics | Argument Types |
|---|---|---|
| * | CMPS (B/W) | CMPS byte-word,byte-word<br>(CMPS right operand is ES) |
| ? | LODS | LODS byte/word,byte/word<br>(LODS one argument = no ES) |
| * | MOVS | MOVS byte/word,byte/word<br>(MOVS left operand is ES) |
| * | SCAS | SCAS byte/word,byte/word<br>(SCAS one argument = ES) |
| * | STOS | STOS byte/word,byte/word<br>(STOS one argument = ES) |

CLD
STD
* DI : scas/stos
* ds: SI / es: DI

## Repeat prefix to string instructions

Mnemonics

| | | | | | |
|---|---|---|---|---|---|
| LOCK | REP | REPE | REPZ | REPNE | REPNZ |

# GENERAL INDEX