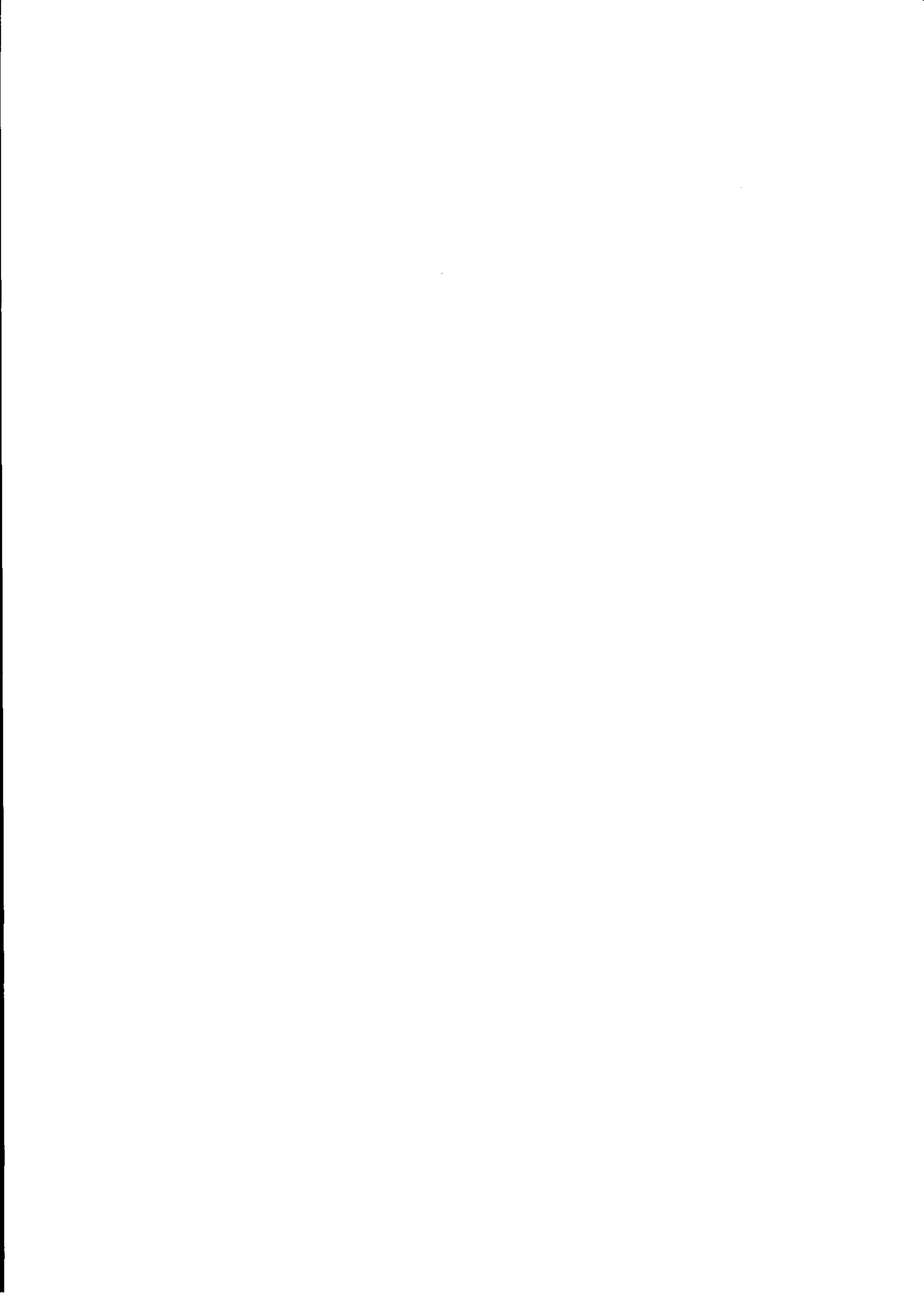
Operating Systems and Languages Library

MS-MACRO ASSEMBLER under MS-DOS

User Guide

olivetti



PREFACE

This manual is a user guide for the MS-DOS Macro Assembler (Microsoft Rel.-1.25) available on the Olivetti Personal Computer.

The MS-DOS Macro Assembler is a powerful assembler for 8086 based computers. This manual is intended for experienced assembly language programmers; it explains how to use MS-DOS utilities and features, but it does not teach you how to program in assembly language.

SUMMARY

This manual is divided into seven chapters providing a general overview, followed by specific information concerning source file creation and assembling, a detailed description of the Macro Assembler instructions and directives, and individual chapters on the Library Manager and the Cross Reference Utility.

RELATED PUBLICATIONS:

Installation and Operations Guide (Code 3986490 W)

MS-DOS User Guide (Code 4001410 G)

DISTRIBUTION: General (G)

SECOND EDITION: December 1984

Olivetti is a trademark of Ing. C. Olivetti & C., S.p.A.
OLITERM is a trademark of Ing. C. Olivetti & C., S.p.A.
SORTP is a trademark of Ing. C. Olivetti & C., S.p.A.

Copyright © Microsoft Corporation 1980÷1984

Copyright © 1984, by Olivetti
All rights reserved.

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 Ivrea (Italy)

TRADEMARKS NOTICE

- ASM-86 is a trademark of Digital Research
- CB-86 is a trademark of Digital Research
- CBASIC-86 is a trademark of Digital Research
- CLEO is a trademark of Phone 1 Inc.
- Concurrent CP/M-86 is a trademark of Digital Research
- Concurrent DOS is a trademark of Digital Research
- CP/M-86 is a trademark of Digital Research
- DDT-86 is a trademark of Digital Research
- Dr. Logo is a trademark of Digital Research
- ETHERNET is a trademark of Xerox Corp.
- GSX-86 is a trademark of Digital Research
- GW is a trademark of Microsoft Corp.
- IBM is a registered trademark of International Business Machines Corp.
- MICROSOFT is a registered trademark of Microsoft Corp.
- MS is a trademark of Microsoft Corp.
- OMNINET is a trademark of Corvus Systems Inc.
- p-System is a trademark of Softech Microsystem, Inc
- PC-DOS is a trademark of International Business Machines Corp.
- PEACHPAK is a trademark of Peachtree Software International Ltd.
- Personal Basic is a trademark of Digital Research
- SID-86 is a trademark of Digital Research
- UCSD and UCSD Pascal are registered trademarks of the Regent of the University of California
- UNIX is a trademark of Bell Laboratories
- Z80 is a registered trademark of Zilog Inc.
- Z8000 is a registered trademark of Zilog Inc.

CONTENTS

1. INTRODUCTION

GENERAL 1-1

SYSTEM CALLS 1-1

SYSTEM REQUIREMENTS 1-1

MEMORY REQUIREMENTS 1-1

DISK DRIVE(S) 1-2

MULTI-LINGUAL ENVIRONMENT 1-2

USING THIS MANUAL 1-3

DESCRIPTION OF SOFTWARE 1-3

2. ASSEMBLER OVERVIEW

OVERVIEW OF MACRO ASSEMBLER OPERATION 2-1

GENERAL 2-1

THE EDITOR 2-3

THE MACRO ASSEMBLER 2-4

MS-LINK (LINKER) AND MS-LIB (LIBRARY) 2-6

DEBUG UTILITY 2-7

FEATURES OF THE MACRO ASSEMBLER 2-8

RELOCATABLE OBJECT CODE 2-8

PROGRAM MODULES 2-8

MACRO INSTRUCTIONS 2-8

DIRECTIVE SYNTAX 2-13

3. THE MACRO ASSEMBLER SOURCE FILE

CREATING A MACRO ASSEMBLER SOURCE FILE 3-1

GENERAL FACTS ABOUT SOURCE FILES 3-1

NAMING YOUR SOURCE FILE 3-1

LEGAL CHARACTERS 3-2

NUMERIC NOTATION 3-2

WHAT'S IN A SOURCE FILE? 3-3

STATEMENT LINE FORMAT 3-4

NAMES 3-4

COMMENTS 3-6

ACTION 3-6

EXPRESSIONS 3-7

NAMES: LABELS, VARIABLES AND SYMBOLS 3-10

LABELS 3-10

VARIABLES 3-13

SYMBOLS 3-16

EXPRESSIONS: OPERANDS AND OPERATORS 3-18

MEMORY ORGANIZATION 3-18

OPERANDS 3-25

OPERATORS 3-32

PTR (Pointer) 3-34

: (colon) (Segment Override) 3-35

CONTENTS

SHORT 3-37

THIS 3-37

HIGH,LOW 3-39

OFFSET 3-41

TYPE 3-42

.TYPE 3-43

LENGTH 3-45

SIZE 3-46

Shift_count 3-48

WIDTH 3-49

MASK 3-50

4. INSTRUCTIONS AND DIRECTIVES

ACTION: INSTRUCTIONS AND DIRECTIVES 4-1

INSTRUCTIONS 4-1

DIRECTIVES 4-2

MEMORY DIRECTIVES 4-3

ASSUME 4-3

COMMENT 4-5

DB,DW,DD,DQ,DT (Define) 4-6

END 4-9

EQU 4-10

EQUAL SIGN 4-11

EVEN 4-12

EXTRN 4-13

GROUP 4-15

INCLUDE 4-17

LABEL 4-18

NAME 4-20

ORG 4-21

PROC 4-22

PUBLIC 4-24

.RADIX 4-25

RECORD 4-26

SEGMENT 4-29

STRUC 4-33

CONDITIONAL DIRECTIVES 4-35

MACRO DIRECTIVES 4-39

MACRO DEFINITION 4-40

CALLING A MACRO 4-41

ENDM (End Macro) 4-43

EXITM (Exit Macro) 4-44

LOCAL 4-45

PURGE 4-47

REPEAT 4-48

CONTENTS

IRP (Indefinite Repeat) 4-50

IRPC (Indefinite Repeat Character) 4-51

LISTING DIRECTIVES 4-55

PAGE 4-55

TITLE 4-57

SUBTITLE 4-58

%OUT 4-59

.LIST AND .XLIST 4-60

.SFCOND 4-61

.LFCOND 4-61

.TFCOND 4-61

.XALL 4-62

.LALL 4-62

.SALL 4-63

.CREF AND .XCREF 4-63

5. ASSEMBLING A MACRO ASSEMBLER SOURCE FILE

ASSEMBLING A MACRO ASSEMBLER SOURCE FILE 5-1

HOW TO START MACRO ASSEMBLER 5-1

METHOD 1: PROMPTS 5-1

METHOD 2: COMMAND LINE 5-2

MACRO ASSEMBLER COMMAND CHARACTERS 5-4

MACRO ASSEMBLER COMMAND PROMPTS 5-5

MACRO ASSEMBLER COMMAND SWITCHES 5-7

FORMATS OF LISTINGS AND SYMBOL TABLES 5-10

PROGRAM LISTING 5-11

DIFFERENCES BETWEEN PASS 1 AND PASS 2 LISTINGS 5-16

6. LIBRARY MANAGER (MS-LIB)

INTRODUCTION 6-1

OVERVIEW OF MS-LIB OPERATION 6-1

RUNNING MS-LIB 6-5

HOW TO START MS-LIB 6-5

COMMAND PROMPTS 6-9

COMMAND CHARACTERS 6-11

ERROR MESSAGES 6-14

7. MS-CREF (CROSS REFERENCE UTILITY)

INTRODUCTION 7-1

OVERVIEW OF MS-CREF 7-1

THE CROSS-REFERENCE FILE 7-1

RUNNING MS-CREF 7-2

HOW TO CREATE A CROSS-REFERENCE FILE 7-2

HOW TO START MS-CREF 7-3

METHOD 1: PROMPTS 7-3

METHOD 2: COMMAND LINE 7-5

CONTENTS

COMMAND CHARACTERS 7-6

FORMAT OF CROSS-REFERENCE LISTINGS 7-6

ERROR MESSAGES 7-8

FORMAT OF MS-CREF COMPATIBLE FILES 7-9

MS-CREF FILE PROCESSING 7-10

FORMAT OF SOURCE FILES 7-10

A. MACRO ASSEMBLER MESSAGES

OVERVIEW A-1

OPERATING MESSAGES A-1

ERROR MESSAGES A-1

I/O HANDLER ERRORS A-13

RUNTIME ERRORS A-14

NUMERICAL ORDER LIST OF ERROR MESSAGES A-15

B. ASCII CHARACTER CODES

C. TABLE OF MACRO ASSEMBLER DIRECTIVES

MEMORY DIRECTIVES C-1

MACRO DIRECTIVES C-2

CONDITIONAL DIRECTIVES C-2

LISTING DIRECTIVES C-2

ATTRIBUTE OPERATORS C-3

PRECEDENCE OF OPERATORS C-4

D. TABLE OF 8086 INSTRUCTIONS

8086 INSTRUCTION MNEMONICS (ALPHABETICAL) D-1

8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE D-4

E. SYNTAX NOTATION

SYNTAX NOTATION E-1

1. INTRODUCTION

ABOUT THIS CHAPTER

This chapter lists the Olivetti Personal Computer hardware requirements and provides a general introduction to the MS-Macro Assembler documentation. This includes an overview of the structure of the manual.

CONTENTS

GENERAL	**1-1**
SYSTEM CALLS	**1-1**
SYSTEM REQUIREMENTS	**1-1**
MEMORY REQUIREMENTS	**1-1**
DISK DRIVE(S)	**1-2**
MULTI-LINGUAL ENVIRONMENT	**1-2**
USING THIS MANUAL	**1-3**
DESCRIPTION OF SOFTWARE	**1-3**

GENERAL

The MS-DOS Macro Assembler is a powerful assembler for 8086 based computers. Macro Assembler runs under the MS-DOS operating system.

SYSTEM CALLS

System Calls are procedures used to interface with I/O or to manage memory. They can be accessed from utility programs written in assembly language, and from some high level languages. Their use frees the programmer from having to perform primitive functions, and makes it easier to write machine-independent programs.

MS-DOS provides two types of system calls: interrupts and function requests. The MS-DOS System Programmer Guide describes the environments from which these routines can be called, how to call them, and the processing performed by each.

SYSTEM REQUIREMENTS

MEMORY REQUIREMENTS

Each assembler utility requires different amounts of memory:

MASM - 96K bytes memory minimum
- 64K bytes for code and static data
- 32K bytes for run space

LINK - 50K bytes memory minimum
- 40K bytes for code
- 10K bytes for run space

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

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

DEBUG - Memory minimum is program dependent
- 13K bytes for code
- Run space depends on program size

Within these memory requirements, a standard system with 128K bytes of memory will be sufficient for most applications.

DISK DRIVE(S)

New BIOS and utilities released for MS-DOS support 160KB, 320KB and 640KB floppy disk drives, as well as the Hard Disk Unit (HDU).

Programs do not usually allow time to swap disks during operation on a one-drive system configuration, therefore two disk drives is a more practical configuration.

One disk drive may be used if and only if output is sent to the same physical drive from which the input was taken.

USING THIS MANUAL

Figure 1-1 is an overview of the documentation package for the MS-Macro Assembler. The various chapters of this package can be used as a set or individually. Each of the chapters identified in the figure is largely self-contained and will refer to the other chapters only at junctures in the software.

The overview of the Macro assembler, given in the next chapter (2), describes the flow of program development from creating a source file through program execution. The processes described in this overview are echoed and expanded in the specific chapters (3 through 5).

The chapters on the other utilities can be used as a set or independently. Each chapter is largely self-contained and will refer to other chapters only at junctures in the software.

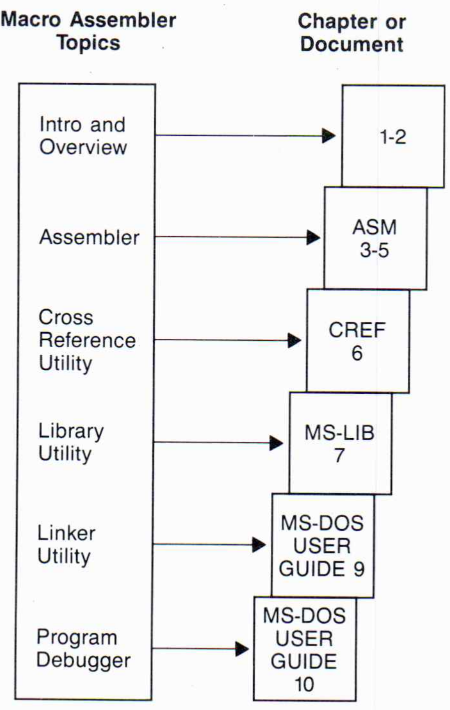


Fig. 1-1 Overview, MS-Macro Assembler User Guide

DESCRIPTION OF SOFTWARE

Macro Assembler Utility

The MS-Macro Assembler is a powerful assembler for 8086 based computers. It supports most of the directives found in Microsoft's Macro Assembler for the 8080. Macros and conditionals are Intel 8080 standard.

The MS-Macro Assembler is upward compatible with Intel's ASM-86, except Intel codemacros, macros, and a few directives.

The MS-Macro Assembler offers relaxed typing so that if you enter a typeless operand for an instruction that accepts only one type of operand, Macro Assembler 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 processes overlays that you define.

MS-LINK can perform multiple library searches, using a dictionary library search method. MS-LINK prompts you 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 your library of program files.

MS-LIB prompts you 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 Utility)

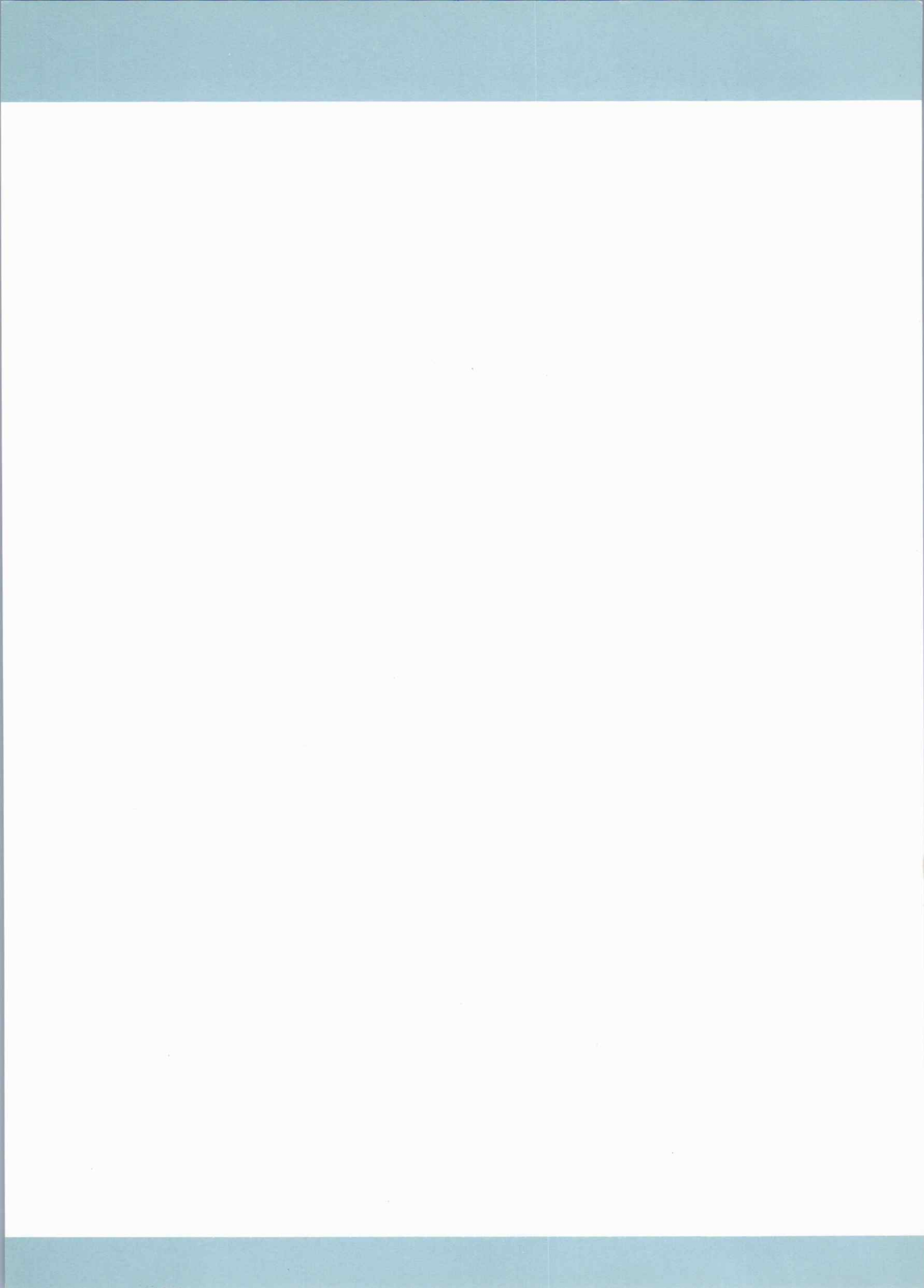
MS-CREF produces a cross-reference listing of all symbolic names in the Macro Assembler source program, giving both the source line number of the definition and the source line numbers of all other references to the symbols.

MS-DEBUG Utility

DEBUG provides a controlled testing environment for binary and executable object files.

DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change.

DEBUG allows you to alter the contents of a file or the contents of a CPU register, and then immediately reexecute a program to check on the validity of the changes.



2. ASSEMBLER OVERVIEW

ABOUT THIS CHAPTER

This chapter provides an overview of the macro assembler, its features, use and functions. It describes the assembly process, and the use of macro instructions. Figures are provided to illustrate the different operations.

CONTENTS

OVERVIEW OF MACRO ASSEMBLER OPERATION **2-1**

GENERAL **2-1**

THE EDITOR **2-3**

THE MACRO ASSEMBLER **2-4**

MS-LINK (LINKER) AND MS-LIB (LIBRARY) **2-6**

DEBUG UTILITY **2-7**

FEATURES OF THE MACRO ASSEMBLER **2-8**

RELOCATABLE OBJECT CODE **2-8**

PROGRAM MODULES **2-8**

MACRO INSTRUCTIONS **2-8**

DIRECTIVE SYNTAX **2-13**

OVERVIEW OF MACRO ASSEMBLER OPERATION

The MS-DOS Macro Assembler is a powerful assembler, incorporating many features usually found only in large computer assemblers (see "Features of the Macro Assembler", later in this chapter).

GENERAL

Figure 2-1 shows, in general terms, the steps required to develop a program. Each of these steps is then fully described in a later chapter or in other documentation:

1. Create — The first task is to create a source file. Use the EDLIN editor, which comes on the MS-DOS diskette, or any other editor available which is compatible with MS-DOS and the 8086 CPU. After writing the program, give the source filename the extension .ASM (Macro Assembler recognizes .ASM as the default).

2. Assemble — When the source file is ready, assemble it with the Macro Assembler as described in the chapter entitled: "Assembling a Macro Assembler Source File."

   The Macro Assembler outputs an assembled object file with the default filename extension .OBJ. During the assembling process, the assembler also outputs both status and error messages.

3. Link — Call the linker to make the file executable. This adds loading information, and creates the .EXE file.

4. Run — Run your assembled and linked program, the .EXE file. Use the MS-DOS operating system to execute your program. Use the DEBUG utility for help in correcting errors.

The Macro Assembler will create, on command, two files: a listing file and a cross-reference file. These receive the default extensions .LST and .CRF (see "Assembler Listings", later in this chapter).

It should be noted that it is also possible to assemble the source file without creating an .OBJ file. All the other assembly steps are performed, but (in response to prompts) the user chooses not to select listings and not to send object code to disk. However, error messages representing erroneous source statements are displayed on the terminal screen.

This practice may be useful in checking source code for errors. Modules can be test assembled quickly and errors corrected before the object code is put on disk.

Refer to Appendix A, "Macro Assembler Messages", for explanations of any messages displayed during or immediately after assembly.

THE EDITOR

Creating the source file involves creating instruction and directive statements that follow the rules and constraints described in Chapters 3-5 in this manual. An editor, such as EDLIN, is required for this task.

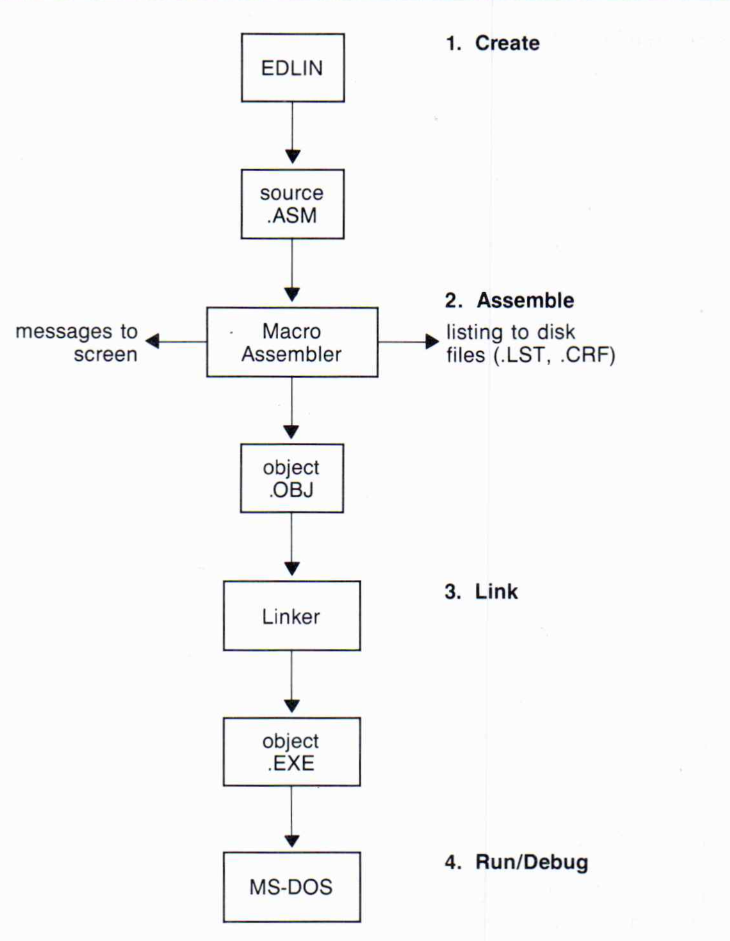


Fig. 2-1 Overview of Macro Assembler Operation

The EDLIN utility is located on the MS-DOS System diskette. It provides an easy and convenient way to create and modify source files. This includes special function keys for editing and corrections.

For a complete description of the EDLIN editor, see the MS-DOS User Guide.

THE MACRO ASSEMBLER

The Macro Assembler (see Figure 2-2) 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
- 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, label, and expression values from the symbol table
- expands macro call statements
- emits the relocatable object code into a file with the default filename extension .OBJ

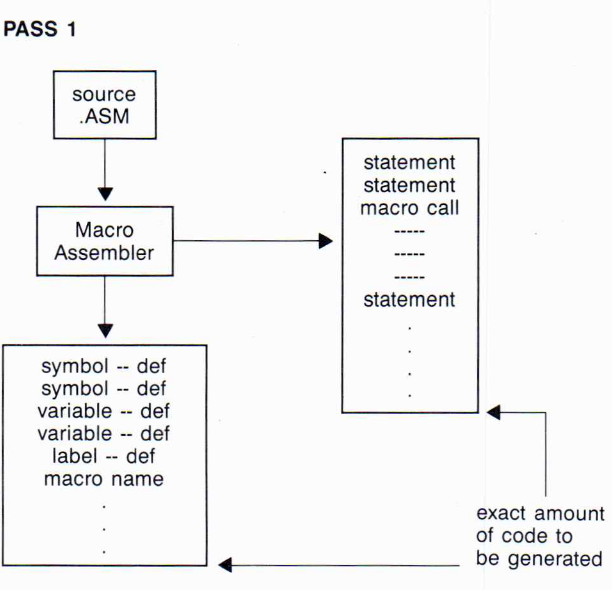
Assembler Listings

The MS-Macro Assembler (see Figure 2-3) creates, on optional command, two types of listing file:

- A normal listing file
- A cross-reference file.

The normal 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. The listing also 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.

PASS 1



PASS 2

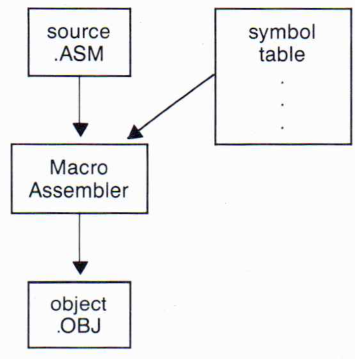


Fig. 2-2 Pass 1 and Pass 2

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 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.

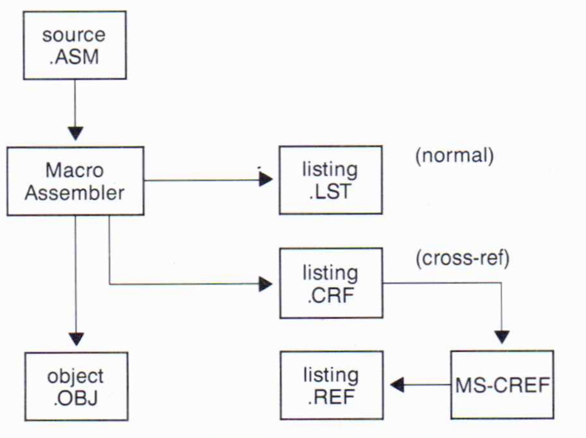


Fig. 2-3 Files That The Macro Assembler Produces

MS-LINK (LINKER) AND MS-LIB (LIBRARY)

The linker produces an executable object file with the default filename extension .EXE.

If the .OBJ file is stored as part of the user's library of object programs (see Figure 2-4), MS-LINK can also be used to link one or more .OBJ modules and produce a single executable object file (refer to the MS-LINK utility for further explanation and instructions).

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

DEBUG UTILITY

If, when running your assembled and linked program, the .EXE file, your program does not run properly, use the DEBUG utility to locate any errors.

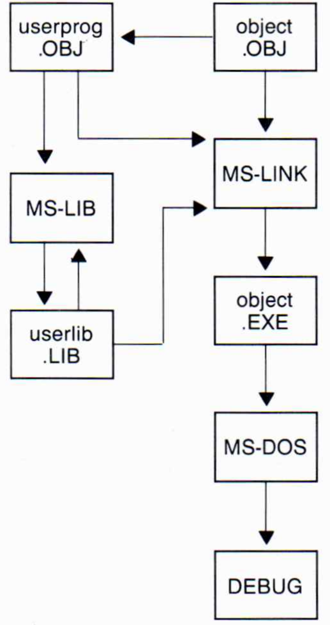


Fig. 2-4 Linker, Library and Debug Utilities

FEATURES OF THE MACRO ASSEMBLER

The MS-DOS Macro Assembler is a very powerful assembler and 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 the CPU (the 8086 microprocessor). Although the Macro Assembler is more complex than any other microcomputer assembler, it is also easy to use.

RELOCATABLE OBJECT CODE

The Macro Assembler produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked using the MS-LINK 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, instead of in some fixed range of memory addresses.

PROGRAM MODULES

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 are 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 assembly process is shown in Figure 2-5.

MACRO INSTRUCTIONS

The Macro Assembler permits the writing of blocks of code for a set of frequently used instructions. The need for recoding these instructions each time they are required in the program is thus eliminated.

Such blocks of code are called macros. The instructions are the macro definition. Each time the set of instructions is needed, only a simple "call" to a macro is placed in the source file. The Macro Assembler expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to

the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are given only once; other occurrences are one-line calls.

Macros can be "nested," that is, a macro can be called from inside another macro block. Nesting of macros is limited only by memory.

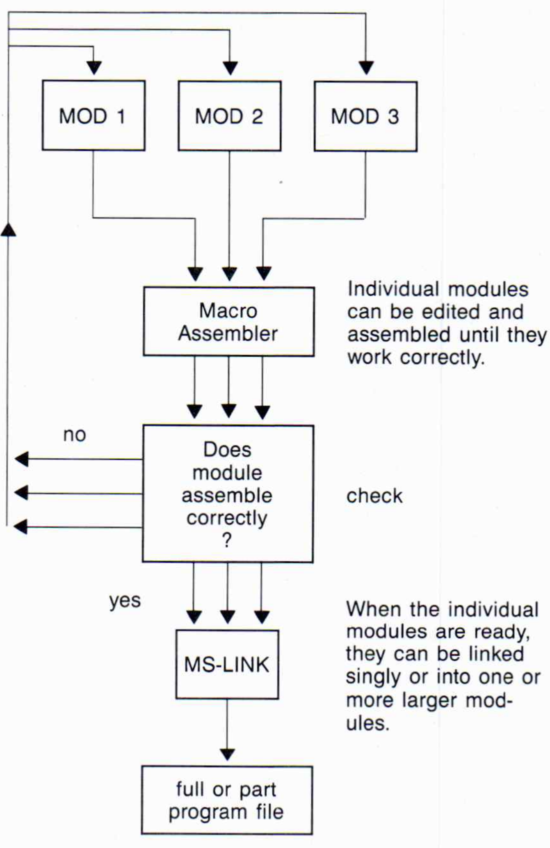


Fig. 2-5 The Assembly Process

The macro facility includes repeat, indefinite repeat, and indefinite repeat character directives for programming repeat block operations. The MACRO directive can also be used to alter the action of any instruction or directive by using its name as the macro name. When any instruction or directive statement is placed in the program, Macro Assembler first checks the symbol table it created to see if the instruction or directive is a macro name. If it is, Macro Assembler "expands" the macro call statement by replacing it with the body of instructions in the macro's definition. If the name is not defined as a macro, Macro Assembler tries to match the name with an instruction or directive. The MACRO directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.

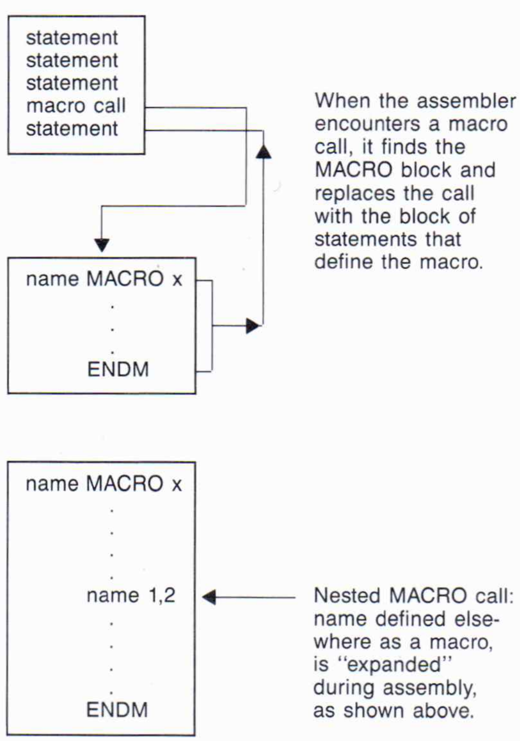
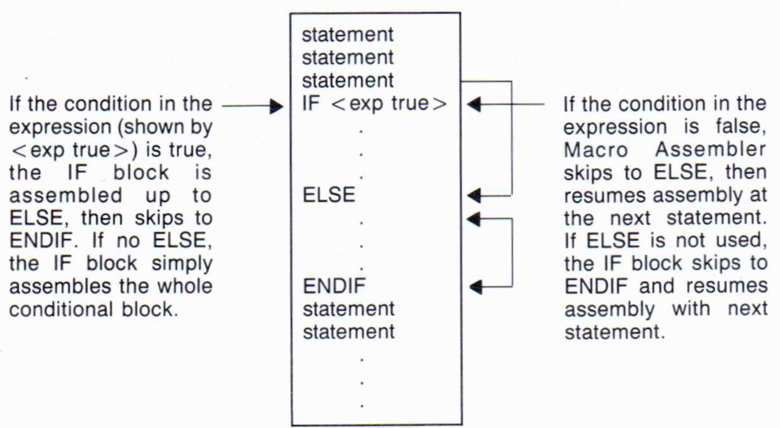
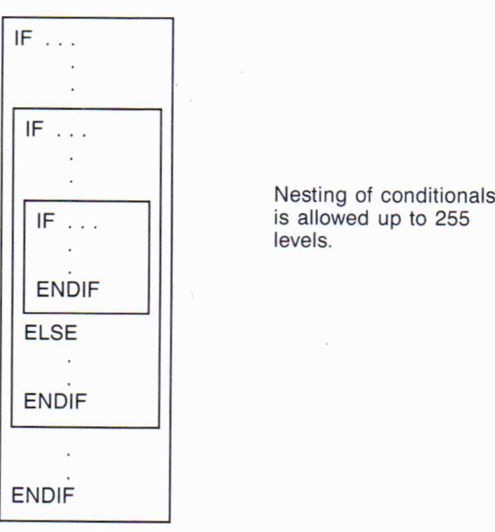


Fig. 2-6 Assembler Macros

The Macro Assembler also 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. The Macro Assembler can test for blank or nonblank arguments, for defined or undefined symbols, for equivalence, for first assembly pass or second, and 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.

The Macro Assembler's conditional assembly facility also supports the nesting of conditionals. Figure 2-7 illustrates the support of conditional assembly blocks. Such blocks can be nested up to 255 levels.



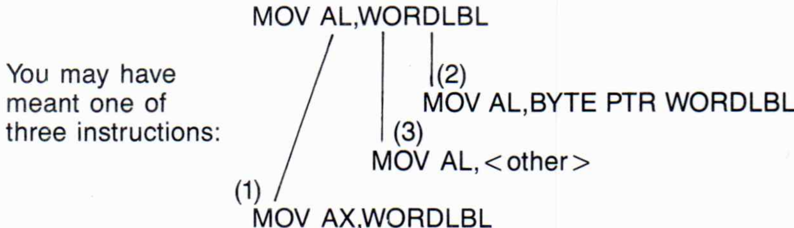


Nesting of conditionals is allowed up to 255 levels.

Fig. 2-7 Conditional Statements

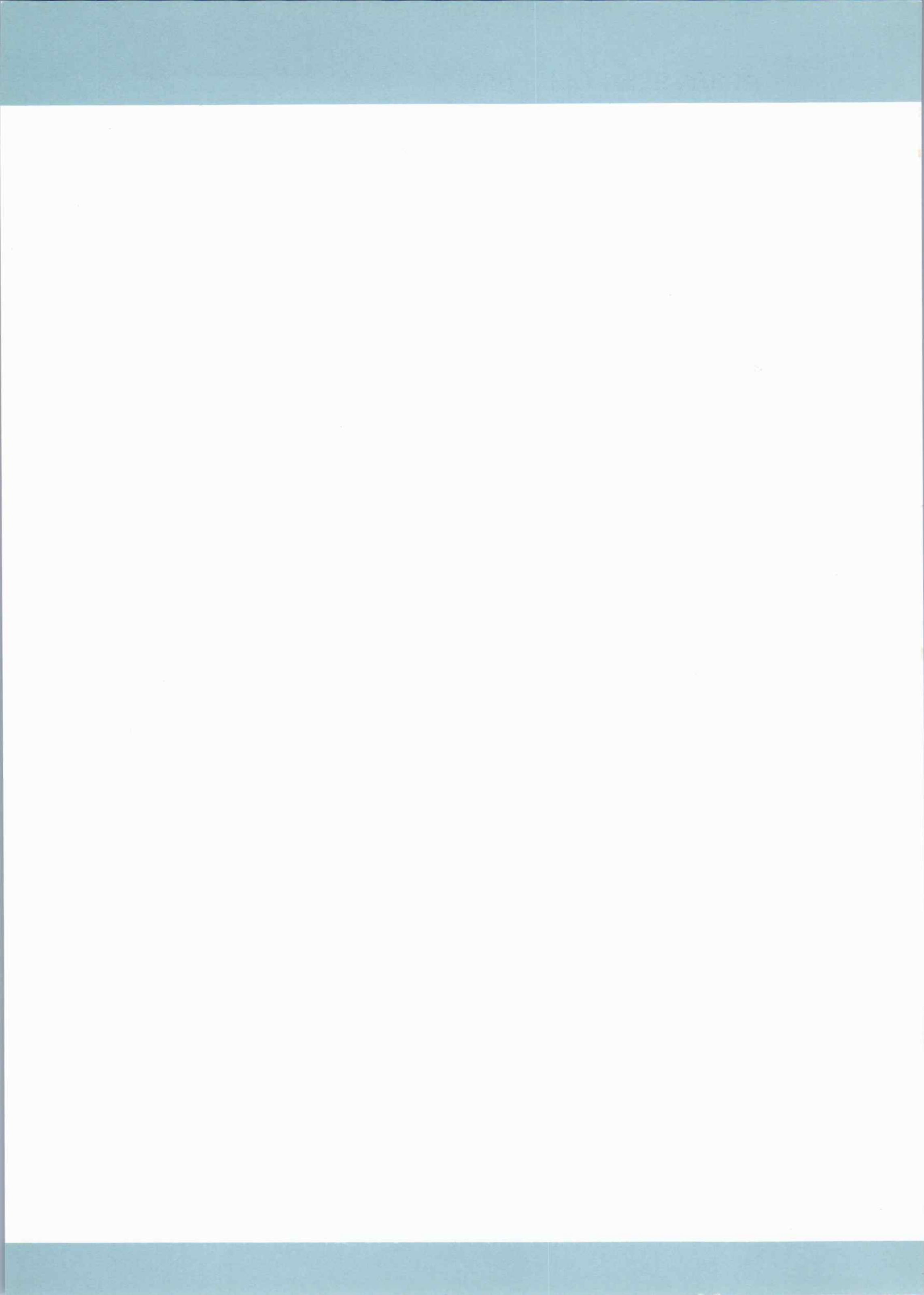
DIRECTIVE SYNTAX

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 would be wasteful to return an error for a lapse of memory or a typographical error. When the wrong type choice is given, the Macro Assembler displays an error message but generates the "correct" code. That is, it always outputs instructions, not just NOP instructions. For example, if you enter:



The Macro Assembler generates instruction (2) because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size". The Macro Assembler accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some mundane debugging chores. An error message is still returned, however, because you may have mis-stated the operand the Macro Assembler assumes is "correct".

For a detailed discussion of Directives, see Chapter 4.



3. THE MACRO ASSEMBLER SOURCE FILE

ABOUT THIS CHAPTER

The first part of this chapter summarizes the steps you will take to create a source file. It discusses the statement format, and describes the parts of a statement: Names, Comments, Actions, and Expressions.

Next is a more detailed discussion of Names: Labels, Variables, and Symbols.

The rest of this chapter describes Expressions: Operands and Operators. Together with the following chapter on Actions, this completes the basic information required to create a source file.

CONTENTS

CREATING A MACRO ASSEMBLER SOURCE FILE 3-1

GENERAL FACTS ABOUT SOURCE FILES 3-1

NAMING YOUR SOURCE FILE 3-1

LEGAL CHARACTERS 3-1

NUMERIC NOTATION 3-2

WHAT'S IN A SOURCE FILE? 3-3

STATEMENT LINE FORMAT 3-4

NAMES 3-4

COMMENTS 3-6

ACTION 3-6

EXPRESSIONS 3-7

NAMES: LABELS, VARIABLES AND SYMBOLS 3-10

LABELS 3-10

VARIABLES 3-13

SYMBOLS 3-16

EXPRESSIONS: OPERANDS AND OPERATORS 3-18

MEMORY ORGANIZATION 3-18

OPERANDS 3-25

OPERATORS 3-32

PTR (Pointer) 3-34

:(colon) (Segment Override) 3-35

SHORT 3-37

THIS 3-37

HIGH,LOW 3-39

OFFSET 3-41

TYPE	**3-42**
.TYPE	**3-43**
LENGTH	**3-45**
SIZE	**3-46**
Shift_count	**3-48**
WIDTH	**3-49**
MASK	**3-50**

CREATING A MACRO ASSEMBLER SOURCE FILE

To create a source file for Macro Assembler, you need to use an editor program, such as EDLIN. 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 when creating the file.

This chapter discusses the statement format and introduces descriptions of its components. You will find full descriptions of names: variables, labels, and symbols. It also provides full descriptions of expressions and their components, operands and operators.

GENERAL FACTS ABOUT SOURCE FILES

NAMING YOUR SOURCE FILE

When you create a source file, you must name it. A filename may be any name that is legal for your operating system. When you run Macro Assembler to assemble your source file, Macro Assembler assumes that your source filename has the extension ".ASM".

You do not need to give your source filename the ".ASM" extension. However, if your source filename has an extension other than ".ASM", you must specify the extension name when you run Macro Assembler. (You do not need to specify the ".ASM" extension if your source filename has an extension of ".ASM". Macro Assembler will supply the default extension for you).

Note that Macro Assembler gives the object file which it outputs the default extension ".OBJ". To avoid confusion or the destruction of your source file, you should avoid giving a source file an extension of ".OBJ". For similar reasons, you should also 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 indicate that the value in the register is an address and not data
( )	(parentheses) operator in DUP expressions and operator to change precedence of operator evaluation.

NUMERIC NOTATION

The default input radix for all numeric values is decimal. The output radix for all listings is hexadecimal for code and data items and decimal for line numbers. The output radix can only be changed to octal radix by giving the /O switch when Macro Assembler is run (see "Macro Assembler Command Switches" in Chapter 5). There are two ways to change the input radix:

1. With the .RADIX directive (see "Memory Directives" in Chapter 4)

2. By special notation appended to a numeric value.

Radix	Range	Notation	Example
Binary	0-1	B	01110100B
Octal	0-7	Q or O	735Q or 621O
Decimal	0-9	none or D	9384 (default) 8149D*
Hexadecimal	0-9 A-F	H	OFFH or 80H**

Tab. 3-1 Numeric Notation

* When .RADIX directive changes from the decimal default radix.
** First character must be numeral from 0-9.

WHAT'S IN A SOURCE FILE?

A source file for Macro Assembler consists of instruction statements and directive statements. Instruction statements are made of 8086 instruction mnemonics and their operands, which command specific processes directly to the 8086 processor. Directive statements are commands to Macro Assembler to prepare data for use in and by instructions.

Statement line format is described in the following section.

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 Assembler where program execution should begin.

"Memory Organization" later in this chapter, describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another and to your programming as a whole. This information, helpful for developing your programs, is presented as a prelude to the discussion of operands and operators.

STATEMENT LINE FORMAT

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

Macro Assembler directive statements consist of four "fields": Name, Action, Expression, Comment.

For example:

```
ARG     DB       0D5E          ;create variable ARG containing the
 |       |        |             |                       ;value 0D5EH
Name    Action   Expression    ;Comment
```

Macro Assembler instruction statements usually consist of three "fields": Action, Expression, Comment. For example:

```
        MOV      CX,ARG        ;here's the count number
         |        |             |
        Action   Expression    ;Comment
```

An instruction statement may have a Name field under certain circumstances; see the following discussion.

NAMES

The name field, when present, is the first entry on the statement line. The name may begin in any column. Normally names start in column 1.

Names may be any length you choose. Macro Assembler considers only the first 31 characters significant when your source file is assembled.

One other significant use for names is with the MACRO directive. Although all the rules covering names apply to MACRO names, the discussion of macro names is better left to the section describing the macro facility.

Macro Assembler supports the use of names in a statement line for three purposes: to represent code, data, and constants.

To make a name represent code, use:

name: [*directive* | *instruction*]

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*
size must be BYTE, WORD, etc.

name {**DB** | **DW** | **DD** | **DQ** | **DT**} *expression*

EXTRN NAME: *size*
size must be 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 described in "Memory Directives" in Chapter 4).

Comments document the processing that is supposed to happen at a particular point in a program. When comments are used in this manner, they can be useful for debugging, for altering code, or for updating code. Consider putting comments at the beginning of each segment, procedure, structure, module, and after each line in the code that begins a step in the processing.

Comments are ignored by Macro Assembler. Comments do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code.

ACTION

The action field contains either an 8086 instruction mnemonic or a Macro Assembler assembler directive. The Macro Assembler directives are described in detail in Chapter 4.

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 in any column, 1 through maximum line length (minus columns for action and expression).

The entry in the action field either directs the processor to perform a specific function or it directs the assembler to perform one of its functions. Instructions tell the processor to perform some action. 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:

opcode		operand		data		data
opcode		operand		addr		addr
supplied		supplied		or		found

supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements. (Opcode 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 others take two. For two-operand instructions, the expression field consists of a destination operand and a source operand, in that order, separated by a comma. For example:

opcode		dest-operand	,	source-operand

For one-operand instructions, the operand is a source or a destination operand, depending on the instruction. If one or both of the operands is omitted, the instruction carries that information in its internal coding.

Source operands are immediate operands, register operands, memory operands, or attribute operands. Destination operands are register operands and memory operands.

For directives, the expression field usually consists of a single operand. For example:

directive		operand

A directive operand is 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 operands. Use of a complex operand permits you to specify addresses or data derived from several places. For example:

```
MOV ARG[BX],AL
```

The destination operand is the result of adding the address represented by the variable ARG and the address found in register BX. The processor is instructed to move the value in register AL to the destination calculated from these two operand elements.
Another example:

```
MOV AX,ARG+5[BX]
```

In this case, the source operand is the result of adding the value represented by the symbol ARG plus 5 plus the value found in the BX register.

Macro Assembler supports the following operands and operators in the expression field (shown in order of precedence):

OPERANDS	OPERATORS
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	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

Tab. 3-2 Macro Assembler Operands and Operators

Note

Some operators can be used as operands or as part of an operand expression. Refer to "Operands" and "Operators" later in this chapter for details.

NAMES: LABELS, VARIABLES AND SYMBOLS

Names are used in several ways throuthout Macro Assembler, wherever any naming is allowed or required.

Names are symbolic representations of values. The values may be addresses, data, or constants.

Names may be any length you choose. However, Macro Assembler will truncate names longer than 31 characters when your source file is assembled.

Names may be defined and used in a number of ways. This section introduces you to the basic ways to define and use names. You will discover additional uses as you study the section on Expressions later in this chapter and Chapter 4 "Instructions and Directives", and as you use Macro Assembler.

Macro Assembler supports three types of names in statements lines: labels, variables, and symbols. This section covers how to define and use these three types of names.

LABELS

Labels are names used as targets for JMP, CALL, and LOOP instructions. Macro Assembler assigns an address to each label as it is defined. When you use a label as an operand for JMP, CALL, or LOOP, Macro Assembler can substitute the attributes of the label for the label name, sending processing to the appropriate place.

Labels are defined in one of the four ways shown in the following table.

LABEL DEFINITION	MEANING
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 which allow a name field. *name*: may also be placed on a line by itself. Examples CLEAR_SCREEN: MOV AL,20H EN: DB OFH SUBROUTINE3:
name LABEL NEAR *name* LABEL FAR	Use the LABEL directive (see "Memory Directives" in Chapter 4). NEAR and FAR are discussed under the Type Attribute below. Examples NAM LABEL NEAR GAM LABEL FAR
name PROC NEAR *name* PROC FAR	Use the PROC directive, discussed in the section on 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

LABEL DEFINITION	MEANING
EXTRN *name*:NEAR EXTRN *name*:FAR	Use the EXTRN directive, discussed in "Memory Directives" in Chapter 4. NEAR and FAR are discussed under the Type Attribute below. Examples EXTRN ARG: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.

ATTRIBUTE	MEANING
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 Assembler 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.

ATTRIBUTE	MEANING
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 have the same CS ASSUME. FAR labels are used for references from segments with a different CS ASSUME, or when there are more than 64K bytes between the label reference and the label definition. NEAR and FAR cause Macro Assembler to generate slightly different code. NEAR labels supply their offset attribute only (a 2-byte pointer). FAR labels supply both segment and offset attributes (a 4-byte pointer).

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 alike in some ways. The differences are important.

Variables are defined in one of the three ways shown below.

VARIABLE DEFINITION	MEANING
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 10DUP(<'Z'>) Note that SMALL will have the same size as the record GARAGE. See the DEFINE, STRUC, and RECORD directives in Chapter 4 "Instructions and Directives".

VARIABLE DEFINITION	MEANING
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 Chapter 4 "Instructions and Directives".
EXTRN *name*: *size*	Use the EXTRN directive with one of the size specifiers described above. See EXTRN directive in Chapter 4 "Instructions and Directives". Example EXTRN ARG:DWORD

Variables also have three attributes: segment, offset, and type (as do labels).

Segment and Offset are the same for variables as for labels. The Type attribute is different.

ATTRIBUTE	MEANING
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.

The size of the *type* attribute is defined in the following table:

DIRECTIVE	TYPE	SIZE
DB	BYTE	1 byte
DW	WORD	2 bytes
DD	DWORD	4 bytes
DQ	QWORD	8 bytes
DT	TBYTE	10 bytes

Tab. 3-3 Variable Type Attributes: Comparative Sizes

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 in the three ways shown below.

SYMBOL DEFINITION	MEANING
name EQU *expression*	Use the EQU directive, described in Chapter 4 "Instructions and Directives". *expression* may be another symbol, an instruction mnemonic, a valid expression or any other entry (such as text or indexed references). Examples ARG EQU 7H ZOO EQU ARG
name = *expression*	Use the equal sign directive, described in "Memory Directives" in Chapter 4. *expression* may be any valid expression. Examples GOO = 0FH GOO = $+2 GOO = GOO+ARG
EXTRN *name:ABS*	Use the EXTRN directive with type ABS. See EXTRN directive in Chapter 4 "Instructions and Directives". Example EXTRN BAN:ABS BAN must be defined by an EQU or = directive to a valid expression.

EXPRESSIONS: OPERANDS AND OPERATORS

The term "expression" is 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 looks like a mathematical expression.

This section describes the types of operands and operators that Macro Assembler supports. The discussion of memory organization in a Macro Assembler program acts as a preface to the descriptions of operands and operators, and as a link to topics discussed in "MS-DOS User Guide", Chapter 9, "The Linker".

MEMORY ORGANIZATION

Most of your assembly language program is written in segments. In the source file, a segment is a block of code that begins with a SEGMENT directive statement and ends with an ENDS directive. In an assembled and linked file, a segment is any block of code that is addressed through the same segment register and is not more than 64K bytes long.

You should note that Macro Assembler leaves everything relating to segments to MS-LINK. MS-LINK resolves all references. For that reason, Macro Assembler does not check (because it cannot) to see if your references are entered with the correct distance type. Values such as OFFSET are also left to MS-LINK 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.)

When the modules are linked together, the several segments become one. References to labels, variables, and symbols within each module acquire the offset from the beginning of the whole segment, not just from the beginning of their portion of the whole segment. (All divisions are removed.)

You have the option of grouping several segments into a group using the GROUP directive. When you group segments, you tell Macro Assembler that you want to be able to refer to all of these segments as a single entity. (This does not eliminate segment identity, nor does it make values within a particular segment less immediately accessible. It does make values relative to a group base). The advantage of grouping is that you can refer to data items without worrying about segment overrides or changing segment registers.

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 Assembler to generate an immediate instruction; that is, to use the address of the value instead of the value itself. There are two kinds of 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 Assembler first looks for the segment of the reference. Macro Assembler scans the segment registers for the SEGMENT of the reference, then the GROUP (if any) of the reference.

However, the use of the OFFSET operator always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must override this restriction by using the GROUP name and the colon operator. For example:

MOV AX,OFFSET *group-name:ref*

If you set a segment register to a group with the ASSUME directive, then you may also override the restriction on OFFSET by using the register name. For example:

MOV AX,OFFSET DS: *ref*

The result of both of these statements is the same.

Code labels have four attributes:

1. Segment - what segment the label belongs to
2. Offset - the number of bytes from the beginning of its segment
3. Type - NEAR or FAR
4. CS ASSUME - the CS ASSUME the label was coded under

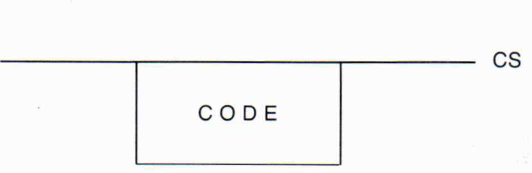
When you enter a NEAR JMP or NEAR CALL, you are changing the offset (IP) in CS. Macro Assembler compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they are different, Macro Assembler returns an error (you must use a FAR JMP or FAR CALL). When you enter a 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 in the following figure.



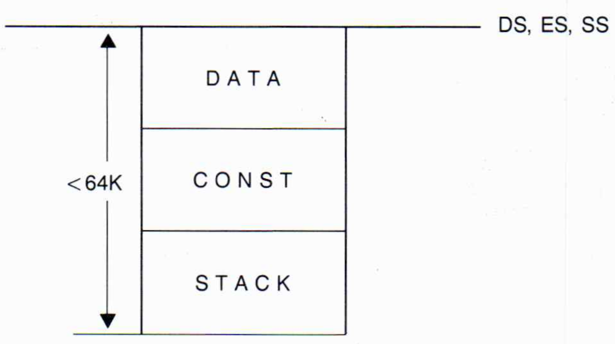


Fig. 3-4 Memory Organization

Given this arrangement, a statement like

MOV AX, *variable*

causes Macro Assembler to find the best segment register to reach this variable. (The "best" register is the one that requires no segment overrides).

A statement like

MOV AX,OFFSET *variable*

tells Macro Assembler to return the offset of the variable relative to the beginning of the variable's segment.

If this *variable* is in the CONST segment and you want to reference its offset from the beginning of DGROUP, you need a statement like the following:

MOV AX,OFFSET DGROUP: *variable*

Macro Assembler is a two-pass assembler. During pass 1, it builds a symbol table and calculates how much code is generated, but does not produce object code. If undefined items are found (including forward references), 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 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 Assembler returns a phase error.

Because pass 1 must keep correct track of the relative offset, some references must be known on pass 1. If they are not known, the relative offset will not be correct.

The following references must be known on pass 1:

- IF/IFE *expression*
  If *expression* is not known on pass 1, Macro Assembler does not know to assemble the conditional block (or which part to assemble if ELSE is used). On pass 2, the assembler would know and would assemble, resulting in a phase error.

- *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.

- .RADIX *expression*
  Because this directive changes the input radix, constants could have a different value, which could cause Macro Assembler to evaluate IF or DUP statements incorrectly.

The biggest problem for the assembler is handling forward references. How can it know the kind of a reference when it still has not seen the definition? This is one of the main reasons for two passes. And, unless Macro Assembler can tell from the statement containing the forward reference what the size, the distance, or any other of its attributes are, the assembler can only take the safe route (generate the largest possible instruction in some cases, except for segment override or FAR). This results in extra code that does nothing. (Macro Assembler figures this out by pass 2, but it cannot reduce the size of the instructions without causing an error, so it puts out NOP instructions (90H).

For this reason, Macro Assembler includes a number of operators to help the assembler. These operators tell Macro Assembler what size instruction to generate when it is faced with an ambiguous choice. As a benefit, you can also reduce the size of your program by using these operators to change the nature of the arguments to the instructions.

Examples

MOV AX,ARG ;ARG = forward constant

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

MOV AX,OFFSET ARG ;OFFSET says use the address of ARG

Because OFFSET tells Macro Assembler to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code.

Similarly, if you have a CALL statement that calls to a label that may be in a different CS ASSUME, you can prevent problems by attaching the PTR operator to the label:

CALL FAR PTR *forward-label*

At the opposite extreme, you may have a JMP forward that is less than 127 bytes. You can save yourself a byte if you use the SHORT operator.

JMP SHORT *forward-label*

However, you must be sure that the target is indeed within 127 bytes or Macro Assembler will not find it.

The PTR operator can be used another way to save yourself a byte when using forward references. If you defined ARG as a forward constant, you might enter the statement:

MOV [BX],ARG

You may want to refer to ARG as a byte immediate. In this case, you could enter either of these statements (they are equivalent):

MOV BYTE PTR [BX],ARG

MOV [BX],BYTE PTR ARG

These statements tell Macro Assembler that ARG is a byte immediate. A smaller instruction is generated.

OPERANDS

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

The following list shows all the types and the items that comprise them:

- Immediate operands
  - Data items
  - Symbols
- Register operands
- Memory operands
  - Direct
    - Labels
    - Variables
    - Offset (fieldname)
  - Indexed
    - Base register
    - Index register
    - [constant]
    - +/- displacement
  - Structure

Immediate Operands

Immediate operands are constant values that you supply when you type a statement line. The value may be typed 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

Macro Assembler recognizes values in forms other than decimal when special notation is appended. The default input radix is decimal. Any numeric values entered without numeric notation appended will be treated as a decimal value. These other values include ASCII characters as well as numeric values.

DATA FORM	FORMAT	EXAMPLE
Binary	xxxxxxxxB	01110001B
Octal	xxxO xxxQ	735O (letter O) 412Q
Decimal	xxxxx xxxxxD	65535 (default) 1000D (when .RADIX changes input radix to nondecimal)
Hexadecimal	xxxxH	0FFFFH (1st digit must be 0-9)
ASCII	'xx' "xx"	'OM' (more than two with DB only;) "OM" both forms are synonymous
10 real	xx.xxE&+xx	25.23E-7 (floating point format)
16 real	x...xR	8F76DEA9R (1st 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).

Tab. 3-5 Data Item Notation

Symbol Constants

Symbol names equated with some form of constant information (see the discussion under Names, earlier in this chapter) 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 type:

```
MOV AX,ARG
```

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

```
ARG EQU 9
```

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.

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 Assembler that an operand is in a different segment than specified in an ASSUME statement. (See the segment override operator in the section on "Attribute Operators").

The flag register is one 16-bit register containing eight 1-bit flags (five 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:

1-bit arithmetic flags		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 below).

Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct Macro Assembler to an address to find some data or instruction.

A memory operand always consists of an offset from a base address.

Memory operands fit into three categories: those that do not use a register (direct memory operands), those that use a base or index register (indexed memory operands), and structure operands.

Direct Memory Operands

Direct memory operands do not use a register, 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,ARG
MOV ARG,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, ARG[5]). The square brackets are treated like plus signs (+). Therefore:

ARG[5] is equivalent to ARG+5
5[ARG] is equivalent to 5+ARG

The only difference between square brackets and plus signs occurs when a register name appears inside the square brackets. Then, the operand is indexed.

The types of indexed memory operands are:

Base registers: [BX] [BP]

BP has SS as its default segment register;
all others have DS as default.

Index registers: [DI] [SI]

[constant]	Immediate in square brackets [8], [ARG]
+/- Displacement	8-bit or 16-bit value. Used only with another indexed operand.

These elements may be combined in any order. The only restriction is that two base registers and two indexed registers cannot 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[ARG]-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*.

Where

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.

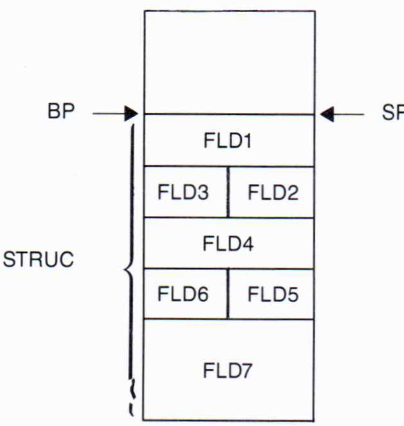
The period (.) must be included.

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 field name indexed through BP; for example:

```
[BP].FLD6
```



This method makes all values on the stack available all the time, not just the value at the top. Therefore, this method makes the stack a handy place to pass parameters to subroutines.

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.

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.

PTR (Pointer)

The PTR operator overrides the type (BYTE, WORD, DWORD) or the distance (NEAR, FAR) of an operand.

---

attribute **PTR** *expression*

Where

attribute is the new attribute; the new type or new distance.

expression is the operand whose attribute is to be overridden.

The most important and frequent use for PTR is to assure that Macro Assembler understands what attribute the expression is 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 the section on "Memory Directives" in Chapter 4.

Examples

```
CALL WORD PTR [BX][SI]
MOV BYTE PTR ARRAY

ADD BYTE PTR ARG,9
```

: (colon) (Segment Override)

The segment override operator overrides the assumed segment of an address expression (which may be a label, a variable, or other memory operand).

Syntax 1:

segment-register **:** *address-expression*

Syntax 2:

segment-name **:** *address-expression*

Syntax 3:

group-name **:** *address-expression*

Where

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.

Remarks

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

Macro Assembler assumes that labels are addressable through the current CS register. Macro Assembler also 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 Assembler through the ASSUME directive, you will need to use a segment override operator. Also, if you want to use a non-default relative base (that is, not the default segment register), you will need to use the segment override operator for forward references. Note that if Macro Assembler can reach an operand through a non-default segment register, it will use it, but the reference cannot be forward in this case.

Examples

```
MOV AX,ES:[BX+SI]
MOV CSEG:FAR_LABEL,AX
MOV AX,OFFSET DGROUP:VARIABLE
```

SHORT

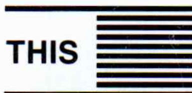
SHORT overrides NEAR distance attributes of labels used as targets for the JMP instruction. SHORT tells Macro Assembler that the distance between the JMP statement and the *label* specified as its operand is not more than 127 bytes in either direction.

SHORT *label*

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

The THIS operator creates an operand. The value of the operand depends on which argument you give.

Syntax 1:

THIS *distance*

Syntax 2:

THIS *type*

The argument to THIS may be:

- A distance (NEAR or FAR)
- A type (BYTE, WORD, or DWORD)

Where

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 and LOW are provided for 8080 assembly language compatibility. HIGH and LOW are byte isolation operators.

Syntax 1:

HIGH *expression*

Syntax 2:

LOW *expression*

Where

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 OFFFFH
```

Value Returning Operators

These operators return the attribute values of the operands that follow them but do not override the attributes.

The value returning operators take labels and variables as their arguments.

Because variables in Macro Assembler have three attributes, you need to use 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 returns the segment value (segment base address) of the segment enclosing the label or variable.

Syntax 1:

SEG *label*

Syntax 2:

SEG *variable*

Example

MOV AX,SEG VARIABLE_NAME
MOV AX, *segment-variable*: *variable*

OFFSET

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).

Syntax 1:

OFFSET *label*

Syntax 2:

OFFSET *variable*

Remarks

OFFSET is chiefly used to tell the assembler that the operand is an immediate operand.

OFFSET does not make the value a constant. Only MS-LINK can resolve the final value.

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 ARG
```

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 ROS is defined in a segment placed in DGROUP, and you want the offset of ROS in the group, you need to enter a statement like:

```
MOV BX,OFFSET DGROUP:ROS
```

You must be sure that the GROUP directive precedes any reference to a group name, including its use with OFFSET.

TYPE

The TYPE Operator returns the number of bytes of the variable type, if the operand is a variable; if the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Syntax 1:

TYPE *label*

Syntax 2:

TYPE *variable*

If the operand is a variable, the following values are returned:

```
BYTE   = 1
WORD   = 2
DWORD  = 4
QWORD  = 8
TBYTE  = 10
STRUC  = the number of bytes declared by STRUC
```

Example

```
MOV AX,(TYPE NAR_GER) PTR [BX+SI]
```

.TYPE

The .TYPE operator returns a byte that describes two characteristics of the *variable*: the mode, and 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.

.TYPE *variable*

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
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

```
ARG     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 returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for the variable that constitutes its argument.

LENGTH *variable*

Remarks

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

```
ARG DW 100 DUP(1)
MOV CX,LENGTH ARG ;get number of elements
                  ;in array
                  ;LENGTH returns 100
```

```
BAN DW 100 DUP(1,10 DUP(?))
```

LENGTH BAN is still 100, regardless of the expression following DUP.

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE

Where

SIZE returns the total number of bytes allocated for a variable.

SIZE *variable*

SIZE is the product of the value of LENGTH times the value of TYPE.

Example

```
ARG DW 100 DUP(1)
MOV BX,SIZE ARG   ;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 are used to isolate fields in a record.

Records are defined by the RECORD directive (see "Memory Directives" in Chapter 4). 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:

OPERATOR	MEANING
Shift count	Number of bits from low end of record to low end of field (number of bits to right shift the record to lowest bits of record).
WIDTH	The number of bits wide the field or record is (number of bits the field or record contains).
MASK	Value of record if field contains its maximum value and all other fields are zero (all bits in field contain 1; all other bits contain 0).

In the following discussions of the record specific operators, these symbols are used:

SYMBOL	MEANING
ARG	a record defined by the RECORD directive ARG RECORD FIELD1:3,FIELD2:6,FIELD3:7.
BAN	a variable used to allocate ARG BAN ARG.
FIELD1, FIELD2, FIELD3	are the fields of the record ARG.

Shift_count

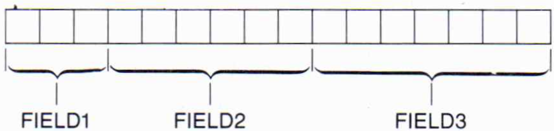
The shift count is derived from the record fieldname to be isolated.

---

record-fieldname

The shift count is the number of bits the field must be shifted right to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (ARG) 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,BAN
MOV CL,FIELD2
SHR DX,CL
```

FIELD2 is now right-shifted, ready for access.

WIDTH

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.

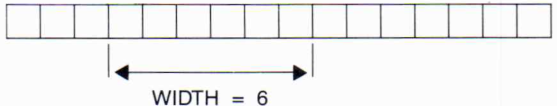
Syntax 1:

WIDTH *record-fieldname*

Syntax 2:

WIDTH *record*

Using the diagram under shift count, WIDTH can be diagrammed as:



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.

MASK

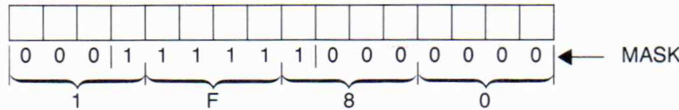
MASK returns a bit-mask defined by 1 for bit positions included by the field and 0 for bit positions not included. The value returned represents the maximum value for the record when the field is masked.

MASK *record-fieldname*

Where

MASK accepts a field name as its only argument.

Using the diagram used for shift count, MASK can be diagrammed as:



The MASK of FIELD2 equals 1F80H.

Example

```
MOV DX,BAN
AND DX,MASK FIELD2
```

FIELD2 is now isolated.

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. The right must be a constant if the left is a constant.

OPERATOR	MEANING
*	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. Example: MOV AX,100 MOD 17 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 shifted right. Example: MOV AX,1100000B SHR 5 The value moved into AX will be 11B (03).

OPERATOR	MEANING
SHL	Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be shifted left. Example: MOV AX,0110B SHL 5 The value moved into AX will be 011000000B (0C00H)
−(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 non-constant and in the same segment.

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.

OPERATOR	MEANING
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.

Logical Operators

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate 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.

OPERATOR	MEANING
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.

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:

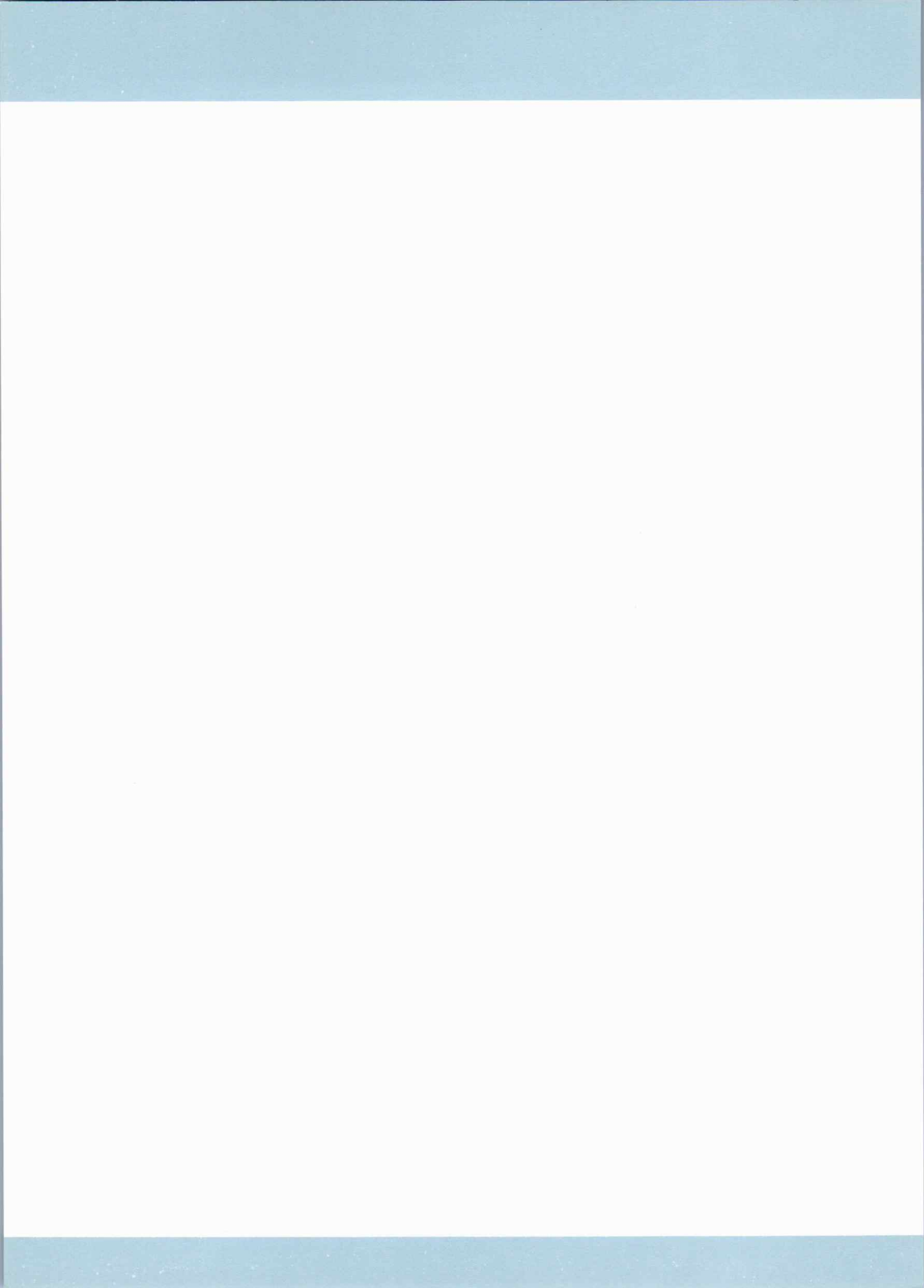
```
MOV AX,101B SHL 2*2 =   MOV AX,00101000B

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.

- LENGTH, SIZE, WIDTH, MASK
  Entries inside: parentheses ( )
  square brackets [ ]
  Structure variable operand: *variable.field*
- Segment override operator: colon (:)
- PTR, OFFSET, SEG, TYPE, THIS
- HIGH, LOW
- *, /, MOD, SHL, SHR
- +, − (both unary and binary)
- EQ, NE, LT, LE, GT, GE
- Logical NOT
- Logical AND
- Logical OR, XOR
- SHORT, .TYPE



4. INSTRUCTIONS AND DIRECTIVES

ABOUT THIS CHAPTER

This chapter describes the action part of the source file statement. It may be an instruction in the form of an 8086 mnemonic, or a directive to the Macro Assembler.

8086 instructions, listed in Appendix D, are not described in detail.

Assembler directives, listed in Appendix C, are described in this chapter alphabetically within groups: Memory, Conditional, Macro, and Listing.

This and the previous chapter cover all the information you need to create an assembler source file.

CONTENTS

ACTION: INSTRUCTIONS AND DIRECTIVES 4-1

INSTRUCTIONS 4-1

DIRECTIVES 4-2

MEMORY DIRECTIVES 4-3

ASSUME 4-3

COMMENT 4-5

DB,DW,DD,DQ,DT,(Define) 4-6

END 4-9

EQU 4-10

EQUAL SIGN 4-11

EVEN 4-12

EXTRN 4-13

GROUP 4-15

INCLUDE 4-17

LABEL 4-18

NAME 4-20

ORG 4-21

PROC 4-22

PUBLIC 4-24

.RADIX 4-25

RECORD 4-26

SEGMENT 4-29

STRUC 4-33

CONDITIONAL DIRECTIVES 4-35

MACRO DIRECTIVES 4-39

MACRO DEFINITION 4-40

CALLING A MACRO 4-41

ENDM (End Macro) 4-43

EXITM (Exit Macro) **4-44**

LOCAL **4-45**

PURGE **4-47**

REPEAT **4-48**

IRP (Indefinite Repeat) **4-50**

IRPC (Indefinite Repeat Character) **4-51**

LISTING DIRECTIVES **4-55**

PAGE **4-55**

TITLE **4-57**

SUBTITLE **4-58**

%OUT **4-59**

.LIST and .XLIST **4-60**

.SFCOND **4-61**

.LFCOND **4-61**

.TFCOND **4-61**

.XALL **4-62**

.LALL **4-62**

.SALL **4-63**

.CREF and .XCREF **4-63**

ACTION: INSTRUCTIONS AND DIRECTIVES

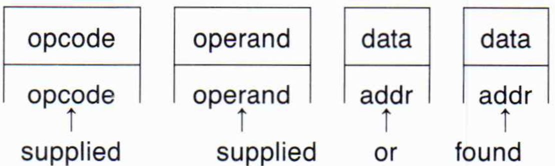
The action field contains either an 8086 instruction mnemonic or a Macro Assembler assembler directive.

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.

INSTRUCTIONS

Instructions tell the command processor to perform some action. 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 expressions in instruction statements. (opcode equates to the binary code for the action of an instruction)

Note that this manual does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this, you will need to consult other texts. The following text is recommended:

Intel's 8086 Instruction Mnemonics.

Appendix D 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.

DIRECTIVES

Directives give the assembler directions and information about 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 as shown below.

GROUP	MEANING
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 (for example, COMMENT).
Conditional Directives	Directives in this group are used to test conditions of assembly before proceeding 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 C contains a table of assembler directives, also grouped by function. Below is an alphabetical list of all the directives that Macro Assembler 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	

MEMORY DIRECTIVES

ASSUME

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register.

Syntax 1:

ASSUME *segment-register* **:** *segment-name*[, ...]

Syntax 2:

ASSUME NOTHING

Remarks

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 *segment-register* entries are:

CS, DS, ES, and SS.

The possible entries for *segment-name* are:

- The name of a segment declared with the SEGMENT directive
- The name of a group declared with the GROUP directive
- An expression: either SEG *variable-name* or SEG *label-name*
- 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 typed for *segment-name,* each reference to variables, symbols, labels, and so forth in a particular segment must be prefixed by a segment register. For example, type DS:ARG instead of simply ARG.

Example

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

COMMENT

COMMENT permits you to enter comments about your program without entering a semicolon (;) before each line.

COMMENT *delim text delim*

Remarks

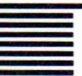
The first non-blank character encountered after COMMENT is the delimiter. The following *text* comprises a comment block which continues until the next occurrence of *delim*.

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
```



DB,DW,DD,DQ,DT,(Define)

The DEFINE directives are used to define variables or to initialize portions of memory.

[*varname*]	**DB**	*exp*[,	*exp*, ...]
[*varname*]	**DW**	*exp*[,	*exp*, ...]
[*varname*]	**DD**	*exp*[,	*exp*, ...]
[*varname*]	**DQ**	*exp*[,	*exp*, ...]
[*varname*]	**DT**	*exp*[,	*exp*, ...]

Remarks

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 Chapter 2.)

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:

- A constant expression
- 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).
- An address expression (for DW and DD only)

- An ASCII string (longer than one character, for DB only)
- *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.
- *exp* DUP(*exp*[,...]) This expression, like the previous one, 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.

Examples

Define Byte (DB):

```
NUM_BASE     DB   16
FILLER       DB   ?                    ;initialize with indetermi-
                                       ;nate value
ONE_CHAR     DB   'M'
MULT_CHAR    DB   'TOM JEREMY EDWARD BOB DEAN'
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
```

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
```

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
```

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(?)                ;uninit.data
FILLER      DQ   1 DUP(?,?)              ;initalized w_./
                                         ;indeterminate
                                         ;value
HEX_REAL    DQ   0FDCBA9A98765432105R
```

Define Tenbytes (DT):

```
ACCUMULATOR       DT    ?
STRING            DT    'CD'          ;no more than 2 char-
                                      ;acters
PACKED_DECIMAL    DT    1234567890
FLOATING_POINT    DT    3.1415926
```

END

The END statement specifies the end of the program.

END [*exp*]

Remarks

If *exp* is present, it is the start address of the program. If several modules are to be linked, only one 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

EQU assigns the value of *exp* to *name.*

---

name **EQU** *exp*

Remarks

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:

- A symbol. *name* becomes an alias for the symbol in *exp.* Shown as an Alias in the symbol table.
- An instruction name. Shown as an Opcode in the symbol table.
- A valid expression. Shown as a Number or L (label) in the symbol table.
- Any other entry, including text, index references, segment prefix and operands. Shown as Text in the symbol table.

Example

```
ARG   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

The equal sign (=) allows the user to set and to redefine symbols.

---

name = *exp*

Where

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).

Remarks

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

```
ARG    =      5        ;the same as ARG EQU 5
ARG    EQU    6;       ;error, ARG cannot be
                       ;redefined by EQU
ARG    =      7        ;ARG can be redefined
                       ;only by another =
ARG    =      ARG+3    ;redefinition may refer
                       ;to a previous definition
```

EVEN

The EVEN directive causes the program counter to go to an even boundary; that is, to an address that begins a word.

EVEN

Remarks

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 a NOP instruction.

EXTRN

The EXTRN directive identifies a procedure or function that resides in another loaded module.

EXTRN *name* **:** *type*[, ...]

Where

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, must be a valid type for *name:*

- BYTE, WORD, or DWORD
- NEAR or FAR for labels or procedures (defined under a PROC directive)
- ABS for pure numbers (implicit size is WORD, but includes BYTE)

Unlike the 8080 assembler, placement of the EXTRN directive is significant. If the directive is given with a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments, then use either

ASSUME *seg-reg* :SEG *name*

or an explicit segment prefix.

Remarks

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 less 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 return an error message.

Example

In Same Segment:	In Another Segment:
In Module 1:	In Module 1:

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

In Module 2:	In Module 2:

```
CSEG    SEGMENT                 EXTRN TAGF:FAR
        EXTRN TAGN:NEAR         CSEGB   SEGMENT
        .                               .
        .                               .
        JMP TAGN                        JMP TAGF
CSEG    ENDS                    CSEGB   ENDS
```

GROUP

The GROUP directive collects the segments named after GROUP *(segment name)* under one name.

---

name **GROUP** *segment-name*[, ...]

Remarks

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 in which the segments are loaded. The order in which the segments are loaded is determined 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.

segment-name may be one of the following:

- A segment name, assigned by a SEGMENT directive. The name may be a forward reference.
- An expression: either SEG *var* or SEG *label* Both of these entries resolve themselves to a segment name (see SEG operator).

Once you have defined a group name, you can use the name:

- As an immediate value:

```
MOV AX,DGROUP
MOV DS,AX
```

  DGROUP is the paragraph address of the base of DGROUP.

- In ASSUME statements:

```
ASSUME DS:DGROUP
```

The DS register can now be used to reach any symbol in any segment of the group.

- As an operand prefix (for segment override):

```
MOV BX,OFFSET DGROUP:ARG
DW   DGROUP:ARG
DD   DGROUP:ARG
```

DGROUP: forces the offset to be relative to DGROUP, instead of to the segment in which ARG is defined.

Example

Using GROUP to combine segments:

In Module A:

```
CGROUP  GROUP    XXX,YYY
XXX     SEGMENT
        ASSUME   CS:CGROUP
          .
          .
          .
XXX     ENDS
YYY     SEGMENT
          .
          .
          .
YYY     ENDS
        END
```

In Module B:

```
CGROUP  GROUP    ZZZ
ZZZ     SEGMENT
        ASSUME   CS:CGROUP
                 .
                 .
                 .
ZZZ     ENDS
        END
```

INCLUDE

The INCLUDE directive inserts source code from an alternate assembly language source file into the current source file during assembly.

INCLUDE *filename*

Remarks

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 displayed, and the assembly aborts.

On a Macro Assembler listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See "Formats of Listings and Symbol Tables," in Chapter 5 for a description of listing file formats.

Example

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

LABEL

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.

---

name **LABEL** *type*

Where

type varies depending on the use of *name; name* may be used for code or for data:

- For code (for example, as a JMP or CALL operand):

  type may be either NEAR or FAR. *name* cannot be used in data manipulation instructions without using a type override.

  If you wish, 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.

- 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 Code:

```
SUBRTF LABEL   FAR
SUBRT: (first instruction)     ;colon = NEAR label
```

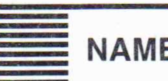
Example

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 ways also permits you to avoid using the PTR operator. The double defining method is especially effective if you access the data different ways. It is easier to give the array a second name than to remember to use PTR.



NAME

Declares the name of a module

NAME *module-name*

Where

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

Remarks

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

Every module has a name. Macro Assembler derives the *module-name* from:

- A valid NAME directive statement
- If the module does not contain a NAME statement, Macro Assembler uses the first six characters of a TITLE directive statement. The first six characters must be legal as a name.

Example

```
NAME CURSOR
```

The location counter is set to the value of *exp*, and the assembler assigns generated code starting with that value.

ORG *exp*

Remarks

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 later in this chapter for an explanation of conditional assembly.

PROC

The PROC directive serves as a structuring device to make your programs more understandable.

---

procedure-name **PROC** [**NEAR**|**FAR**]

.
.
.
RET

procedure-name **ENDP**

Remarks

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 usually contains a RET statement.

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

Places a PUBLIC directive statement in any module that contains symbols you want to use in other modules without defining the symbol again. PUBLIC means 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.

PUBLIC *symbol* [, ...]

Where

symbol may be a number, a variable, a label (including PROC labels). It may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes.

Example 1

```
         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 2

Illegal PUBLIC:

```
           PUBLIC PIE_BALD,HIGH_VALUE
PIE_BALD   EQU    3.1416
HIGH_VALUE EQU 999999999
```

.RADIX

The RADIX directive permits you to change the input radix to any base in the range 2 to 16.

.RADIX *exp*

Where

exp is always in decimal radix, regardless of the current input radix.

Remarks

The default input base (or radix) for all constants is decimal.

Example 1

```
        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 2

```
                .RADIX 16
NUM_HAND        DT     773   ;773 = decimal
HOT_HAND        DQ     773Q  ;773 = octal here only
COOL_HAND       DD     773H  ;now 773 = hexadecimal
```

RECORD

A record is a bit pattern you define to format bytes and words for bit-packing.

---

recordname **RECORD** *fieldname* : *width*[= *exp*] ,[...]

Where

fieldname is the name of the field. *fieldname* becomes a value that can be used in expressions. When you use *fieldname* in an expression, its value in 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 specifies the number of bits in the field defined by *fieldname*. It is a constant in the range 1 to 16. 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.

exp contains the initial (or default) value for the field. If the field is at least 7 bits wide, you can use an ASCII character as the *exp*.

Remarks

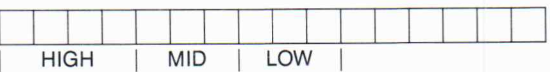
Forward references are not allowed in a RECORD statement.

The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right 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.

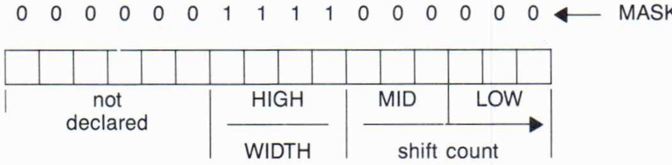
Example 1

ARG RECORD HIGH:4,MID:3,LOW:3

Initially, the bit map would be:



In this example the total bits are greater than 8, so the field requires two bytes; however, the total bits are less than 16, so the data is right shifted and all unused bits are at the left.



To initialize records, use the same method used for DB. The syntaxes are:

Syntax 1

[*name*] *recordname* <[*exp*][,...]>

Syntax 2

[*name*] *recordname* [*exp*] **DUP (**<[*exp*][,...]>**)**

Where

name is optional. When given *name* is a label for the first byte or word of the record storage area.

recordname in the name used as a label for the RECORDS directive.

exp for both forms, contains the values you want placed into the fields of the record. If *exp* is left blank, either the default values applies (the value given in the original record definition), or the value is indeterminate (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) those fields.

For example:

ARG ,,7

From the previous example, the 7 would be placed into the LOW field of the record ARG. The fields HIGH and MID would be left as declared (in this case, uninitialized).

A record may be used in an expression (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 3

```
ARG  RECORD  HIGH:5,MID:3,LOW:3
         .
         .
BAX  ARG     < >                 ;leave undetermined here
JANE ARG     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

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.

---

segname **SEGMENT** [*align*][*combine*][*'class'*]

.
.
.

segname **ENDS**

Where

segname must be a unique, legal name. The *segname* must not be a reserved word.

align may be PARA (paragraph - default), BYTE, WORD, or PAGE.

combine may be PUBLIC, COMMON, AT *exp,* STACK, MEMORY, or no entry (which defaults to not combinable, called Private in the LINK section of the manual).

class name is used to group segments at link time.

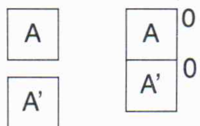
All three operands are passed to MS-LINK.

The alignment type 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)

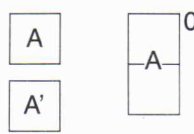
The combine type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

- None (not combinable or Private)



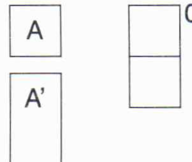
Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

- 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

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.

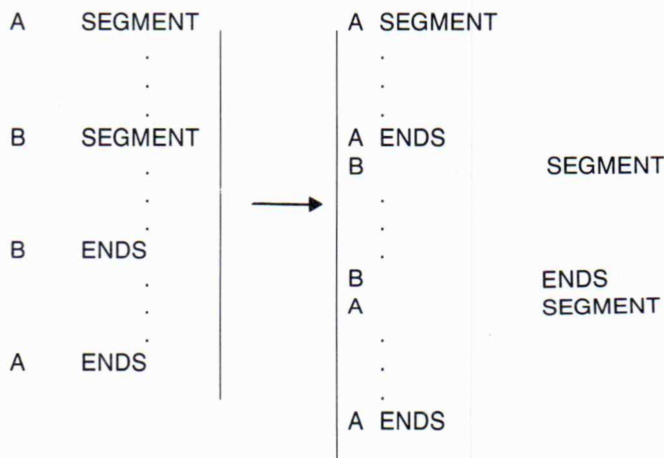
  Note

  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.

  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.

Example 1



The following arrangement is not allowed:

```
A       SEGMENT
        .
        .
        .
B       SEGMENT
        .
        .
        .
A       ENDS          ;This is illegal!
        .
        .
B       ENDS
```

Example 2

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

The STRUC directive is very much like RECORD, except STRUC has a multiple byte capability.

---

structure-name **STRUC**

.
.
.

structure-name **ENDS**

Remarks

The allocation and initialization of a STRUC block are 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 field values are 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:

```
ARG  DB  1,2         ;is not overridable
BAN  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

Where

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:

```
ARG   STRUCTURE
        .
        .
        .
ARG   ENDS
GOO   ARG          <,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   'NARGHED'         ;overridable
S      ENDS
```

The Define directives in this example define the fields of the structure, and the order corresponds to the order values 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
```

CONDITIONAL DIRECTIVES

Conditional directives allow users to design blocks of code which test for specific conditions.

All conditionals follow the format:

IF*xxxx* [*argument*]

.
.
.

[**ELSE**]

.
.
.

ENDIF

Each IF*xxxx* 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 nonzero 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.

The following is a list of Macro Assembler conditional directives:

- **IF** *exp*

  If *exp* evaluates to nonzero, the statements within the conditional block are assembled.

- **IFE** *exp*

  If *exp* evaluates to 0, the statements in the conditional block are assembled.

- **IF1** Pass 1 Conditional

  If the assembler is in pass 1, the statements in the conditional block are assembled. IF1 takes no expression.

- **IF2** Pass 2 Conditional

  If the assembler is in pass 2, the statements in the conditional block are assembled. IF2 takes no expression.

- **IFDEF** *symbol*

  If the *symbol* is defined or has been declared External, the statements in the conditional block are assembled.

- **IFNDEF** *symbol*

  If the *symbol* is not defined or not declared External, the statements in the conditional block are assembled.

- **IFB** <*arg*>

  The angle brackets around *arg* are required.

  If the <*arg*> is blank (none given) or null (two angle brackets with nothing in between, < >), the statements in the conditional block are assembled.

  IFB (and IFNB) are normally used inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy will be replaced by a parameter passed by the macro call. If the macro call does not specify a parameter to replace the dummy following IFB, the expression is blank, and the block will be assembled. (IFNB is the opposite case.) Refer to "Macro Directives," later in this chapter 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 "Macro Directives", later in this chapter 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 dummies will be replaced by parameters passed by the macro call. If the macro call specifies two identical parameters to replace the dummies, the block will be assembled. (IFDIF is the opposite case.) Refer to "Macro Directives", later in this chapter 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 dummies will be replaced by parameters passed by the macro call. If the macro call specifies two different parameters to replace the dummies, 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. ELSE 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.

MACRO DIRECTIVES

The macro directives allow you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition directive or one of the repetition directives, and end with the ENDM directive. All of the macro directives may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro directives of the Macro Assembler include:

macro definition:
  MACRO

termination:
  ENDM
  EXITM

unique symbols within macro blocks:
  LOCAL

undefine a macro:
  PURGE

repetitions:
  REPT (repeat)
  IRP (indefinite repeat)
  IRPC (indefinite repeat character)

The macro directives also include some special macro operators:

& (ampersand)
;; (double semicolon)
! (exclamation mark)
% (percent sign)

MACRO DEFINITION

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 **MACRO** [*dummy*, ...]

.
.
.

ENDM

Where

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.

dummy is formed as any other name is formed. A *dummy* is a place holder that is replaced by a parameter in a one-for-one text substitution when the macro block is used. You should include all *dummys* used inside the macro block on this line. The number of *dummys* is limited only by the length of a line. If you specify more than one *dummy*, they must be separated by commas. Macro Assembler interprets a series of *dummys* the same as any list of symbol names.

Remarks

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 *dummy*s.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler *expands* the macro call statement by bringing in and assembling the appropriate macro block.

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable, or symbol. When Macro Assembler evaluates a statement, it first looks at the macro table it builds during pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: Macro Assembler evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, Macro Assembler will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

CALLING A MACRO

To use a macro, enter a macro call statement with the following format.

---

name [<*parameter*> , ...]

Where

name is the name of the macro block.

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:

```
ARG 1,2,3,4,5
```

passes five parameters to the macro, but

```
ARG <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   NARG,DON,ARG
```

the assembler generates the statements:

```
      MOV     AX,NARG
      ADD     AX,DON
      MOV     ARG,AX
```

On your program listing, these statements will be preceded by a plus sign (+) to indicate that they came from a macro block.

ENDM (End Macro)

ENDM tells the assembler that the MACRO or Repeat block is ended.

ENDM

Remarks

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.

EXITM (Exit Macro)

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.

EXITM

Remarks

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

```
ARG   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

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.

LOCAL *dummy*[, *dummy*...]

Remarks

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
                ARG      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
                         ARG    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
                         ARG        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 Assembler substitutes LABEL names in the form ??nnnn for the instances of the dummy symbols.

PURGE

PURGE deletes the definition of the macro(s) listed after it.

PURGE *macro-name* [, ...]

Remarks

PURGE provides three benefits:

- It frees text space of the macro body.
- It returns any instruction mnemonics or directives that were redefined by macros to their original function.
- 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:

- 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.

- 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.

REPEAT

Repeats block of statements between REPT and ENDM *exp* times.

REPT *exp*

.

.

.

ENDM

Where

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
```

IRP (Indefinite Repeat)

IRP *dummy*, *<parameters>*

```
.
.
.
```

ENDM

Remarks

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 to 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:

```
ARG   MACRO   X
      IRP     Y,<X>
      DB      Y
      ENDM
      ENDM
```

When the macro call statement

```
ARG <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 will be removed, and all items are passed as a single parameter.

IRPC (Indefinite Repeat Character)

IRPC *dummy*, *string*

.
.
.

ENDM

Remarks

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 to 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 ampersand 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 an ampersand. To form a symbol from text and a dummy, put an ampersand between them.

For example:

```
ERRGEN          MACRO   X
ERROR&X:                PUSH   BX
                MOV     BX,'&X'
                JMP     ERROR
                ENDM
```

The call ERRGEN A will then generate:

```
ERRORA:         PUSH    B
                MOV     BX,'A'
                JMP     ERROR
```

In Macro Assembler, the ampersand will not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, extra ampersands may be needed. You need to supply as many ampersands as there are levels of nesting.

For example:

Correct form	Incorrect form
ARG MACRO X IRP Z,<1,2,3> X&&Z DB Z ENDM ENDM	ARG MACRO X IRP Z,<1,2,3> X&Z DB Z ENDM ENDM

When called, for example, by ARG BAN, the expansion would be (correctly in the left column, incorrectly in the right):

- MACRO build, find *dummys* and change to d1

```
       IRP    Z,<1,2,3>          IRP    Z,<1,2,3>
d1&Z   DB     Z           d1Z    DB     Z
       ENDM                      ENDM
```

- MACRO expansion, substitute parameter text for d1

```
       IRP    Z,<1,2,3>          IRP    Z,<1,2,3>
BAN&Z  DB     Z           BANZ   DB     Z
       ENDM                      ENDM
```

- IRP build, find *dummys* and change to d1

```
BAN&d1 DB     d1          BANZ   DB     d1
```

- IRP expansion, substitute parameter text for d1

```
BAN1   DB     1           BANZ   DB     1
BAN2   DB     2           BANZ   DB     2  <-
BAN3   DB     3           BANZ   DB     3   |
                                 ;here it's an error,
                                 ;multi-defined symbol
```

<text> Angle brackets cause Macro Assembler to treat the text between the angle brackets as a single literal. Placing parameters to a macro call inside angle brackets; or placing the list of parameters following the IRP directive inside angle brackets causes two results:

- All text within the angle brackets is seen as a single parameter, even if commas are used.
- 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 "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 *
```

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 directives turn on and off the listing of all or part of the assembled file.

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.

Syntax 1:

PAGE [*length*][, *width*]

Syntax 2:

PAGE [+]

Remarks

The PAGE directive with either the length or width argument 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 one. 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 specifies a title to be listed on the first line of each page.

TITLE *text*

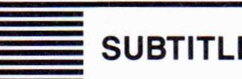
Where

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 specifies a subtitle to be listed in each page heading on the line after the title.

SUBTTL *text*

Where

text is truncated after 60 characters.

Remarks

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

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 *text*

Remarks

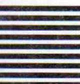
%OUT will output on both passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See "Conditional Directives," earlier in this chapter for descriptions of the IF1 and IF2 directives.

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 and .XLIST

The .LIST directive lists all lines with their code (the default condition). .XLIST suppresses all listing.

Syntax 1:

.LIST

Syntax 2:

.XLIST

Remarks

If you specify a listing file following the Listing: prompt, a listing file with all the source statements included will be printed.

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.

The .LIST directive overrides all other listing directives. 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

Remarks

The .SFCOND directive suppresses portions of the listing that contain conditional false expressions.

.LFCOND

.LFCOND

Remarks

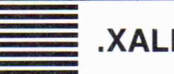
The .LFCOND directive assures the listing of conditional expressions that evaluate false. This is the default condition.

.TFCOND

.TFCOND

Remarks

The .TFCOND directive 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 the assembler is running. When /X is used, .TFCOND will cause false conditionals to list. When /X is not used, .TFCOND will suppress false conditionals.



.XALL

.XALL

Remarks

.XALL is the default.

The .XALL directive lists source code and object code produced by a macro, but source lines which do not generate code are not listed.

.LALL

.LALL

Remarks

The .LALL directive 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

Remarks

The .SALL directive suppresses listing of all text and object code produced by macros.

.CREF and .XCREF

Syntax 1:

.CREF

Syntax 2:

.XCREF [*variable-list*]

Characteristics

The .CREF directive is the default condition. .CREF remains in effect until Macro Assembler encounters .XCREF.

The .XCREF directive without arguments turns off the .CREF (default) directive. .XCREF remains in effect until Macro Assembler encounters .CREF. Use .XCREF to suppress the creation of cross-references in selected portions of the file. Use .CREF to restart the creation of a cross-reference file after using the .XCREF directive.

If you include one or more variables following .XCREF, these variables will not be placed in the listing or cross-reference file. All other cross-referencing, however, is not affected by an .XCREF directive with arguments. Separate the variables with commas.

Neither .CREF nor .XCREF without arguments takes effect 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,ARG,NAR,BAN,ZAN
            ;these variables will not be in the
            ;listing or cross-reference file
```

5. ASSEMBLING A MACRO ASSEMBLER SOURCE FILE

ABOUT THIS CHAPTER

This chapter tells you how to assemble a source file once you have created it.

To start Macro Assembler, commands may be entered on the command line, or in response to prompts for the names of source, object, listing, and cross-reference files.

There are two optional command characters to simplify command entry. Three optional switches, typed at the end of a command line or prompt response, control assembler functions.

Samples from program and symbol portions of listings are provided, with explanatory notes.

CONTENTS

ASSEMBLING A MACRO ASSEMBLER SOURCE FILE 5-1

HOW TO START MACRO ASSEMBLER 5-1

METHOD 1: PROMPTS 5-1

METHOD 2: COMMAND LINE 5-2

MACRO ASSEMBLER COMMAND CHARACTERS 5-4

MACRO ASSEMBLER COMMAND PROMPTS 5-5

MACRO ASSEMBLER COMMAND SWITCHES 5-7

FORMATS OF LISTINGS AND SYMBOL TABLES 5-10

PROGRAM LISTING 5-11

DIFFERENCES BETWEEN PASS 1 AND PASS 2 LISTINGS 5-16

ASSEMBLING A MACRO ASSEMBLER SOURCE FILE

Assembling a program with Macro Assembler requires two types of commands: a command to start Macro Assembler, and answers to command prompts. In addition, four switches control alternate Macro Assembler features. Usually, you will type all the commands to Macro Assembler on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in response (batch) file. Two command characters are provided to assist you while entering assembler commands. These command characters are described later in this chapter.

HOW TO START MACRO ASSEMBLER

Macro Assembler may be started in two ways. By the first method, you type the commands in response to individual prompts. By the second method, you type all commands on the line used to start Macro Assembler.

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

Tab. 5-1 Summary of Methods to Start Macro Assembler

METHOD 1: PROMPTS

Type:

MASM

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

At the end of each line, you may specify one or more switches, each of which must be preceded by a forward slash (/).

The command prompts are summarized here and described in more detail in the section on "Macro Assembler Command Prompts".

PROMPT	RESPONSES
Source filename [.ASM]:	List .ASM file to be assembled. (There is no default: a filename response is required.)
Object filename [source.OBJ]:	List filename for relocatable object code. (The default is source-filename. OBJ)
Source listing [NUL.LST]:	List filename for listing. (The default is no listing file.)
Cross reference [NUL.CRF]:	List filename for cross-reference file (used with MS-CREF to create a cross-reference listing). (The default is no cross-reference file.)

METHOD 2: COMMAND LINE

Type:

MASM *source*, *object*, *listing*, *cross-ref* [*/switch...*]

Macro Assembler will be loaded into memory. Then Macro Assembler immediately begins assembly. The entries following MASM are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

Where

source is the file to receive the relocatable output

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 comma or after the *cross-ref*, as shown).

To select the default for a field, simply enter a second comma without space in between (see the example below).

Example

```
MASM FUN,,FUN/D/X,FUN
```

This example causes Macro Assembler to be loaded, then causes the source file FUN.ASM to be assembled. Macro Assembler then outputs the relocatable object 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 switches are given but no filename, the switches are ignored.

MACRO ASSEMBLER COMMAND CHARACTERS

Macro Assembler provides two command characters.

Semicolon — 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 eliminates the need to enter a series of 3 commas or 3 carriage returns.

Once the semicolon has been entered, you can no longer respond to any of the prompts for that assembly. Therefore, do not use the semicolon if you wish only to skip SOME prompts. In the following example, the first parameter is entered, followed by 3 commas:

```
Source filename [.ASM]: FUN,,,
```

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

To achieve this same result, you could have typed:

```
Source filename [.ASM]: FUN ;
```

This response produces the same files as the previous example.

CTRL C — Use **CTRL 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 **CTRL C** to exit Macro Assembler. You can then restart Macro Assembler. If the error has been typed and not entered, you may delete the erroneous characters, but for that line only.

MACRO ASSEMBLER COMMAND PROMPTS

Macro Assembler is commanded by entering responses to tour text prompts. When you have typed a response to the current prompt, the next appears. When the last prompt has been answered, Macro Assembler begins assembly automatically without further command. When assembly is finished, Macro Assembler exits to the operating system. When the operating system prompt is displayed, Macro Assembler has finished successfully. If the assembly is unsuccessful, Macro Assembler displays the appropriate error message.

Macro Assembler prompts you for the names of source, object, listing, and cross-reference files.

All command prompts accept a file specification as a response. You may type:

- A filename only
- A device designation only
- A filename and an extension
- A device designation and filename, or
- A device designation, filename, and extension.

Do not type only a filename extension.

The following is a discussion of the command prompts that are displayed when you start Macro Assembler with Method 1:

Source filename [.ASM]:

> Type the filename of your source program. Macro Assembler assumes by default that the filename extension is .ASM, as shown in square brackets in the prompt text. If your source program has any other filename extension, you must specify it along with the filename. Otherwise, the extension may be omitted.

Object filename [source.OBJ]:

Type 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 type your choice in response to this prompt. If you want to change only the filename but keep the .OBJ extension, type the filename only. To change the extension only, you must type both the filename and the extension.

Source listing [NUL.LST]:

Type the name of the file you want to receive the source listing. If you press the carriage return key, Macro Assembler does not produce this listing file. If you type a filename only, the listing is created and placed in a file with the name you type plus the filename extension .LST. You may also type 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]:

Type the name of the file you want to receive the cross-reference file. If you press only the **CR** (carriage return) key, Macro Assembler does not produce this cross-reference file. If you type a filename only, the cross-reference file is created and placed in a file with the name you type plus the filename extension .CRF. You may also type your own extension.

The cross-reference file is used as the source file for the CREF Cross-Reference Utility (MS-CREF). 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.

MACRO ASSEMBLER COMMAND SWITCHES

The three Macro Assembler switches control assembler functions. Switches must be typed at the end of a prompt response, regardless of which method is used to start Macro Assembler. Switches may be grouped at the end of any one of the responses, or may be scattered at the ends of several. If more than one switch is typed at the end of one response, each switch must be preceded by a forward slash (/). Do not specify only a switch as a response to a command prompt.

Switch	Function
/D	Produces a source listing on both assembler passes. The listings will, when compared, show where in the program phase errors occur and will, possibly, give you a clue to why the errors occur. The /D switch does not take effect unless you command Macro Assembler to create a source listing (type a filename in response to the Source listing: command prompt).
/O	Outputs 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.

Switch	Function
/X	Suppresses the listing of false conditionals. If your program contains conditional blocks, the listing file will show the source statements, but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not generate code, use the /X switch to suppress the blocks that evaluate false from your listing. The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives. If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, the second restores listing of false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When Macro Assembler encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing. Of course, the /X switch has no effect if no listing is created. See additional discussion under the .TFCOND directive in "Listing Directives" 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

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.

Tab. 5-2 Summary of Command Switches

FORMATS OF LISTINGS AND SYMBOL TABLES

The source listing produced by Macro Assembler (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.

PROGRAM LISTING

The program portion of the listing is essentially your source program file 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, or 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.

Part of a listing file follows this discussion, 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 will easily fit on the same line as the rest of the statement. Explanatory notes are spliced into the listing at points of special interest.

The following table summarizes the listing symbols used.

SYMBOL	DEFINITION
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

SYMBOL	DEFINITION
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 Assembler  1-Dec-81   PAGE 1-3

EXTX  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
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 data segment value
0003  8E D8                MOV    DS,AX ;Set DS seg

0005  8C 06 0022 R         MOV    CESXQQ,ES
  |    Generated    Name       Action  Expression   Comment
Offset
                                                  .
                                                  .
                                                  .
000C  26: 8B 1E 0002       MOV     BX,ES:2  ;Highest
                                              ;paragraph
       |__________________ Segment override__|
```

```
Microsoft 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      MAIN_STARTUP  ENDS

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

```
Microsoft 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
```

DIFERENCES BETWEEN PASS 1 AND PASS 2 LISTING

If you specify the /D switch when you run Macro Assembler to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that 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 instruction's code now contains 03 instead of 00; this is a jump of 3 bytes.

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

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, and is useful when 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.

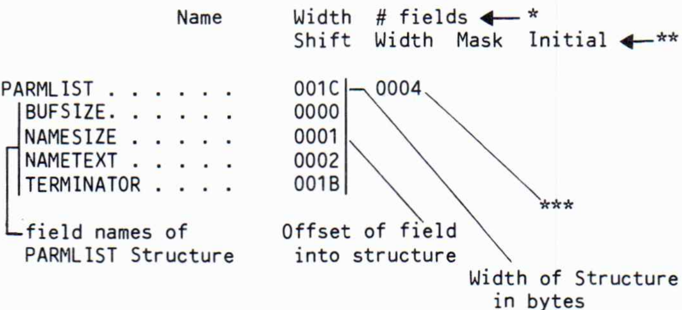
```
Macro Assembler MACRO
Assembler     date     PAGE   Symbols-1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXPM1M.ASM)

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 . . . . . . . .         0000
  NAMETEXT . . . . . . . .         0002
  TERMINATOR . . . . . . .         001B
```

```
Segment 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 in memory
BIOSCALL  . . . . . . . .           0002
DISPLAY . . . . . . . . .           0005
DOSCALL . . . . . . . . .           0002
KEYBOARD  . . . . . . . .           0003
LOCATE  . . . . . . . . .           0003
SCROLL  . . . . . . . . .           0004
  ^
  |
names of macros
```

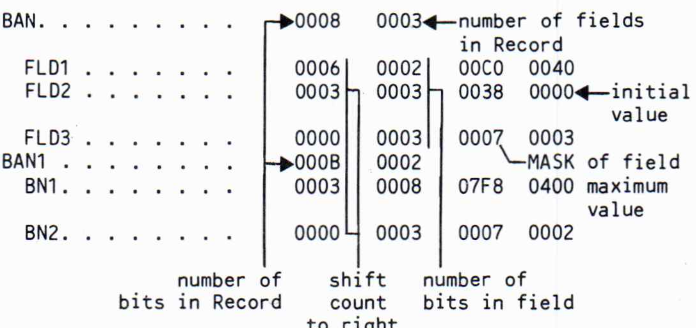
This section of the symbol table tells you the names of your macros and how big they are in 32-byte block units. In this listing, the macro DISPLAY is 5 blocks long or (5 X 32 bytes =) 160 bytes long.

Structures and records

Example for Structures



Example for Records



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

** This line for fields of Records (indented).

*** Number of fields in Structure.

This section lists your Structures and/or Records and their fields. The upper line of column headings applies to Structure names, Record names, and field names of Structures. The lower line of column headings applies to field names of Records.

For Structures:

- Width (upper line) shows the number of bytes your Structure occupies in memory.
- # fields shows how many fields comprise your Structure.

For Records:

- Width (upper line) shows the number of bits the Record occupies.
- # fields shows how many fields comprise your Record.

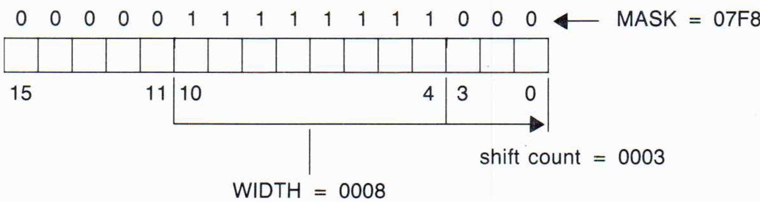
For Fields of Structures:

- Shift shows the number of bytes the fields are offset into the Structure.
- The other columns are not used for fields of Structures.

For Fields of Records:

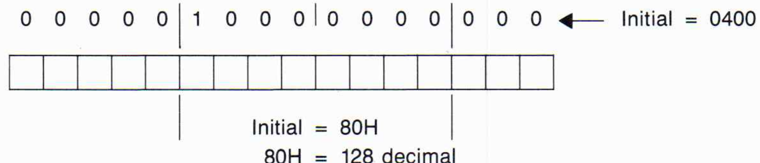
- Shift is the shift count to the right.
- Width (lower line) shows the number of bits this field occupies.
- Mask shows the maximum value of the record, expressed in hexadecimal, if one field is masked and ANDed (the field is set to all 1's and all other fields are set to all 0's).

Using field BN1 of the Record BAN1 above to illustrate:



- Initial shows the value specified as the initial value for the field, if any.
  When naming the field, you specified:

  fieldname: # =value
- Fieldname is the name of the field
- # is the width of the field in bits
- Value is the initial value you want this field to hold. The symbol table shows this value as if it is placed in the field and all other fields are masked (equal 0). Using the example and diagram from above:




```
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:

- Size is the number of bytes the Segment occupies.
- Align is the type of boundary where the segment begins:

  PAGE=page - address is xxx00H (low byte=0); begins on a 256-byte boundary

  PARA=paragraph - address is xxxx0H (low nibble=0); default

  WORD=word - address is xxxxeH (e=even number; low bit of low byte=0)
  bit map - |x|x|x|x|x|x|x|0|

  BYTE=byte - address is xxxxxH (anywhere)

- Combine describes how the LINK Linker Utility will combine the various segments.
- Class is the class name under which MS-LINK will combine segments in memory.

```
Symbols:

             Name      Type      Value    Attr

ARG. . . . . .         Number    0005          |
ARG1 . . . . .         Text      1.234         |
ARG2 . . . . .         Number    0008          |  all formed by
ARG3 . . . . .         Alias     ARG           |  EQU or =
ARG4 . . . . .         Text      5[BP][DI]     |  directive
ARG5 . . . . .         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 Assembler knows this length as one of the
      type lengths (BYTE, WORD, DWORD, QWORD,
      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
  Text
  Opcode } all defined by EQU or=directive

  These entries may be combined to form the various types shown in the example.

  For all procedures, the legth 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:

  BAN LABEL ARG

  where ARG is a STRUC that is 31 bytes long.

  BAN will be shown in the symbol table with the L 0031 enty. 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 least entry applies to PROC types only. This is a length=entry, which is the length of the procedure.

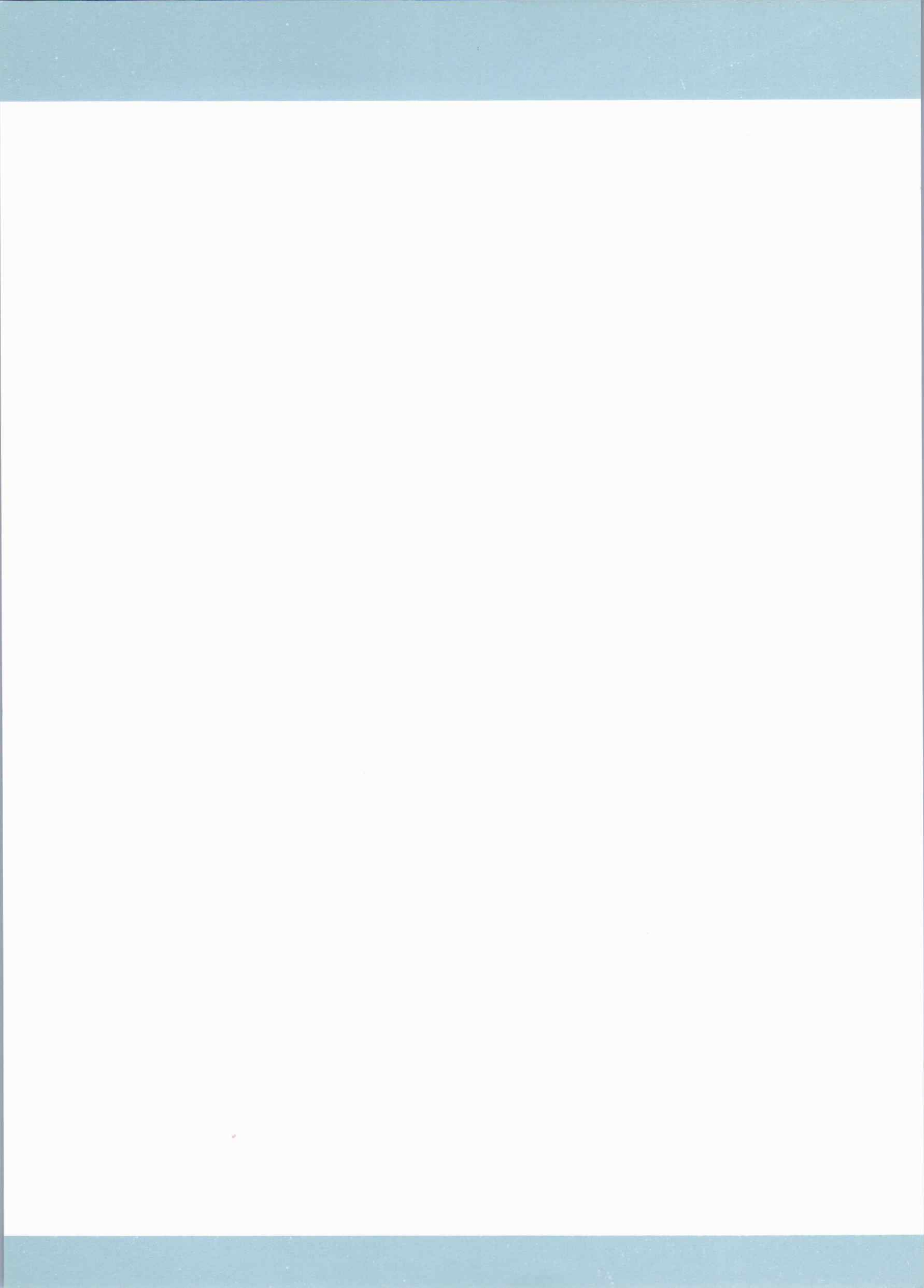
  If Type is Number, Opcode, Alias, or Text, the Symbols section of the listing will be structured differently. Whenever you see one of these four entries under Type, the symbol was created by an EQU directive or an=directive. All information that follows one of these entries is considered its "value", even if the "value" is simple text.

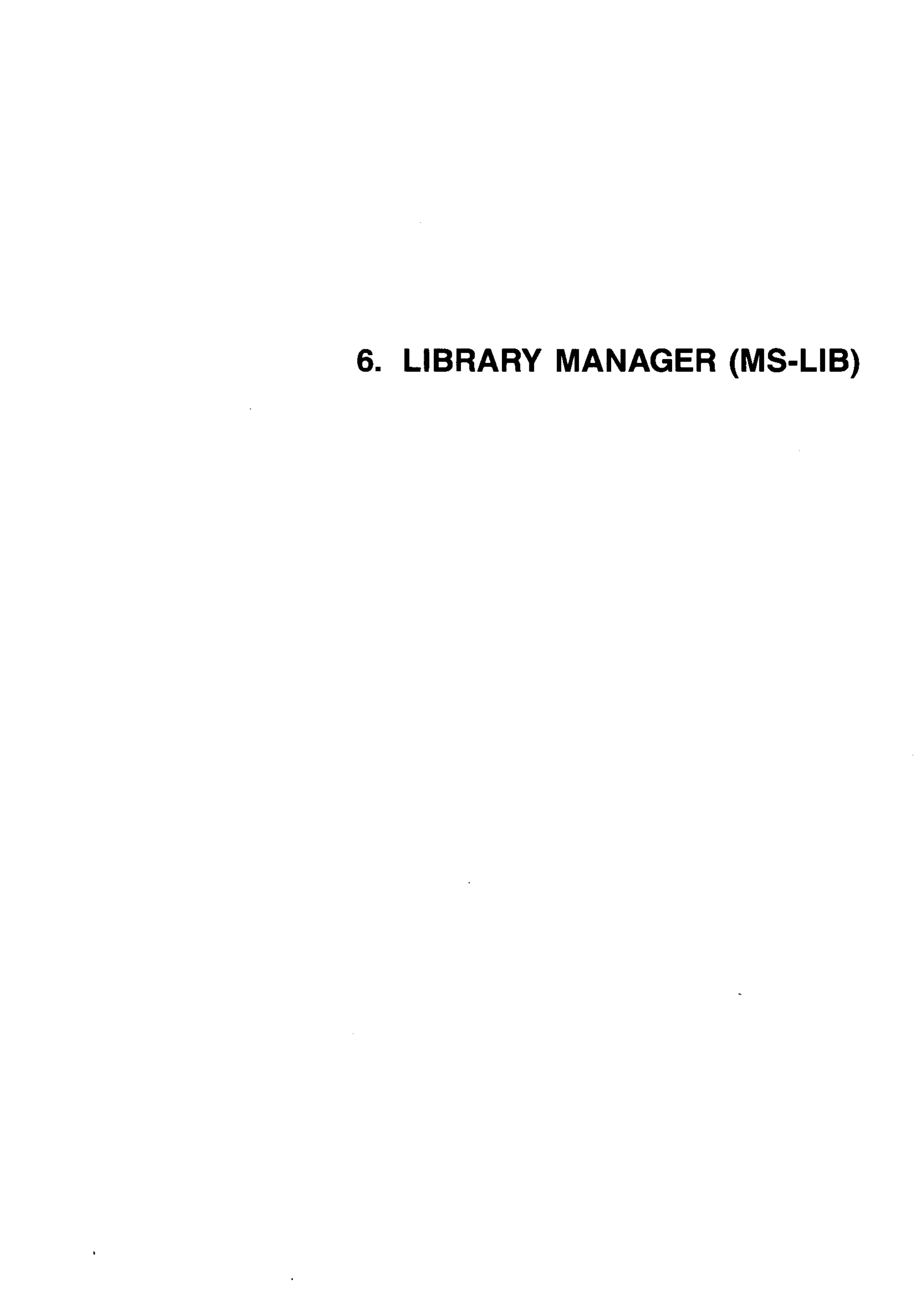
Each of the types shows a value as follows:

- Number shows a constant numeric value.
- Opcode shows a blank. The symbol is an alias for an instruction mnemonic.
- Sample directive statement: ARG EQU ADD
- Alias shows a symbol name which the named symbol equals.
- Sample directive statement: ARG 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'
  BAN EQU DS:8[BX]
  ZOO EQU 1.234
  ```





6. LIBRARY MANAGER (MS-LIB)

ABOUT THIS CHAPTER

This chapter discusses the MS-LIB (library manager) utility. It describes its function and its use.

With MS-LIB the programmer can:

- Create and modify library files that are used with the MS-LINK utility.
- Add object files to a library.
- Delete modules from a library.
- Extract modules from a library and place the extracted modules into separate object files.

CONTENTS

INTRODUCTION 6-1

OVERVIEW OF MS-LIB OPERATION 6-1

RUNNING MS-LIB 6-5

HOW TO START MS-LIB 6-5

COMMAND PROMPTS 6-9

COMMAND CHARACTERS 6-11

ERROR MESSAGES 6-14

INTRODUCTION

MS-LIB is a library manager which can create either general libraries, which can be used by a variety of programs, or special libraries, created for the use of specific programs. By using MS-LIB the user can achieve faster linking and more efficient execution, whether for a language compiler or for just one program.

The user can modify individual modules within a library by extracting the modules, making changes, then adding the modules to the library again. The user 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 also used in MS-LINK, MS-Pascal and MS-FORTRAN. Command syntax is straightforward, and MS-LIB prompts you for commands that you have not supplied.

OVERVIEW OF MS-LIB OPERATION

MS-LIB performs five library manager functions:

- Deletes modules
- Extracts a module and places it in a separate object file
- Appends an object file as a module of a library
- Replaces a module in the library file with a new module
- Creates a library file

During each library session, MS-LIB deletes or extracts modules, then appends new ones to the library file. 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 that module into memory but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places that module at the end of the last module written. This procedure effectively "closes up" the disk space to keep the library file from growing too large.

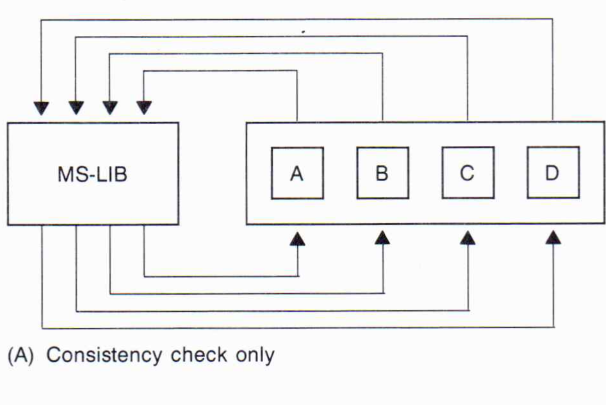
When MS-LIB has read the library file, it appends any new modules to the end of the file. Finally, MS-LIB creates the index, which MS-LINK uses to find modules and symbols in the library file. MS-LIB will output a cross-reference listing of the PUBLIC symbols in the library, if you request such a listing.

Example

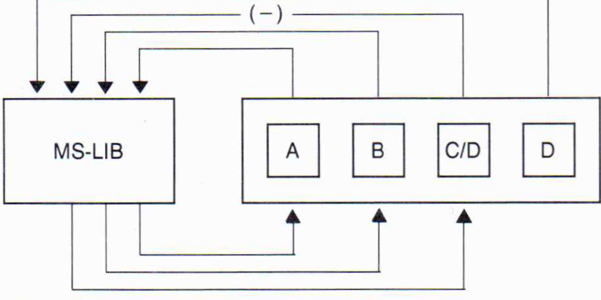
LIBx PASCAL+HEAP-HEAP;

This command first deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. 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. This order of execution prevents confusion in MS-LIB when a new version of a module replaces a version in the library file.

The following figure (Figure 6-1) illustrates the MS-LIB operation.

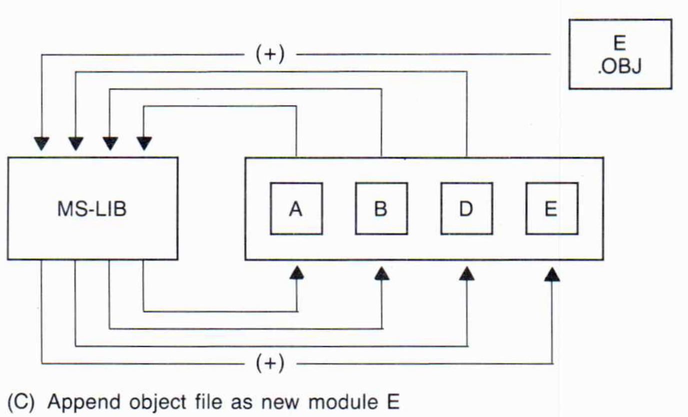


(A) Consistency check only

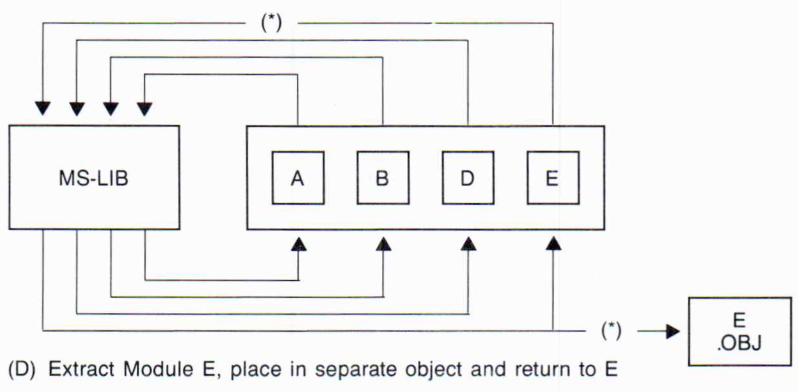


(B) Delete Module C; move Module D to C's space

Fig. 6-1 MS-LIB Operation

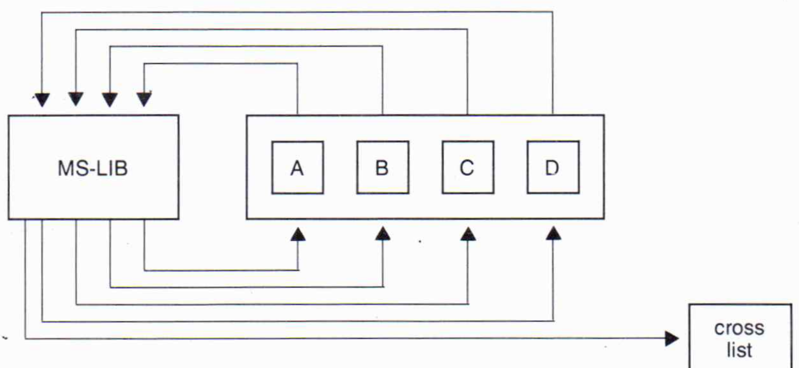


(C) Append object file as new module E



(D) Extract Module E, place in separate object and return to E

Fig. 6-1 MS-LIB Operation (continued)



(E) Make consistency check, output a cross reference listing of PUBLIC symbols

Fig. 6-1 MS-LIB Operation (continued)

RUNNING MS-LIB

Running MS-LIB requires two types of commands: a command to start MS-LIB and answers to command prompts. Usually you will type all the commands to MS-LIB on a command line or in response to MS-LIB prompts. As an option, answers to the command prompts may be contained in a response file. Command characters can be used to assist you while giving commands to MS-LIB.

HOW TO START MS-LIB

There are three ways to start MS-LIB. With the first method, you type the commands as answers to individual prompts. By the second method, you type all commands on the line used to start MS-LIB. As a third option, you can create a response file that contains all the necessary commands.

Method 1	LIB
Method 2	LIB *library operations, listing*
Method 3	LIB@ *filespec*

Tab. 6-2 Summary of Methods to Start MS-LIB

Method 1: Prompts

To start MS-LIB with method 1, type:

LIB

MS-LIB will be loaded into memory. MS-LIB will then display three text prompts that appear one at a time. You answer the prompts, commanding MS-LIB to perform specific tasks.

The command prompts are summarized below and described more fully later in this chapter.

Prompt	Responses
Library File:	List filename of library to be manipulated. (The default is the filename extension .LIB.)
Operation:	List command character(s) followed by module name(s) or object filename(s). (The default is no changes. The default object filename extension is .OBJ.)
List file:	List filename for a cross-reference listing file. (The default is NUL; i.e., no file.)

Remarks

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

Method 2: Command Line

To start MS-LIB with method 2, type:

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, or asterisk (+ , − , or *). 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 .LIB, 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?

Type Yes to create a new library file. Type No to abort the library session.

operations is a command to delete a module, append an object file as a module, or extract a module as an object file from the library file. Use the three command characters plus, minus, and asterisk to direct MS-LIB to append, delete, or extract modules and object files. (The default object filename extension is .OBJ).

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.

Remarks

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

Examples

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). The MS-LIB semicolon command character indicates that MS-LIB should use the default responses for the remaining prompts. Refer to "Command Characters" in Chapter 2 for more information.

LIB PASCAL

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

LIB PASCAL,PASCROSS.PUB

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

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

Method 3: Response File

To start MS-LIB with method 3, type:

LIB *@filespec*

Where

filespec is the name of a response file. A response file contains answers to the MS-LIB prompts. Method 3 permits you to conduct the MS-LIB session without user responses to the MS-LIB prompts.

Note

Before using method 3 to start MS-LIB, you must first create a response file.

A response file has one text line 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 you would for responses typed on the keyboard.

When the library session begins, each prompt will be displayed 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 will be made to the modules currently in the library file, and no cross-reference listing file will be created.) If you type 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 type a library filename, a carriage return, a comma, and then a list filename, MS-LIB will perform its consistency check of the library file, then produce the cross-reference listing file.

Example

```
PASCAL
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes MS-LIB to delete the module HEAP from the PASCAL.LIB library file; extract the module FOIBLES and place it 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.

COMMAND PROMPTS

You command MS-LIB by typing responses to three text prompts. After you have typed 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. You will see the operating system prompt when MS-LIB has finished the library session successfully. If the library session is unsuccessful, MS-LIB will display 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).

Command Prompts

Library File: Type 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 type 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 type 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 displayed.

If the filename you type does not exist, MS-LIB will display the prompt:

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

You must type either Yes or No.

Operation: Type one of the three command characters for manipulating modules (+, −, *); followed immediately (no space) by the module name or the object filename. The plus sign appends an object file as the last module in the library file (see further discussion under the description of plus sign in the next section). The minus sign deletes a module from the library file. The 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), type an ampersand (&) as the last character on the line. MS-LIB will repeat the Operation: prompt, which permits you to type additional module names and object filenames.

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

More information about modules is given in the description of each command character.

List file: If you want a PUBLIC symbols cross-reference list for the modules in the library file, type the name of a file in which you want MS-LIB to place the cross-reference listing. If you do not type a filename, no cross-reference listing is generated.

The response to the List file: prompt is a file specification. You can specify a drive (or device) designation and a filename extension with the filename. 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 typing 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.

COMMAND CHARACTERS

MS-LIB provides six command characters. Three of the command characters are required in response to the Operation: prompt. The other three command characters provide you with helpful commands to MS-LIB.

Plus sign Use the plus sign (+), followed by an object filename, to append the object file as the last module in the library named in response 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
```

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

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

Asterisk — Use the asterisk, followed by a module name, to extract the module from the library file and place it into a separate object file. The module will still exist in the library. (The extraction process copies 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 response to the Operation: prompt of:

```
*CURSOR
```

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

```
A:CURSOR.OBJ
```

The drive designation and filename extension cannot be overridden. You can, after the file is created, 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.

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

Remarks

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

Example

```
Library file: FUN
Operation: +CURSOR;
```

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

Ampersand

Use the ampersand to extend the current line. This command character is only used in response to the Operation: prompt. The number of modules you can append is limited only by disk space. The number of modules you can replace or extract is also limited only by disk space. The number of modules you can delete is limited by the number of modules in the library file.

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, and then you can type more responses. For example:

```
Library File: FUN
Operation: +CURSOR-HEAP+HEAP*FOIBLES&
Operation: *INIT+ASSUME+RIDE;
```

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 that MS-LIB allows you to type your Operation: responses in any order. You may use the ampersand character as many times as needed.

CTRL BREAK Use **CTRL BREAK** to abort the library session at any time. If you type an incorrect response, such as the wrong filename or module name, or an incorrectly spelled filename or module name, you must press **CTRL BREAK** to exit MS-LIB; then you must restart MS-LIB. If the error has been typed and you have not pressed the **CR** key, you may delete the erroneous characters for that line only.

Summary of Command Characters

Character	Action
+	Appends an object file as the last module
–	Deletes a module from the library
*	Extracts a module and places in an object file
;	Use default responses to remaining prompts
&	Extends current physical line; repeats command prompt
CTRL BREAK	Aborts library session

The order of precedence (see Example, method 3) is: delete (–), then extract (*), and then append (+).

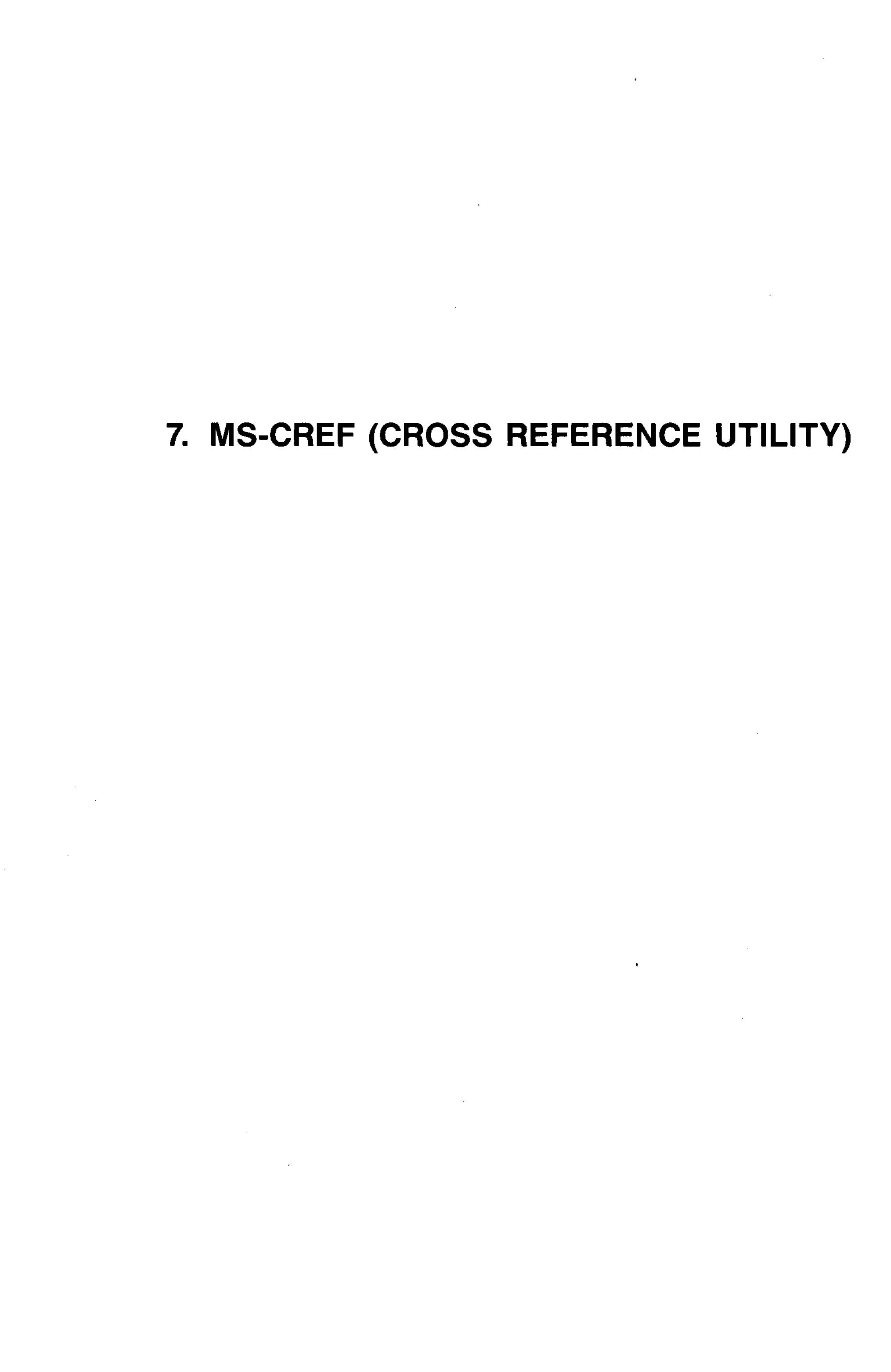
ERROR MESSAGES

The following are MS-LIB error messages:

ERROR MESSAGE	CAUSE
symbol is a multiply defined PUBLIC. Proceed?	Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol. You can remove the PUBLIC declaration from one of the object modules and recompile or reassemble. If you respond No, the library will be left in an indeterminate state.

Error Message	Cause
Allocate error on VM.TMP	Out of disk space
Cannot create extract file	No room in directory for extract file
Cannot create list file	No room in directory for library file
Cannot nest response file	@filespec in response (or indirect) file
MS-LIB cannot open VM.TMP	There is no room for VM.TMP in disk directory
Cannot write library file	Out of disk space
Close error on extract file	Out of disk space
Error: An internal error has occurred	
Fatal Error: Cannot open input file	You mistyped an object filename
Fatal Error: Module is not in the library	You tried to delete a module that is not in the library
Input file read error	Bad object module or faulty disk
Invalid object module/library	Bad object module and/or library

Error Message	Cause
Library Disk is full	No more room on disk
Listing file write error	Out of disk space
No library file specified	No response to Library File: prompt
Read error on VM.TMP	Disk not ready for read
Symbol table capacity exceeded	Too many public symbols (about 30K chars in symbols)
Too many object modules	More than 500 object modules
Too many public symbols	1024 public symbols maximum
Write error on library/extract file	Out of disk space
Write error on VM.TMP	Out of disk space



7. MS-CREF (CROSS REFERENCE UTILITY)

ABOUT THIS CHAPTER

The MS-CREF (Microsoft Cross-Reference) utility is designed to help the user in debugging assembly language programs.

MS-CREF outputs two listings:

- An alphabetical listing of all the symbols to a special file created by the assembler.
- A cross-reference listing giving symbol locations.

This chapter tells you how to make these listings. With them you can quickly locate all occurrences of any symbol in your source file, speeding your search and allowing faster debugging.

CONTENTS

INTRODUCTION 7-1

OVERVIEW OF MS-CREF 7-1

THE CROSS-REFERENCE FILE 7-1

RUNNING MS-CREF 7-2

HOW TO CREATE A CROSS-REFERENCE FILE 7-2

HOW TO START MS-CREF 7-3

METHOD 1: PROMPTS 7-3

METHOD 2: COMMAND LINE 7-5

COMMAND CHARACTERS 7-6

FORMAT OF CROSS-REFERENCE LISTINGS 7-6

ERROR MESSAGES 7-8

FORMAT OF MS-CREF COMPATIBLE FILES 7-9

MS-CREF FILE PROCESSING 7-10

FORMAT OF SOURCE FILES 7-10

INTRODUCTION

The MS-CREF (Cross Reference) utility outputs an alphabetical listing of all the symbols to a special file created by the assembler. With this listing, the user can quickly locate all occurrences of any symbol in his or her source program by line number.

MS-CREF also produces a cross-reference listing, giving the user symbol locations, speeding the search and allowing faster debugging.

The MS-CREF listing is used with the symbol table produced by the assembler.

The symbol table listing shows the value, type, and length of each symbol. This information is needed to correct erroneous symbol definitions or uses.

OVERVIEW OF MS-CREF

Operating with MS-CREF requires two steps:

- First, create a cross-reference file
- Second, start MS-CREF

THE CROSS-REFERENCE FILE

Before running CREF, you must first create a cross-reference file. This is done using the assembler. MS-CREF then converts this cross-reference file (which has the filename extension .CRF) 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 where the symbol occurs in the source program. The line numbers are listed in ascending sequence. The line number where the symbol is defined is indicated by a # symbol.

Figure 7-1 illustrates the MS-CREF operation.

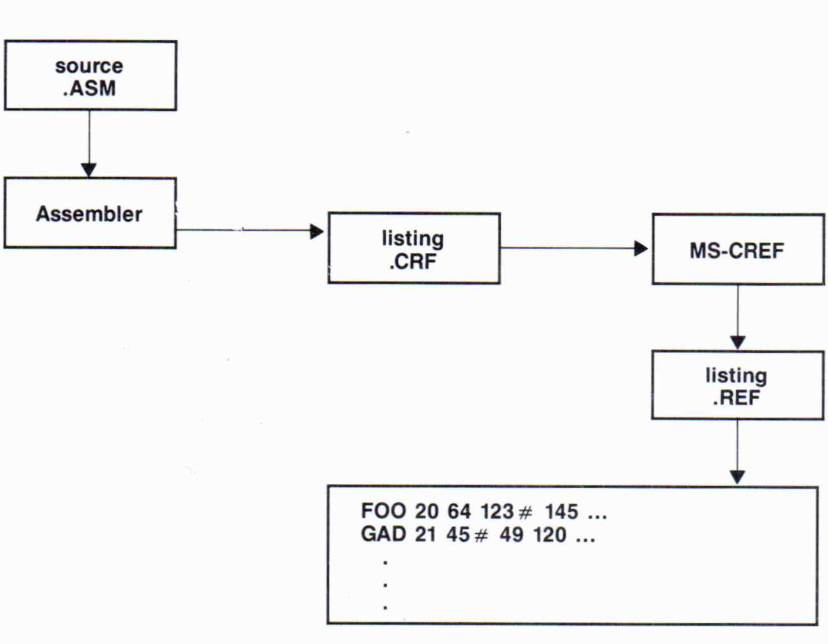


Fig. 7-1 MS-CREF Operation

RUNNING MS-CREF

Before running MS-CREF you must first create a cross-reference file, using the assembler.

HOW TO CREATE A CROSS-REFERENCE FILE

A cross-reference file is created during an assembly session. To create a cross-reference file, use the Macro Assembler and answer the fourth command prompt with the name of the cross-reference file you want to create.

The fourth assembler prompt is:

Cross-reference [NUL.CRF]:

If you do not type a filename in response to this prompt, or if you use the default response, the assembler will not create a cross-reference file. Therefore, you MUST type a filename if you want to create a cross-reference file.

You may also specify which drive or device you want the file saved on, and the filename extension (if different from .CRF). If you assign a filename extension other than .CRF, you must specify the filename extension when naming the file in response to the first MS-CREF prompt. (Refer to "How to Start MS-CREF", below, for a description of MS-CREF prompts).

You are now ready to use MS-CREF to convert the cross-reference file produced by the assembler into a cross-reference listing.

HOW TO START MS-CREF

MS-CREF may be started two ways. By the first method, you type the commands as answers to individual prompts. By the second method, you type all commands on the line used to start MS-CREF.

Method 1	CREF
Method 2	CREF *crffile, listing*

Tab. 7-2 Summary of Methods to Start MS-CREF

METHOD 1: PROMPTS

To start MS-CREF using prompts, type:

CREF

MS-CREF will be loaded into memory. Then, MS-CREF displays 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

Cross reference [.CRF]:

Type the name of the cross-reference file you want MS-CREF to convert to a cross-reference listing. The filename is the name you specified when you directed the assembler 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 type 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 that extension when typing the filename.

Refer to "Format of MS-CREF Compatible Files", later in this chapter 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 an MS assembler.

Listing [crffile.REF]:

Type 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 your cross-reference listing to have the same filename as the cross-reference file but with the filename extension .REF, simply press the **CR** key when the Listing: prompt appears. If you want your cross-reference listing file to be named anything else, or to have any other filename extension, you must type a response following the Listing: prompt.

If you want the listing file placed on a drive or device other than the default drive, specify that drive or device when typing your response to the Listing: prompt.

METHOD 2: COMMAND LINE

To start MS-CREF using the command line, type:

CREF *crffile,listing*

MS-CREF will be loaded into memory. Then, MS-CREF converts your cross-reference file into a cross-reference listing.

The entries following CREF are responses to the command prompts. The *crffile* and *listing* fields must be separated by a comma.

The input *crffile* is the name of the cross-reference file produced by your assembler. MS-CREF assumes that the filename extension is .CRF. You may override this default 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

MS-CREF will be aborted and the operating system prompt will appear.

The input *listing* is the name of the file you want to contain the cross-reference listing of symbols in your program.

To select the default filename and extension for the listing file, type a semicolon after the *crffile* name. Refer to "Command Characters" for more information on how to use the semicolon.

Examples

CREF FUN;

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 filename, extension, or destination, simply specify it when you type 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 disk in drive B:.

COMMAND CHARACTERS

MS-CREF provides two command characters.

Semicolon — 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.

CTRL BREAK — Use **CTRL BREAK** at any time to abort the MS-CREF session. If you make a mistake (for example, typing the wrong filename or incorrectly spelling a filename), you must press **CTRL BREAK** to exit MS-CREF, and then restart MS-CREF. If the error has been typed but you have not pressed the **CR** key, you may delete the erroneous characters, but for that line only.

FORMAT OF CROSS-REFERENCE LISTINGS

The cross-reference listing is an alphabetical list of all the symbols in your program. Each page begins with the title of the program or program module. Then the symbols are listed. 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 the symbol # appended to it.

An example of a cross-reference listing follows:

Example Of Cross-Reference Listing

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#
ENTX   PASCAL entry for initializing programs

 Symbol Cross-Reference     (# is definition)   Cref-2

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

ERROR MESSAGES

All errors cause MS-CREF to abort. Control is returned to the operating system.

All error messages are displayed in the following 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 Irrecoverable disk I/O error
101	Device name error Illegal device specification (for example, X:ARG.CRF)
103	Internal error *
104	Internal error *
105	Device offline Disk drive door open, no printer attached, or similar device is offline.
106	Internal error *
108	Disk full
110	File not found
111	Disk is write protected
112	Internal error *
113	Internal error *
114	Internal error *
115	Internal error *

* MS-CREF

FORMAT OF MS-CREF COMPATIBLE FILES

MS-CREF will process files other than those generated by the assembler, as long as the file conforms to the valid MS-CREF format.

MS-CREF FILE PROCESSING

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 one end-of-line character (see table below).

MS-CREF places a heading at the top of every page of the listing. The name MS-CREF uses is passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol (see table below). If MS-CREF encounters more than one title symbol in the source file, it will use 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 will be blank.

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 the following 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 is 58 lines; line length is 80 characters).

Control Symbols

The two tables below show the types of records that MS-CREF recognizes and the byte values and placement it uses to recognize record types.

Records have a control symbol (which identifies the record type) either as the first byte of the record or as the last byte.

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

Tab. 7-3 Records That Begin 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

Tab. 7-4 Records That End with a Control Symbol

* For all record types, the byte value represents a control character, as follows:

01 **CRTL A**
02 **CRTL B**
04 **CRTL D**
05 **CRTL E**
06 **CRTL F**
07 **CRTL G**

The Control Symbols are defined as follows:

Reference symbol

Record contains the name of a symbol that is referenced. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

Define symbol

Record contains the name of a symbol that is defined. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

End-of-line

Record is an end-of-line symbol character only (04H or **CTRL D**).

End-of-file

Record is the end-of-file character (1AH).

Title defined

ASCII characters of the title are 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 will be left blank.

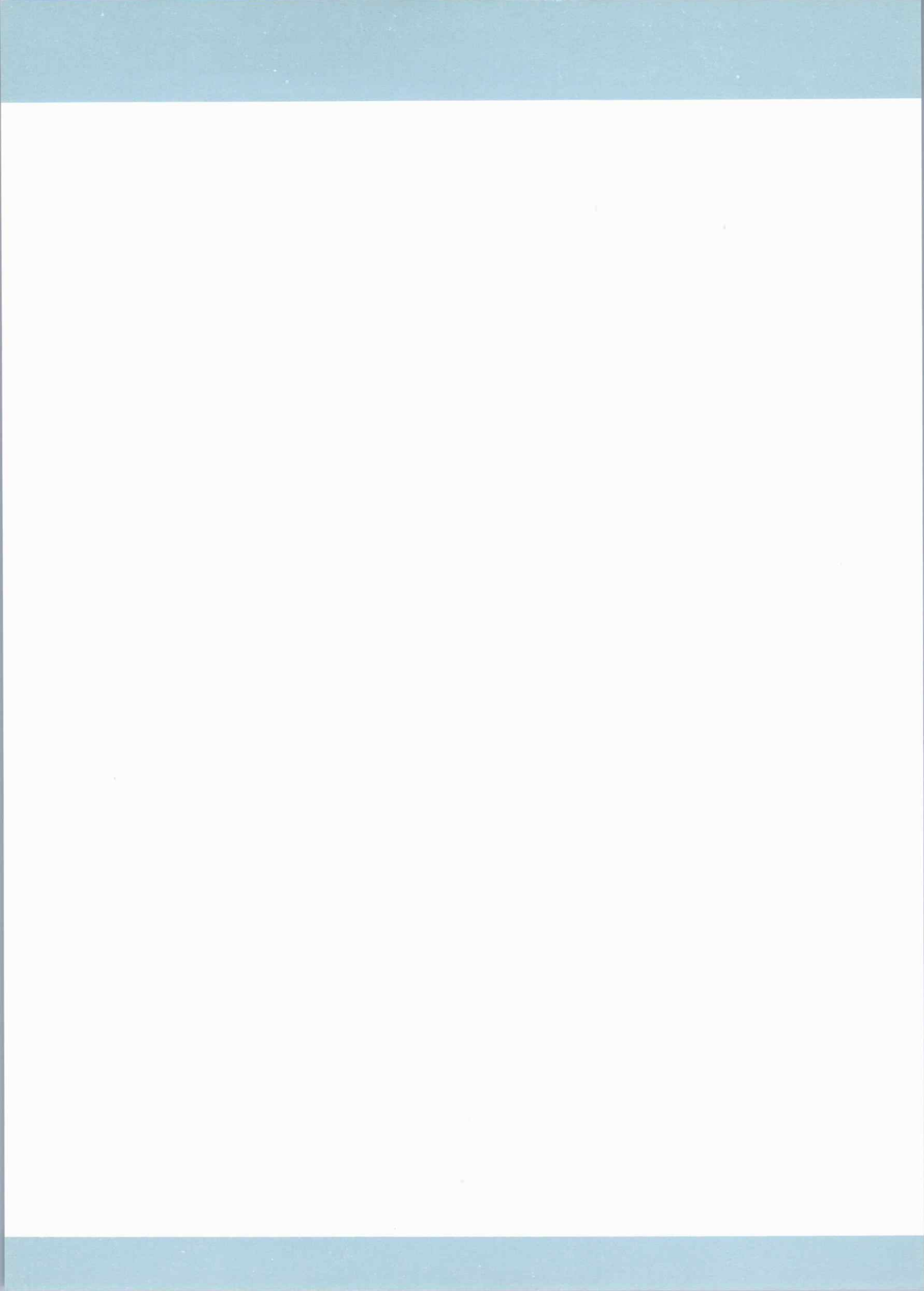
Page length/line length

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.

The following table illustrates CRF file record contents by byte and length of record.

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

Tab. 7-5 Summary of CRF File Record Contents



A. MACRO ASSEMBLER MESSAGES

ABOUT THIS APPENDIX

This appendix contains a list of Macro Assembler messages; both error messages and operating messages.

CONTENTS

OVERVIEW	A-1
OPERATING MESSAGES	A-1
ERROR MESSAGES	A-1
I/O HANDLER ERRORS	A-13
RUNTIME ERRORS	A-14
NUMERICAL ORDER LIST OF ERROR MESSAGES	A-15

OVERVIEW

Most of the messages output by Macro Assembler are error messages. The non-error messages output by Macro Assembler are the banner Macro Assembler displays when first started, the command prompt messages, and the end of (successful) assembly message. These non-error messages are classified here as operating messages. The error messages are classified as assembler errors, I/O handler errors, and runtime errors.

OPERATING MESSAGES

Command Prompts

Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing [NUL.LST]:
Cross reference [NUL.CRF]:

End of Assembly Message

Warning Errors	Fatal Errors	
n	n	(n=number of errors)

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 that 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.

LI "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 the closing 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.

8087 opcode can't be emulated (Code 84)

Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

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 is 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)

UP 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)

Illegal use of index display.

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 you furnished a symbol -- was not a register.

Must be segment or group (Code 20)

Expecting segment or group and something else was specified.

Must be structure field name (Code 37)

Expecting a structure field name but received something else.

Must be symbol type (Code 22)

Must be WORD, DW, 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 instruction where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)

Illegal use of the subtraction operator.

Operand must have segment (Code 43)

Illegal use of SEG directive.

Operand must have size (Code 35)

Expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)

Access of operand is impossible because it is not in the current IP segment.

Operand types must match (Code 31)

Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

Operand was expected (Code 27)

Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)

Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)

In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

Override with DUP is illegal (Code 79)

In a STRUC initialization statement, you tried to use DUP in an override.

Phase error between passes (Code 6)

The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes (see in Chapter 5 "Macro Assembler Command 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 to +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 your dealer.

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

CODE	MESSAGE
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 ? (indeterminate) bad (Code 75)

CODE	MESSAGE
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
84	8087 opcode can't be emulated
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

B. ASCII CHARACTER CODES

ABOUT THIS APPENDIX

This appendix consists of a table of the ASCII character codes.

ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	033	21H	!
001	01H	SOH	034	22H	"
002	02H	STX	035	23H	#
003	03H	ETX	036	24H	$
004	04H	EOT	037	25H	%
005	05H	ENQ	038	26H	&
006	06H	ACK	039	27H	'
007	07H	BEL	040	28H	(
008	08H	BS	041	29H	)
009	09H	HT	042	2AH	*
010	0AH	LF	043	2BH	+
011	0BH	VT	044	2CH	,
012	0CH	FF	045	2DH	–
013	0DH	CR	046	2EH	.
014	0EH	SO	047	2FH	/
015	0FH	SI	048	30H	0
016	10H	DLE	049	31H	1
017	11H	DC1	050	32H	2
018	12H	DC2	051	33H	3
019	13H	DC3	052	34H	4
020	14H	DC4	053	35H	5
021	15H	NAK	054	36H	6
022	16H	SYN	055	37H	7
023	17H	ETB	056	38H	8
024	18H	CAN	057	39H	9
025	19H	EM	058	3AH	:
026	1AH	SUB	059	3BH	;
027	1BH	ESCAPE	060	3CH	<
028	1CH	FS	061	3DH	=
029	1DH	GS	062	3EH	>
030	1EH	RS	063	3FH	?
031	1FH	US	064	40H	@
032	20H	SPACE			

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

Table B-1 ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[	123	7BH	{
092	5CH	\	124	7CH	|
093	5DH	]	125	7DH	}
094	5EH	^	126	7EH	~
095	5FH	_	128	7FH	DEL
096	60H	`			

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

Tab. B-1 ASCII Character Codes

C. TABLE OF MACRO ASSEMBLER DIRECTIVES

ABOUT THIS APPENDIX

This appendix lists all the Macro Assembler directives.

CONTENTS

MEMORY DIRECTIVES C-1

MACRO DIRECTIVES C-2

CONDITIONAL DIRECTIVES C-2

LISTING DIRECTIVES C-2

ATTRIBUTE OPERATORS C-3

PRECEDENCE OF OPERATORS C-4

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

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 numberv

CONDITIONAL DIRECTIVES

ELSE

IF *exp*

IFB *arg*

IFDEF *symbol*

IFDIF *arg1*, *arg2*

IFE *exp*

IFIDN *arg1*, *arg2*

IFNB *arg*

IFNDEF *symbol*

IF1

IF2

LISTING DIRECTIVES

.CREF

.LALL

.LFCOND

.LIST

%OUT *text*

PAGE *exp*

.SALL
.SFCOND
SUBTTL *text*
.TFCOND
TITLE *text*
.XALL
.XCREF
.XLIST

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*

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)

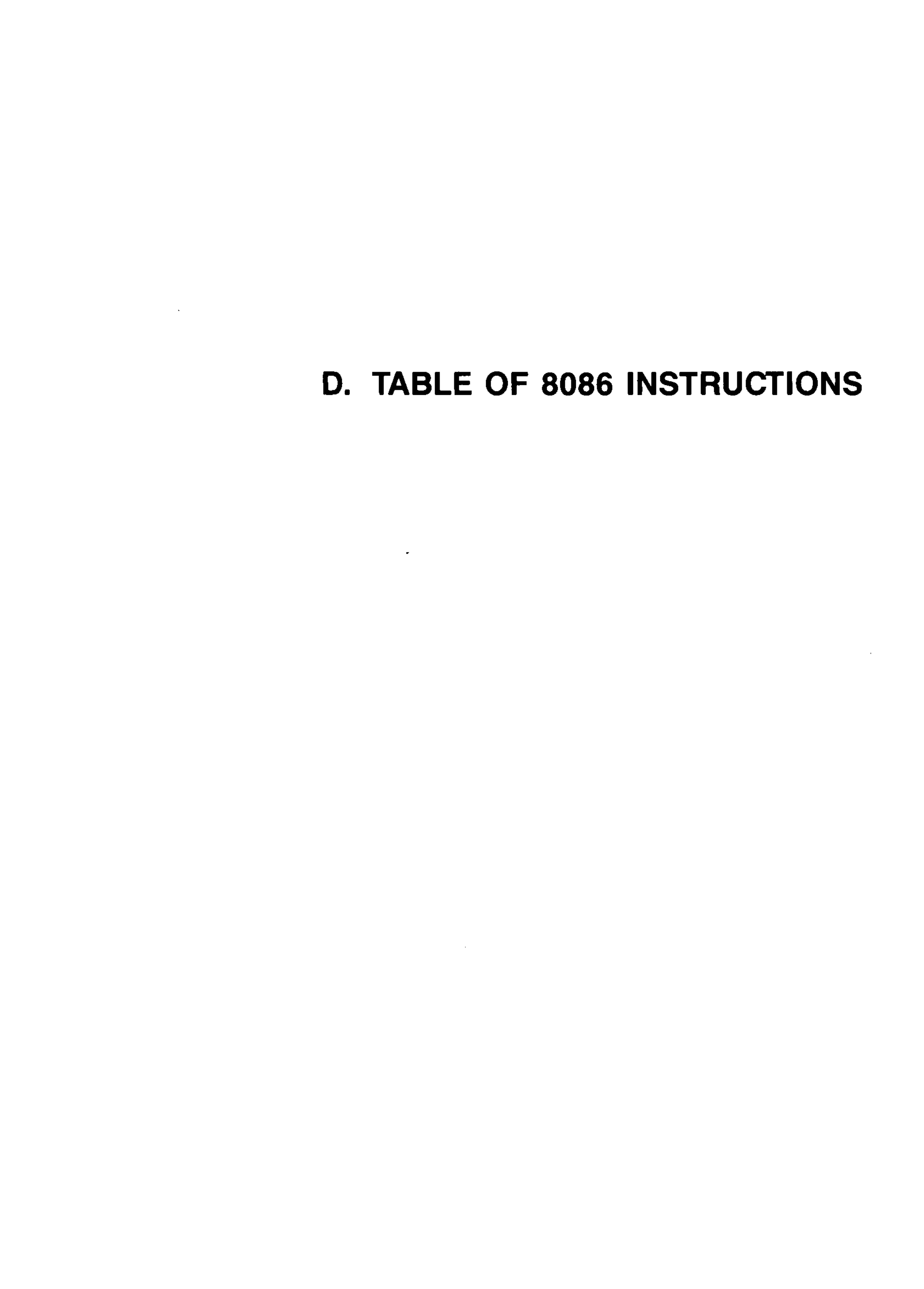
7. EQ, NE, LT, LE, GT, GE

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT,.TYPE



D. TABLE OF 8086 INSTRUCTIONS

ABOUT THIS APPENDIX

This appendix contains the Macro Assembler instruction mnemonics.

CONTENTS

8086 INSTRUCTIONS MNEMONICS (ALPHABETICAL) D-1

8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE D-4

ABOUT THIS APPENDIX

The Macro Assembler 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.

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

Table D-1 8086 Instruction Mnemonics, Alphabetical

Mnemonic	Full Name
IDIV	Integer divide
IMUL	Integer multiply
IN	Input byte or word
INC	Increment
INT	Interrupt
INTO	Interrupt on overflow
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

Table D-1 8086 Instruction Mnemonics, Alphabetical (cont.)

Mnemonic	Full Name
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
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

Table D-1 8086 Instruction Mnemonics, Alphabetical (cont.)

Mnemonic	Full Name
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

Table D-1 8086 Instruction Mnemonics, Alphabetical (cont.)

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	

In addition, add to the arguments a sign extent for word immediate.

CALL and JUMP type instructions

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

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)

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	RET	NOP	STC	XLATB
CLC	CWD	LAHF	POPF	STD	

Load instructions

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

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)
	OP r/m

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
IDIV	r/m, except NEG)
MUL	
IMUL	
NEG	(NEG implies AX OP NOP)
NOT	

Interrupt instruction

Mnemonic	Argument Types
INT	INT 3 (value 3 is one-byte instruction) INT byte-immed

Exchange instruction

Mnemonic	Argument Types
XCHG	XCHG accum,reg XCHG reg,accum XCHG reg,r/m 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	CMPS byte-word,byte-word (CMPS right operand is ES)
LODS	LODS byte/word,byte/word (LODS one argument = no ES)
MOVS	MOVS byte/word,byte/word (MOVS left operand is ES)
SCAS	SCAS byte/word,byte/word SCAS one argument = ES)
STOS	STOS byte/word,byte/word (STOS one argument = ES)

Repeat prefix to string instructions

Mnemonics

LOCK
REP
REPE
REPZ
REPNE
REPNZ

E. SYNTAX NOTATION

ABOUT THIS APPENDIX

This appendix contains an explanation of the syntax notation used in this manual.

CONTENTS

SYNTAX NOTATION E-1

SYNTAX NOTATION

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

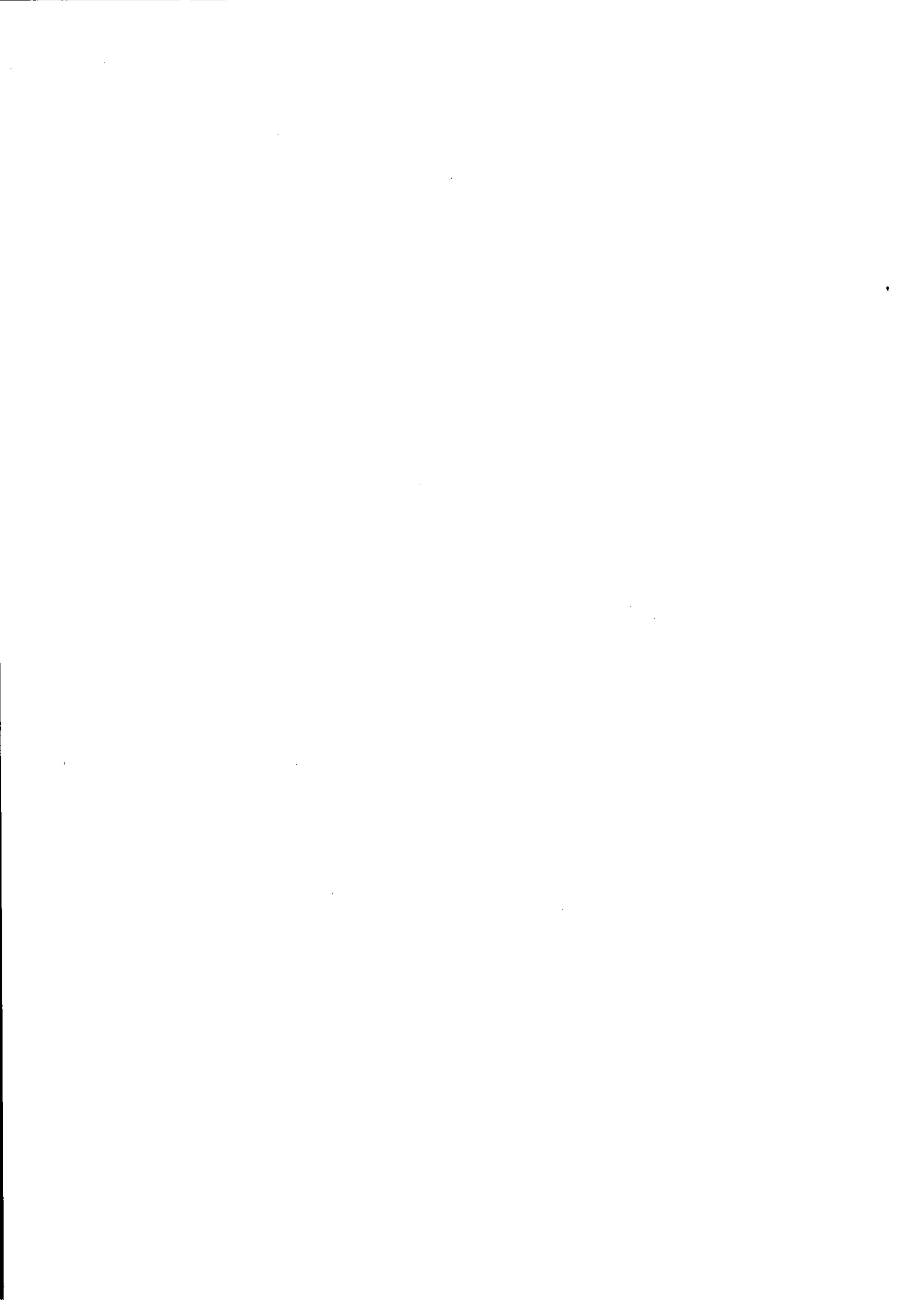
[ ]	Square brackets indicate that the enclosed entry is optional.
{ }	Braces indicate that you have a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical stroke (''or'' sign).
...	Ellipses 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.
lowercase	lowercase letters and words in italics represent ''variable information'' that the user must provide.
-	Hyphen, to join multiple-name parameters.

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

NOTICE

Ing. C. Olivetti & C., S.p.A. reserves the right to make any changes in the product described in this manual at any time and without notice.

This manual is licensed to the Customer under the conditions contained in the User License enclosed with the Program to which the manual refers.



Code 4001470 E (1)
Printed in Italy

olivetti