Programmer's Tool Kit

Volume II

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing February, 1983.

ISBN 0-88182-024-5 Printed in U.S.A.

CONTENTS Programmer's Tool Kit, Volume II

1. Introduction

2. MS-LIB

3. MS-LINK

4. MS-CREF

5. DEBUG

6. MACRO-86

7. SYSELECT

CONTENTS Programmer's Tool Kit, Volume I

1. FABS/86

2. AUTOSORT/86

3. PMATE-86

4. EFONT

5. KEYGEN

6. MODCON

IMPORTANT SOFTWARE DISKETTE INFORMATION

For your own protection, do not use this product until you have made a backup copy of your software diskette(s). The backup procedure is described in the user's guide for your computer.

Please read the DISKID file on your new software diskette. DISKID contains important information including:

- The product name and revision number.
- The part number of the product.
- The date of the DISKID file.
- A list of the files on the diskette, with a description and revision number for each one.
- Configuration information (when applicable).
- Release notes giving special instructions for using the product.
- Information not contained in the current manual, including updates, additions, and deletions.

To read the DISKID file onscreen, follow these steps:

1. Load the operating system.
2. Remove your system diskette and insert your new software diskette.
3. Enter —

   TYPE DISKID

   and press Return.
4. The contents of the DISKID file is displayed on the screen. If the file is large (more than 24 lines), the screen display will scroll. Type ALT-S to freeze the screen display; type ALT-S again to continue scrolling.

INTRODUCTION

CONTENTS

1. Major Features of Volume II 1-1
1.1 MS-LINK Linker Utility 1-1
1.2 MS-LIB Library Manager 1-1
1.3 MS-CREF Cross-Reference Facility 1-1
1.4 MS-DEBUG Debug Utility 1-2
1.5 MACRO-86 Macro Assembler 1-2
1.6 SYSELECT 1-2

2. Using Volume II 2-1
2.1 Syntax Notation 2-1
2.2 Learning More About Assembly Language Programming 2-3

3. Overview of Program Development 3-1

EXHIBITS

2a: Using the Programmer's Tool Kit, Volume II, Software Package 2-2
2b: Syntax Notation 2-2
3a: Program Development 3-2

CHAPTERS

1. Major Features of Volume II 1

2. Using Volume II 2

3. Overview of Program Development 3

1

MAJOR FEATURES OF VOLUME II

MS-LINK LINKER UTILITY 1.1

- MS-LINK is a virtual linker that can link programs that are larger than available memory.
- MS-LINK produces relocatable executable object code.
- MS-LINK handles user-defined overlays.
- MS-LINK performs 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 1.2

- 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 FACILITY 1.3

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

1.4 MS-DEBUG DEBUG UTILITY

DEBUG is a debugging program used to provide a controlled testing environment for binary and executable object files. Note that text editors are used to alter source files; DEBUG is the text editor's counterpart for binary files. DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It allows you to alter the contents of a file or the contents of a CPU register, and then to reexecute a program immediately to check the validity of the changes.

1.5 MACRO-86 MACRO ASSEMBLER

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

MACRO-86 supports most of the directives found in Microsoft's MACRO-86 Macro Assembler. Macros and conditionals are Intel 8086 standard.

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

Some prefer relaxed typing. If you enter a typeless operand for an instruction that accepts one type of operand, MACRO-86 assembles the statement correctly instead of returning an error message.

1.6 SYSELECT

SYSELECT is an operating system builder that allows you to create an operating system with custom keyboard tables, character sets, default printer types, serial port specifications, logos, and banners.

2

USING VOLUME II

The Sections in this volume are designed to be used both as a set and individually. Each Section is mostly self-contained and refers to the other Sections only at junctures in the software. The following overview 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 overviews in each of the Sections.

SYNTAX NOTATION 2.1

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

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, you must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, you must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that you have a choice between two or more entries. You must choose at least one of the entries enclosed in braces unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Uppercase letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown. Refer to Exhibit 2b.

Exhibit 2a: Using the Programmer's Tool Kit, Volume II, Software Package

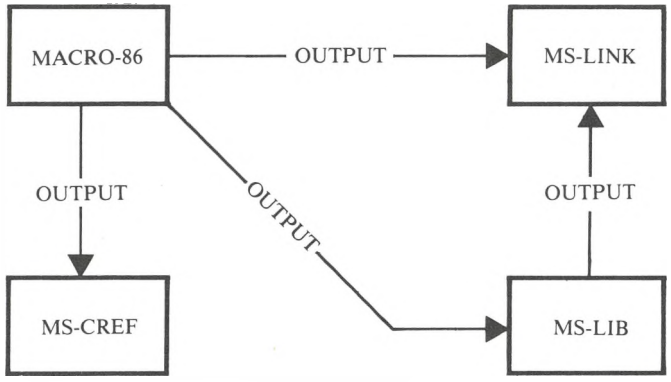
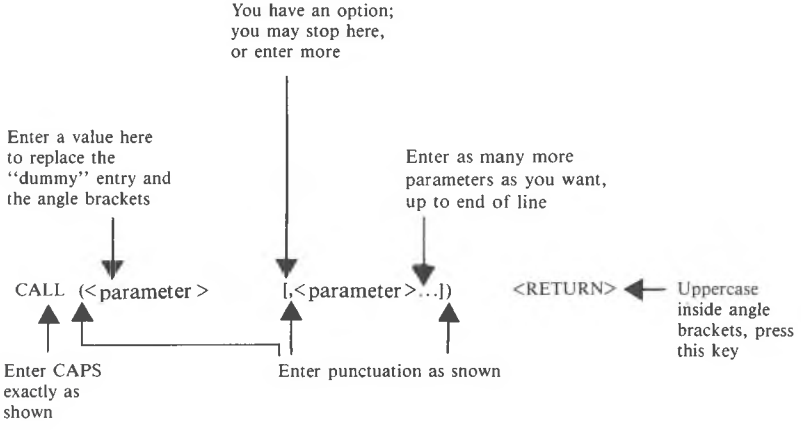


Exhibit 2b: Syntax Notation



LEARNING MORE ABOUT ASSEMBLY LANGUAGE PROGRAMMING 2.2

These Sections explain how to use the *Programmer's Tool Kit,* Volume II, but they do not teach you how to program in assembly language.

We assume that the user of this volume has had some experience programming in assembly language. If you do not have any experience, we suggest two courses:

1. Gain some experience on a less sophisticated assembler.

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

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

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

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

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

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

3

OVERVIEW OF PROGRAM DEVELOPMENT

This overview describes generally the steps of program development. Each step is described fully in the individual product Sections. The numbers in parentheses match the numbers in Exhibit 3a.

1. Use an MS-DOS editor to create an 8086 assembly language source file. Give the source file the filename extension .ASM (MACRO-86 recognizes .ASM as default).

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

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

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

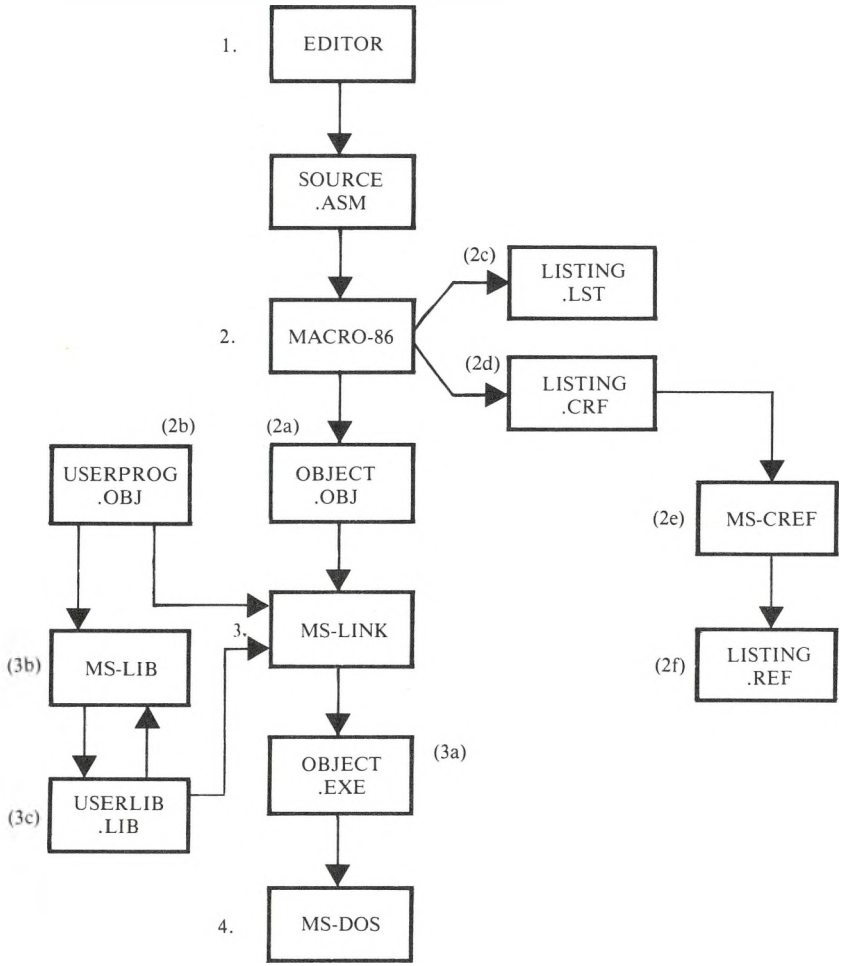
   (2d) A cross-reference file, a special file with special control characters that allow MS-CREF (2e) to create a list showing the source line number of every symbol's definition and all references to it (2f). When a cross-reference file is created, the normal listing file (with the .LST extension) has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

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

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

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

Exhibit 3a: Program Development



MS-LIB

First VICTOR printing February, 1983.

ISBN 0-88182-012-1 Printed in U.S.A.

CONTENTS

1. Introduction 1-1

1.1 Features and Benefits of MS-LIB 1-1
1.2 Overview of MS-LIB Operation 1-1

2. Running MS-LIB 2-1

2.1 Invoking MS-LIB 2-1

Method 1: LIB 2-1
Method 2: LIB <library><operations>,<listing > 2-2
Method 3: LIB @<filespec> 2-4

2.2 Command Prompts 2-5

Library File 2-5
Operation 2-6
List File 2-7

2.3 Command Characters 2-7

3. Error Messages 3-1

EXHIBITS

2a: Summary of Command Prompts 2-2
2b: Summary of Command Characters 2-2

CHAPTERS

1. Introduction 1

2. Running MS-LIB 2

3. Error Messages 3

1

INTRODUCTION

FEATURES AND BENEFITS OF MS-LIB 1.1

MS-LIB creates and modifies the library files used by the MS-LINK Linker Utility. MS-LIB can add object files to a library, delete modules from a library, or extract modules from a library and place them into separate object files.

MS-LIB lets you create general or special libraries for a variety of programs. With MS-LIB you can create a library for a language compiler, or you can create a library for one program only (allowing very fast linking and possibly more efficient execution).

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

The command scanner in MS-LIB is the same one used in MS-LINK, MS-Pascal, MS-FORTRAN, and other 16-bit Microsoft products. If you have used any of these programs, MS-LIB should seem familiar. Command syntax is straightforward, and MS-LIB prompts you for any command it needs.

OVERVIEW OF MS-LIB OPERATION 1.2

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

- Deleting modules.

- Extracting modules and placing them into separate object files.
- Appending object files as modules of a library.
- Replacing modules in the library file with a new module.
- Creating library files.

During each library session, MS-LIB deletes or extracts modules and then appends new ones. In a single operation, MS-LIB reads each module into memory, checks it for consistency, and writes it back to the file. If you delete a module, MS-LIB reads in that module but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places that module at the end of the last module written to the file.

When MS-LIB has read through the entire library file, it appends any new modules to the end of the file. Then, MS-LIB creates an index to the file (which MS-LINK uses to find modules and symbols in the library file) and outputs a cross reference listing of the PUBLIC symbols in the library, if you request such a listing. (Building the library index may take extra time, up to 20 seconds in some cases.)

For example:

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

deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. This order of execution keeps MS-LIB from getting confused when a new version of a module replaces one already in the library file.

2

RUNNING MS-LIB

Two types of commands are used in running MS-LIB: a command to invoke MS-LIB and commands issued as you respond to command prompts. Usually, commands to MS-LIB are entered at the keyboard; however, answers to the prompts can be contained in a response file. MS-LIB also uses command characters, either as a required part of commands or to assist you while entering them.

INVOKING MS-LIB 2.1

MS-LIB is invoked in three ways. With the first method, you enter commands as answers to individual prompts. With the second, you enter commands on the line used to invoke MS-LIB. The third method involves creating a response file that contains all the necessary commands.

METHOD 1: LIB

Enter:

```
LIB
```

MS-LIB is loaded into memory and returns a series of three text prompts, one at a time. Your answers to the prompts tell MS-LIB to perform specific tasks.

Exhibits 2a and 2b summarize the command prompts and command characters used by MS-LIB. They are fully described later in this chapter.

Exhibit 2a: Summary of Command Prompts

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

Exhibit 2b: Summary of Command Characters

CHARACTER	ACTION
+	Append an object file as the last module
–	Delete a module from the library
*	Extract a module and place it in an object file
;	Use default responses to remaining prompts
&	Extend current physical line; repeat command prompt
^C	Abort library session

METHOD 2: LIB<library><operations>,<listing>

Enter:

LIB <library><operation>,<listing>

where:

library is the name of a library file. MS-LIB assumes a file extension of .LIB.If the file name you give does not exist, MS-LIB prompts you:

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

Enter Y to create a new library file. Enter N to abort the library session.

operation can be deleting a module, appending an object file as a module, or extracting a module as an object file from the library file. Use the three command characters (+, −, and *) to tell MS-LIB what to do with each module or object file.

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

All the entries following LIB are responses to the command prompts. The library and operations fields and all operations entries must be separated by command characters. If a cross reference listing is wanted, the name of the file must be separated from the last operations entry by a comma (,). If you want to select the default value for the remaining field(s), enter a semicolon (;).

If you enter a library file name followed by a semicolon, MS-LIB reads through the library file and performs a consistency check. No changes are made to the modules in the library file. If you enter a library file name followed immediately by a comma and a list file name, MS-LIB performs its consistency check of the library file, then produces the cross reference listing file.

Examples:

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

deletes the module HEAP from the library file PASCAL.LIB, then appends the object file HEAP.OBJ as the last module of PASCAL.LIB (the module will be named HEAP).

If you want to do several operations during a library session, use the ampersand (&) command character to extend the command line. This lets you enter additional object file names and module names. Always remember to include one of the operations command characters before the name of each module or object file name.

LIB PASCAL ↵

performs a consistency check of library file PASCAL.LIB. No other action is performed.

LIB PASCAL,PASCROSS.PUB

performs a consistency check of the library file PASCAL.LIB, then outputs a cross reference listing file named PASCROSS.PUB.

METHOD 3: LIB @<filespec>

Enter:

LIB @<filespec>

where:

filespec is the name of a response file. A response file contains answers to the MS-LIB prompts.

This method lets you conduct the MS-LIB session without interactive (direct) user responses to the MS-LIB prompts. Remember to create the response file before you use this method.

A response file contains text lines, one for each prompt. Responses must appear in the same order as the command prompts appear. Command characters are used just as they are when entering responses on the keyboard.

When the library session begins, each prompt is displayed along with the matching responses from the response file. If the response file does not contain answers for all the prompts, MS-LIB uses the default responses.

If you enter a library file name followed by a semicolon, MS-LIB reads through the library file and performs a consistency check. No changes are made to the modules in the library file.

Example:

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

This deletes the module HEAP from the PASCAL.LIB library file, and extracts the module FOIBLES. Then it creates an object file named A:FOIBLES.OBJ, and appends object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, MS-LIB creates a cross reference file named CROSSLST.

COMMAND PROMPTS 2.2

You command MS-LIB by entering responses to three text prompts. These ask 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. When you enter your response to one prompt, the next one appears. When the last prompt has been answered, MS-LIB does its library management functions without further command. When the library session is over, MS-LIB exits to the operating system. (You'll know that this has occurred when the operating system prompt appears on the screen.) If the library session is unsuccessful, MS-LIB returns the appropriate error message.

LIBRARY FILE

Enter the name of the library file that you want to manipulate. Unless you enter a file extension when you give the library file name, MS-LIB assumes that the file extension is .LIB. Because MS-LIB can manage only one library file at a time, you can enter only one file name in response to this prompt. Except for the semicolon command character, additional responses are ignored.

If you enter a library file name followed by a semicolon, MS-LIB does a consistency check and returns to the operating system. Any errors in the file are reported.

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

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


You must answer Y or N . If N , or any other character is entered, MS-LIB terminates and returns to the operating system.

OPERATION

Enter one of the three command characters for manipulating modules (+ , − , and *), followed immediately by the module name or the object file name. (Do not put a space between the command character and the module name or object file name.) If you choose the plus-sign command character, an object file is appended as the last module in the library file. A minus sign (−) deletes a module from the library file. Entering an asterisk (*) extracts a module from the library and places it into a separate object file having the same name as the module and file extension .OBJ.

Operations on modules and object file names can be entered in any order. When you have a large number of modules to manipulate, enter an ampersand (&) as the last character on the line. MS-LIB repeats the Operation prompt, allowing you to enter additional module names and object file names.

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

LIST FILE

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

The cross reference listing file contains two lists. The first list is an alphabetical listing of all PUBLIC symbols where 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.

When you respond to the list file prompt, you can specify (along with the file name) a drive or device designation and a file extension. If you want the file to have a file extension, you must specify it when entering the file name.

COMMAND CHARACTERS 2.3

MS-LIB has six command characters: three of these are required in responses to the Operation prompt; the other three give you additional commands.

+ When followed by an object file name, the plus sign appends the object file as the last module in the library specified at the library file prompt. MS-LIB assumes that the file extension is .OBJ. You can override this assumption by specifying another extension.

  MS-LIB strips the drive designation and extension from the object file specification, leaving only the file name. 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

  strips off the B: and the .OBJ, leaving only CURSOR. This becomes the name of the module added to the library file.

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

– Followed by a module name, a minus sign deletes that module from the library file. MS-LIB then "closes up" the file space left empty by the deletion. This cleanup action keeps the library file from containing a lot of empty space. Remember that new modules are added at the end of the file, not stuffed into space vacated by deleted modules.

* When followed by a module name, the asterisk makes a copy of that module and places the copy into a separate object file. (This process is called "extraction.")The module name is used as the file name. MS-LIB adds the default drive designation and the file 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**

extracts the module named CURSOR from the library file and copies it into an object file with the file specification of A:CURSOR.OBJ.

The drive designation and file extension cannot be overridden. However, you can rename the file and give it a new file extension, or you can copy the file to a new disk drive, giving a new file name and/or file extension.

; A single semicolon followed immediately by a Return selects default responses to the remaining prompts. This feature saves time and eliminates the need to answer additional prompts.

NOTE: Once you enter a semicolon, you can't respond to any of the remaining prompts in that library session. Do not use the semicolon if you only want to skip some of the prompts. In that case, use the carriage return instead.

Example:

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

The remaining prompts do not appear, and MS-LIB uses the default value (no cross reference file).

& MS-LIB can perform many functions during a single library session. 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 only by the number of modules in the library file. However, the line length for a response to any prompt is limited to the line length of your system. To enter a large number of responses to the Operation prompt, place an ampersand at the end of a line. MS-LIB displays the Operation prompt again, then you can enter more responses. You can use the ampersand as many times as you like.

For example:

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

MS-LIB deletes the module HEAP, extracts the modules FOIBLES and INIT (creating two files, A: FOIBLES.OBJ and A: INIT.OBJ), then appends the object files CURSOR, HEAP, ASSUME and RIDE.

^C Alt-C aborts the library session. If you enter an incorrect response, such as the wrong file name or an incorrectly spelled module name, press Alt-C to exit MS-LIB. Then, reinvoke MS-LIB and start over.

3

ERROR MESSAGES

These error messages are used by MS-LIB:

<symbol>is a multiply defined PUBLIC. Proceed?

Two modules define the same PUBLIC symbol. You need to confirm the removal of the definition of the old symbol. If you answer No, then the library is left in an undetermined state.

To correct this error condition, remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

Allocate error on VM.TMP

There is no space left on the disk.

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

Caused by "@filespec" in response (or indirect) file.

Cannot open VM.TMP

No room for VM.TMP in disk directory.

Cannot write library file

No space left on disk.

Close error on extract file

Out of space.

Error: An internal error has occurred

Contact Victor Technologies, Inc.

Fatal Error: Cannot open input file

Caused by mistyped object file name.

Fatal Error: Module is not in the library

This error occurs if you try 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.

Library Disk is full

No more room on disk.

Listing file write error

Out of space.

No library file specified

Occurs if you fail to respond to library file prompt.

Read error on VM.TMP

Disk not ready for read.

Symbol table capacity exceeded

Too many PUBLIC symbols.

Too many object modules

There are more than 500 object modules.

Too many PUBLIC symbols

1024 PUBLIC symbols maximum.

Write error on library/extract file

Out of space.

Write error on VM.TMP

Out of space.

MS-LINK

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing February, 1983.

ISBN 0-88182-013-X Printed in U.S.A.

CONTENTS

1. Introduction 1-1
   1.1 Features and Benefits of MS-LINK 1-1
   1.2 Overview of MS-LINK Operation 1-1
   1.3 Definitions 1-2
      Segment 1-2
      Group 1-3
      Class 1-3
   1.4 How MS-LINK Combines and Arranges Segments 1-3
   1.5 Files Used by MS-LINK 1-6
      Input Files 1-6
      Output Files 1-7
      VM.TMP File 1-7

2. Running MS-LINK 2-1
   2.1 Invoking MS-LINK 2-1
      Method 1: LINK 2-1
      Method 2: LINK <filenames> [/switches 2-4
      Method 3: LINK @<filespec> 2-5
   2.2 Command Prompts 2-6
      Object Modules 2-6
      Run File 2-6
      List File 2-7
      Libraries 2-7
   2.3 Switches 2-8
      /DSALLOCATE 2-9
      /HIGH 2-9
      /LINENUMBERS 2-9
      /MAP 2-10
      /PAUSE 2-10
      /STACK: <number> 2-10

3. Error Messages 3-1

EXHIBITS

1a: MS-LINK Operation 1-2
2a: LINK Command Prompts 2-2
2b: LINK Switches 2-2

CHAPTERS

1. Introduction .....

2. Running MS-LINK .....

3. Error Messages .....

1

INTRODUCTION

FEATURES AND BENEFITS OF MS-LINK 1.1

MS-LINK is a relocatable linker designed to link separately produced modules of 8086 object code. The object modules must be 8086 files only. MS-LINK can link files totalling 384K bytes.

MS-LINK is user-friendly. When a command needs to be issued (or when there is a choice of several commands), MS-LINK prompts you for that command. Your answers to the prompts are the commands. The MS-LINK output file (run file) is not bound to specific memory addresses and can be loaded and executed by your computer's operating system at any convenient address.

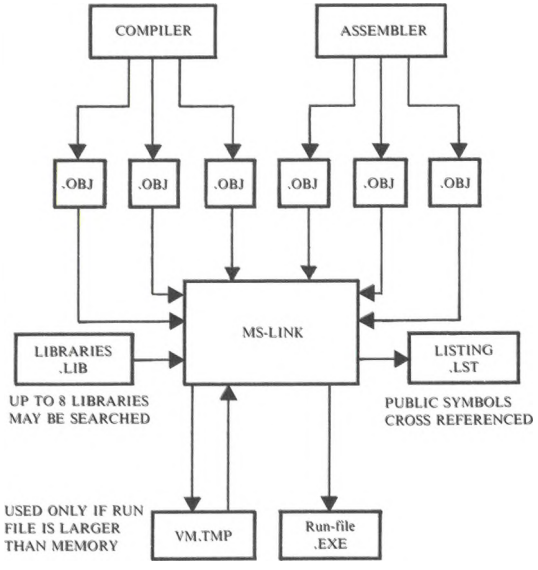
MS-LINK uses a dictionary-indexed library search method, which substantially reduces link time for sessions involving library searches.

OVERVIEW OF MS-LINK OPERATION 1.2

MS-LINK combines several object modules into one relocatable load module, or run file. As it combines modules, MS-LINK resolves external references between object modules and searches multiple library files for the definition of any unresolved external references. MS-LINK also produces a list file that shows the external references resolved and any error messages.

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

Exhibit 1a: MS-LINK Operation



1.3 DEFINITIONS

Three terms appear in some of the error messages listed in Chapter 2. An understanding of these terms will give you a good idea of how MS-LINK works.

SEGMENT

A segment is a contiguous area of memory up to 64K bytes in length that can be located anywhere in memory on a "paragraph" (16-byte) boundary. The contents of a segment are addressed by a segment register/offset pair.

GROUP

A group is a collection of segments that fits within 64K bytes of memory. The segments are named to a group by MS-LINK, by the compiler, or by you. You assign the group name while in the assembly language program except in high-level languages (BASIC, FORTRAN, COBOL, Pascal), where naming is done by the compiler.

The group is used for addressing segments in memory. Each group is addressed by a single segment register. The segments within the group are addressed by the segment register and an offset. MS-LINK checks to see that the object modules of a group meet the 64K-byte constraint.

CLASS

A class is a collection of segments used to control the order and relative placement of segments in memory. You assign the class name while in the assembly language program except for high-level languages (BASIC, FORTRAN, COBOL, Pascal), where naming is done by the compiler. Segments are named to a class at compile time or assembly time, and are loaded into memory contiguously. Within a class, segments are ordered as MS-LINK encounters them in the object files. One class precedes another in memory only if a segment in the first class precedes all segments in the second class in the input to MS-LINK.

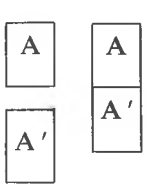
Classes can be loaded across 64K-byte boundaries and are divided into groups for addressing.

HOW MS-LINK COMBINES AND ARRANGES SEGMENTS 1.4

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

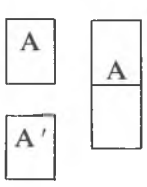
MS-LINK combines segments for these combine types as follows.

► **Private**



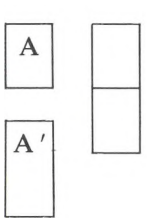
Private segments are loaded separately and remain separate. They can be contiguous physically but not logically, even if the segments have the same name. Each private segment has its own base address.

► **Public**



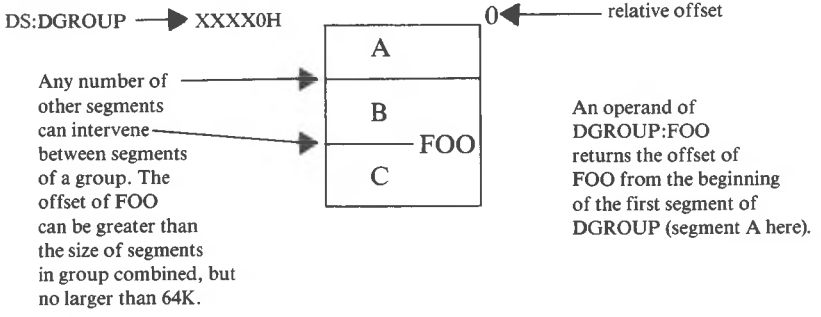
Public segments of the same name and same class are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class. (Stack and memory combine types are treated the same as public. However, the stack pointer is set to the first address of the first stack segment.)

► **Common**



Common segments of the same name and class 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.

If segments are placed into the assembler in a group, it provides offset addressing of items from a single base address for all segments in that group.



Segments are grouped by their class names. MS-LINK loads all the segments in the first class name encountered, then all the segments of the next class name encountered, and so on until all classes have been loaded.

If your program contains:	They will be loaded as:
A SEGMENT 'FOO'	'FOO'
B SEGMENT 'BAZ'	A
C SEGMENT 'BAZ'	E
D SEGMENT 'ZOO'	'BAZ'
E SEGMENT 'FOO'	B
	C
	'ZOO'
	D

If you are writing assembly language programs, you can control the ordering of classes in memory by writing a dummy module and listing it as the first entry after the MS-LINK Object Modules prompt. The dummy module declares segments into classes in the order you want the classes loaded.

WARNING: Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs. Let the compiler and the linker work in the normal way.

Example:

```
A  SEGMENT  'CODE'
A  ENDS
B  SEGMENT  'CONST'
B  ENDS
C  SEGMENT  'DATA'
C  ENDS
D  SEGMENT  STACK  'STACK'
D  ENDS
E  SEGMENT  'MEMORY'
E  ENDS
```

Make sure that you declare all classes to be used in your program in this module. If you don't, you lose absolute control over the ordering of classes.

If you want the memory combine type to be loaded as the last segments of your program, use this method. Just add MEMORY between SEGMENT and 'MEMORY ' in the E segment line above. However, these segments are loaded last only because you imposed this control on them, not because of any inherent capability of the linker or assembler operations.

1.5 FILES USED BY MS-LINK

MS-LINK uses several kinds of files. It works with one or more input files and produces two output files. MS-LINK can create a virtual memory file, and can search up to eight library files. For each type of file, you can give a three-part file specification. The MS-LINK file specification format is:

drv:filename.ext

where:

drv: is the drive designation. Legal drive designations are A: through O:. The colon is always required.

filename is any legal file name of up to eight characters.

.ext is a one to three character extension that describes the type of file. The period is always required.

INPUT FILES

If no extension is explicitly set, MS-LINK provides the following default extension for the input file:

File	Default Extension
Object	.OBJ
Library	.LIB

OUTPUT FILES

MS-LINK gives the output files the following default extensions:

File	Default Extension
Run	.EXE (cannot be overridden)
List	.MAP (can be overridden)

VM.TMP FILE

MS-LINK uses available memory for the link session. If the files to be linked create an output file larger than available memory, MS-LINK creates a temporary file named VM.TMP. When this happens, the following message is displayed:

```
VM.TMP has been created.
Do not change diskette in drive, <drv:>
```

Do not remove the diskette from the default drive until the link session ends. If the diskette is removed, the operation of MS-LINK becomes unpredictable, and this message may appear:

```
Unexpected end of file on VM.TMP
```

MS-LINK uses VM.TMP as a virtual memory. The contents of VM.TMP are subsequently written to the file named in response to the run file: prompt. VM.TMP is deleted at the end of the linking session.

WARNING: Do not assign the name VM.TMP to any file. If you do this, MS-LINK erases the contents of your VM.TMP file if it needs to create its own VM.TMP file. When this happens, the contents of your VM.TMP file are lost.

2

RUNNING MS-LINK

Two types of commands are used when running MS-LINK: a command to invoke MS-LINK and commands given in response to command prompts. In addition, there are six switches that control alternate MS-LINK features. Usually, you'll enter all the commands to MS-LINK at the keyboard; however, answers to the command prompts and any switches can be kept in a response file. Some command characters are provided to help you enter linker commands.

INVOKING MS-LINK 2.1

You can invoke MS-LINK in three ways. If you use the first method, commands are entered as responses to individual prompts. With the second method, you enter commands on the line used to invoke MS-LINK. If you use the third method, all necessary commands are contained in a response file.

METHOD 1: LINK

Enter:

LINK

MS-LINK is loaded into memory and returns a series of four text prompts that appear one at a time. Your answers to the prompts tell MS-LINK to perform specific tasks.

At the end of each line, you can enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, MS-LINK will not perform the function controlled by that switch.

The command prompts and switches are summarized here and described in more detail later in this chapter.

Exhibit 2a: LINK Command Prompts

PROMPT	RESPONSES
Object Modules [.OBJ]:	List .OBJ files to be linked, separated by blank spaces or plus signs. If a plus sign is the last character entered, the prompt reappears. (No default; response required)
Run File [Object-file.EXE]:	List file name for executable object code. (Default: first-Object-filename.EXE)
List File [Run-file.MAP]:	List file name for listing. (Default: RUN filename)
Libraries [ ]:	List file names to be searched, separated by blank spaces or plus signs. If a plus sign is the last character entered, the prompt reappears. (Default: no search)

Exhibit 2b: LINK Switches

SWITCH	ACTION
/DSALLOCATE	Load data at high end of Data Segment. Required for Pascal and FORTRAN programs.
/HIGH	Place run file as high as possible in memory. Do not use with Pascal or FORTRAN programs.
/LINENUMBERS	Include line numbers in list file.
/MAP	List all global symbols and definitions.
/PAUSE	Halt link session and wait for carriage return.
/STACK:<number>	Set fixed stack size in run file.

Command Characters

MS-LINK has three command characters.

+ Separates entries and extends the current physical line after the Object Modules and Libraries prompts. (A space can be used to separate object modules.) To enter a large number of responses, use a plus sign/carriage return sequence at the end of the physical line (to extend the logical line). If the plus sign/carriage return is the last entry after the Object Modules or Libraries prompts, MS-LINK will prompt you for more module names. When either prompt reappears, continue to enter responses. When all the modules to be linked have been listed, make sure that the response line ends with a module name and a carriage return.

Example:

```
Object Modules [.OBJ]: FUN TEXT TABLE CARE +
Object Modules [.OBJ]: FOO + FLIPFLOP + JUNQUE +
Object Modules [.OBJ]: CORSAIR
```

; A semicolon followed by a carriage return selects the default responses to all remaining prompts. This saves time and eliminates having to enter a series of carriage returns.

NOTE: Do not use the semicolon if you want to skip some, but not all, of the remaining prompts. Once the semicolon has been entered, you can no longer respond to any of the prompts for that link session.

Example:

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

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

^C An Alt-C immediately aborts the link session. If you enter a wrong response, such as the wrong file name or an incorrectly spelled file name, press Alt-C to exit MS-LINK. Then, reinvoke MS-LINK and start over.

METHOD 2: LINK <filenames>[/switches]

Enter:

LINK <object-list>,<runfile>,<listfile>,<lib-list>[/switch...]

where:

object-list is a list of object modules, separated by plus signs.

runfile is the name of the file that will receive the executable output.

listfile is the name of the file that will receive the listing.

lib-list is a list of library modules to be searched.

/switch are optional switches, which can be placed after any of the response entries, before a comma or after <lib-list>. The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas. To select the default for a field, simply enter a second comma without spaces in between (see the example).

Example:

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

This loads MS-LINK and then causes object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ to be loaded. MS-LINK then pauses (this is caused by the /P switch). When you press any key, MS-LINK links the object modules, produces a global symbol map (the /M switch), and defaults to FUN.EXE run file. Then, MS-LINK creates a list file named FUNLIST.MAP, and searches the library file COBLIB.LIB.

METHOD 3: LINK @<filespec>

Enter:

LINK @<filespec>

where:

filespec is the name of a response file.

A response file contains answers to the MS-LINK prompts and may also contain any of the switches. Method 3 lets you conduct the MS-LINK session without interactive (direct) user responses to the MS-LINK prompts.

A response file contains text lines, one for each prompt. Responses must appear in the same order as the command prompts appear. Switches and command characters in the response file are used in the same way as when you respond to MS-LINK prompts.

When the MS-LINK session begins, each prompt will be displayed along with the response you put into the response file. If the response file does not contain answers for all the prompts, MS-LINK will display any prompt that is without a response and wait for you to enter a legal response. When you enter a legal response, the link session continues.

Example:

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

This response file causes MS-LINK to load the four object module. MS-LINK will pause before creating a public symbol map that allows you to swap diskettes. (Be sure you understand how to use the /PAUSE switch before using this feature.) When any key is pressed, the output files will be named FUNLIST.EXE and FUNLIST.MAP, MS-LINK will search the library file COBLIB.LIB, and will use the default settings for the flags.

2.2 COMMAND PROMPTS

You command MS-LINK by entering responses to four text prompts. After you enter a response to the current prompt, the next prompt appears. When the last prompt has been answered, MS-LINK starts linking without further command.

When the link session is finished, MS-LINK exits to the operating system. If MS-LINK has finished successfully, the operating system prompt is displayed. If the link session is unsuccessful, MS-LINK returns the appropriate error message.

MS-LINK prompts the user for the names of object, run, and list files, and for libraries. In the following sections, the prompts are listed in their order of appearance. If a prompt has a default value, that value is shown in square brackets ([ ]) following the prompt.

OBJECT MODULES [.OBJ]:

Enter a list of the object modules to be linked. MS-LINK assumes by default that the file extension is .OBJ. If an object module has any other file extension, that extension must be given here.

Modules must be separated by plus signs (+).

Remember that MS-LINK loads segments into classes in the order that they are encountered.

RUN FILE [First-Object-filename.EXE]:

After you enter a file name, MS-LINK uses that file to store the run (executable) file that results from the link session. All run files receive the file extension .EXE, even if you specify another extension (the user-specified extension is ignored).

If no response is entered to the run file prompt, MS-LINK will use the first file name entered in response to the Object Modules prompt as the RUN file name.

Example:

```
Run File [FUN.EXE]:      B:PAYROLL/P
```


This tells MS-LINK to create the run file PAYROLL.EXE on drive B:. It also tells MS-LINK to pause so you can insert a new diskette to receive the run file.

LIST FILE [Run-Filename.MAP]:

The list file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the run file.

The default response is the run file name with the default file extension .MAP.

LIBRARIES [ ]:

You can respond with up to eight library file names or just a carriage return (indicating that there will be no library search). Library files must have been created by a library utility. MS-LINK assumes a default extension of .LIB for library files.

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

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

If MS-LINK cannot find a library file on the drive specified, it returns the message:

```
Cannot find library <library-name>
Enter new drive letter:
```

Simply press the letter for the drive designation (for example, B).

MS-LINK does not search within each library file sequentially. Instead, it uses a method called dictionary-indexed library search. This means that MS-LINK finds definitions for external references by index access rather than by searching the entire file for each reference. This indexed search reduces substantially the link time for sessions involving library searches.

2.3 SWITCHES

Six switches control alternate linker functions. These switches must be entered at the end of a prompt response, regardless of the method used to invoke MS-LINK. Switches can be grouped at the end of a single response, or they can be entered at the ends of several. If more than one switch is placed at the end of a response, each switch must be preceded by a slash (/).

All switches may be abbreviated; those abbreviations can consist of anything from a single letter to the complete switch name. The only restriction is that an abbreviation must be a sequential sub-string; no gaps or transpositions are allowed. For example, here are some legal and illegal abbreviations of the /DSALLOCATE switch:

LEGAL	ILLEGAL
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE

/DSALLOCATE tells MS-LINK to load all data (DGroup) at the high end of the Data Segment. At run time, the DS pointer is set to the lowest possible address, allowing the entire DS segment to be used.

If you use /DSALLOCATE in combination with the default load low (that is, the /HIGH switch is not used), application programs can allocate dynamically any available memory below the area specifically allocated within DGroup. The data will remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

NOTE: Your application program can dynamically allocate up to 64K bytes (or the actual amount available less the amount allocated within DGroup).

/HIGH

/HIGH tells MS-LINK to place the run image as high as possible in memory. Otherwise, MS-LINK places the run file as low as possible.

IMPORTANT: Do not use /HIGH with Pascal or FORTRAN programs.

/LINENUMBERS

The /LINENUMBERS switch tells MS-LINK to include in the list file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the list file.

NOTE: Not all compilers produce object modules that contain line number information. In these cases, MS-LINK cannot include line numbers.

/MAP

/MAP tells MS-LINK to list all public (global) symbols defined in the input modules. If /MAP is set, MS-LINK lists only errors (which includes undefined globals).

The symbols are listed alphabetically. For each symbol, MS-LINK gives its value and its segment:offset location in the run file. The symbols are listed at the end of the list file.

/PAUSE

Normally, MS-LINK performs a linking session from beginning to end without stopping. /PAUSE causes MS-LINK to pause in the link session at the point where the switch is encountered. This allows you to change diskettes before MS-LINK outputs the run (.EXE) file.

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

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

MS-LINK resumes processing when you press any key.

CAUTION: Do not swap the diskette which will receive the list file, or the diskette used for the VM.TMP file, if created.

/STACK: <number>

<number> represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If the /STACK switch is not used in a link session, MS-LINK calculates the necessary stack size automatically. If a value from 1 to 511 is entered, MS-LINK uses 512.

All compilers and assemblers should provide information in the object modules that allows the linker to compute the required stack size. At least one object (input) module must contain a stack allocation statement. If not, MS-LINK returns a "WARNING: NO STACK STATEMENT" error message.

3

ERROR MESSAGES

All errors cause the link session to abort. After the cause is found and corrected, MS-LINK must be rerun.

ATTEMPT TO ACCESS DATA OUTSIDE OF SEGMENT BOUNDS, POSSIBLY BAD OBJECT MODULE

Probably a bad object file.

BAD NUMERIC PARAMETER

Numeric value not in digits.

CANNOT OPEN TEMPORARY FILE

MS-LINK is unable to create the file VM.TMP because the disk directory is full. This can be remedied by inserting a new diskette. Do not change the diskette that will receive the list.MAP file.

ERROR: DUP RECORD TOO COMPLEX

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

ERROR: FIXUP OFFSET EXCEEDS FIELD WIDTH

An assembly language instruction refers to an address with a short instruction instead of a long one. Edit the assembly language source and reassemble.

INPUT FILE READ ERROR

Probably a bad object file.

INVALID OBJECT MODULE

Object module(s) incorrectly formed or incomplete (as when assembly was stopped in the middle).

SYMBOL DEFINED MORE THAN ONCE

MS-LINK found two or more modules that define a single symbol name.

PROGRAM SIZE OR NUMBER OF SEGMENTS EXCEEDS CAPACITY OF LINKER

The total size cannot exceed 384K bytes; the number of segments cannot exceed 255.

REQUESTED STACK SIZE EXCEEDS 64K

Use the /STACK switch to specify a size smaller than 64K bytes.

SEGMENT SIZE EXCEEDS 64K

64K bytes is the addressing system limit.

SYMBOL TABLE CAPACITY EXCEEDED

Many long names have been entered,exceeding approximately 25K bytes.

TOO MANY EXTERNAL SYMBOLS IN ONE MODULE

The limit is 256 external symbols per module.

TOO MANY GROUPS

The limit is ten groups.

TOO MANY LIBRARIES SPECIFIED

The limit is eight libraries.

TOO MANY PUBLIC SYMBOLS

The limit is 1024.

TOO MANY SEGMENTS OR CLASSES

The limit is 256 (segments and classes taken together).

UNRESOLVED EXTERNALS: <list>

The external symbols listed have no defining module among the modules or library files specified.

VM READ ERROR

A disk problem; not caused by MS-LINK.

WARNING: NO STACK SEGMENT

Appears after entering the /STACK switch. None of the object modules specified contains a statement allocating stack space.

WARNING: SEGMENT OF ABSOLUTE OR UNKNOWN TYPE

A bad object module or an attempt to link modules MS-LINK cannot handle (e.g., an absolute object module).

WRITE ERROR IN TMP FILE

No disk space available for VM.TMP file expansion.

WRITE ERROR ON RUN FILE

Usually means not enough disk space for run file.

MS-CREF

First VICTOR printing February, 1983.

ISBN 0-88182-014-8 Printed in U.S.A.

CONTENTS

1. Introduction 1-1
1.1 Features and Benefits of MS-CREF 1-1
1.2 Overview of MS-CREF Operation 1-1

2. Running MS-CREF 2-1
2.1 Creating a Cross-Reference File 2-1
2.2 Invoking MS-CREF 2-2
Method 1: CREF 2-2
Method 2: CREF <crffile>,<listing> 2-3
2.3 Format of Cross-Reference Listings 2-4

3. Error Messages 3-1

4. Format of MS-CREF Compatible Files 4-1
4.1 MS-CREF File Processing 4-1
4.2 Format of Source Files 4-1
First Three Bytes 4-2
Control Symbols 4-2

EXHIBITS

1a: Overview of MS-CREF Operation 1-2
4a: Records That Begin with a Control Symbol 4-2
4b: Records That End with a Control Symbol 4-3

CHAPTERS

1. Introduction 1

2. Running MS-CREF 2

3. Error Messages 3

4. Format of MS-CREF Compatible Files 4

1

INTRODUCTION

FEATURES AND BENEFITS OF MS-CREF 1.1

The MS-CREF Cross Reference Facility helps you debug assembly language programs. MS-CREF produces an alphabetical listing of all the symbols in a special file produced by your assembler. With this listing, you can quickly locate the line number of each occurrence of any symbol in your source program.

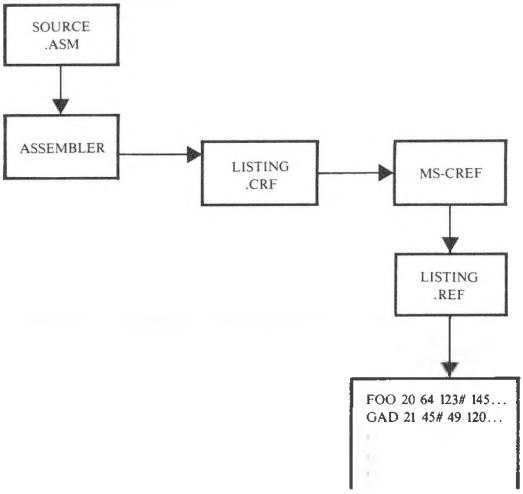
The MS-CREF produced listing is meant to be used with the symbol table produced by your assembler. The symbol table lists the value of each symbol and its type and length. This information is needed to correct wrong symbol definitions or uses.

OVERVIEW OF MS-CREF OPERATION 1.2

MS-CREF produces a file that lists each symbol used in your program along with the line numbers where it appears, and the line number where it is defined. To create this listing, you must first use the assembler to make a cross-reference file. Then, MS-CREF takes this cross-reference file and turns it into an alphabetical listing of the symbols in the file.

Beside each symbol in the listing, MS-CREF lists (in ascending sequence) the line numbers in the source program where the symbol occurs. The line number where the symbol is defined is indicated by a pound sign (#).

Exhibit 1a: Overview of MS-CREF Operation



2

RUNNING MS-CREF

Two types of commands are used when running MS-CREF: a command that invokes MS-CREF and commands issued as you respond to command prompts. All commands are entered at the keyboard. Command characters exist to help you enter MS-CREF commands.

CREATING A CROSS-REFERENCE FILE 2.1

Before you can use MS-CREF to create a cross-reference listing, you must first create a cross-reference file with the MACRO-86 Macroassembler. To create a cross-reference file, answer the fourth assembler command prompt with the name of the file you want to receive the cross-reference file.

The fourth assembler prompt is:

```
Cross-reference [NUL.CRF]:
```

If you don't enter a file name, the assembler will not create a cross-reference file. When you enter the file name, you can also specify the drive or device that you want to receive the file, and what extension you want the file to have, if other than .CRF. If you change the extension from .CRF to something else, remember to specify the file extension when naming the file at the first MS-CREF prompt.

After you have responded to the fourth assembler prompt, the cross-reference file will be generated during the assembly session. Then, you can convert the cross-reference file produced by the assembler into a cross-reference listing using MS-CREF.

2.2 INVOKING MS-CREF

MS-CREF is invoked in two ways: (1) by entering commands as answers to individual prompts; and (2) by entering all commands on the line used to invoke MS-CREF.

METHOD 1: CREF

Enter:

CREF

MS-CREF is loaded into memory and then returns two text prompts, one at a time. Your answers to these prompts tell MS-CREF to convert a cross-reference file into a cross-reference listing.

Command Prompts

`Cross-reference [.CRF]:`

Enter the name of the cross-reference file that you want MS-CREF to convert into a cross-reference listing. The file name should be the name you gave your assembler when you commanded it to produce the cross-reference file.

If you don't specify a file extension when you enter the cross-reference file name, MS-CREF will look for a file with the name you specified and the extension .CRF. If your cross-reference file has a different extension, make sure to specify it when entering the file name.

Chapter 3 describes what MS-CREF expects to see in the cross-reference file. You will need this information if your cross-reference file was not produced by a Microsoft assembler.

`Listing [crffile.REF]:`

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

If you want your cross-reference listing to have the same file name as the cross-reference file (except that it will have the extension .REF), press the carriage return key when the Listing prompt appears. If you want to give it another name, or a different extension, you must enter a response to the Listing prompt.

If you want the listing file placed anywhere other than the default drive, specify that drive or device when responding to the Listing prompt.

Special Command Characters

; A single semicolon (;) followed by a carriage return selects the default response to the listing prompt. This feature saves time and makes it unnecessary to answer the Listing prompt.

If you use the semicolon, MS-CREF gives the listing file the same name as the cross-reference file and the default file extension .REF. For example:

Cross reference [.CRF]: FUN;

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

^C Alt-C will abort the MS-CREF session. If you make a wrong response or enter an incorrectly spelled file name, press Alt-C to exit MS-CREF. Then, reinvoke MS-CREF and start over.

METHOD 2: CREF <crffile>,<listing>

Enter:

CREF <crffile>,<listing>

where:

<crffile> is the name of a cross-reference file produced by your assembler.

<listing> is the name of the file you want to receive the cross-reference listing of symbols in your program.

MS-CREF is loaded into memory and starts converting 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.

MS-CREF assumes that the file name extension is .CRF; you can override this by specifying a different extension. To select the default file name and extension for the listing file, enter a semicolon after you enter the <crffile> name. If the file named for the <crffile> does not exist, MS-CREF displays the message:


```
Fatal I/O Error 110
in File: <crffile>.CRF
```

Control will return to your operating system. For example:

CREF FUN; ↵

causes MS-CREF to process the cross-reference file FUN.CRF and to produce a listing file named FUN.REF.

To give the listing file a different name, extension, or destination, simply specify these differences when entering the command line.

CREF FUN,B:WORK.ARG

causes MS-CREF to process the cross-reference file RUN.CRF and to produce a listing file named WORK.ARG which will be placed on drive B:.

2.3 FORMAT OF CROSS-REFERENCE LISTINGS

The cross-reference listing is an alphabetical list of all the symbols in your program. Each page is headed with the title of the program or program module, followed by the list of symbols. After each symbol name is a list of the line numbers where the symbol occurs in your program. The number of the line where the symbol is defined is followed by a pound sign (#).

Here is a sample cross-reference listing.

```
MS-CREF     (vers no.)          (date)
                                comes from
ENTX PASCAL entry           < TITLE directive

Symbol Cross-Reference      (# is definition)              Cref-1

AAAXQQ  .......              37#  38

BEGHQQ  .......              83   84# 154   76
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

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

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

PNUXQQ      .......     69    70   150

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

ENTX        PASCAL entry for initializing programs

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

3

ERROR MESSAGES

All errors cause MS-CREF to abort. Control will be returned to your operating system. All error messages have this format:

```
Fatal I/O Error <error number>
in File: <filename>
```


where:

<filename> is the name of the file where the error occurs.

<error number> is one of the numbers in the following list of errors.

NUMBER	ERROR
101	Hard data error Unrecoverable disk I/O error
102	Device name error Illegal device specification (for example, X:FOO.CRF)
103	Internal error Report to Victor Technologies, Inc.
104	Internal error Report to Victor Technologies, Inc.
105	Device offline Disk drive door open, no printer attached, etc.
106	Internal error Report to Victor Technologies, Inc.
108	Disk full
110	File not found
111	Disk is write-protected

NUMBER	ERROR
112	Internal error Report to Victor Technologies, Inc.
113	Internal error Report to Victor Technologies, Inc.
114	Internal error Report to Victor Technologies, Inc.
115	Internal error Report to Victor Technologies, Inc.

4

FORMAT OF MS-CREF COMPATIBLE FILES

MS-CREF will process files other than those generated by the MACRO-86 Macroassembler as long as the files conform to the format that MS-CREF expects.

MS-CREF FILE PROCESSING 4.1

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 using the number of end-of-line characters it encounters. Every line in the source file must contain at least one end-of-line character.

MS-CREF attempts to place a heading at the top of every page of the listing. The name used as a title is the text passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol. If there is more than one title symbol in the source file, MS-CREF uses the last title read for all page headings. If MS-CREF does not encounter a title symbol in the file, the title line is left blank.

FORMAT OF SOURCE FILES 4.2

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 takes arguments for page length and line length. It passes this information to the cross-reference file.

- First byte: The number of lines to be printed on a page. Page length can range from 1 to 255 lines.
- Second byte: The number of characters per line. Line length can range from 1 to 132 characters.
- Third byte: The page symbol (07) that tells MS-CREF that the two preceding bytes define the listing page size.

If MS-CREF does not see these first three bytes in a file, it uses default values for page size (page length: 58 lines; line length: 80 characters).

CONTROL SYMBOLS

Exhibits 4a and 4b 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) as the first or last byte of the record.

Exhibit 4a: Records That Begin with a Control Symbol

BYTE VALUE	CONTROL SYMBOL	SUBSEQUENT BYTES
01	Reference symbol	Record is a reference to a symbol name (1 to 80 characters).
02	Define symbol	Record is a definition of a symbol name (1 to 80 characters).
04	End of line	(none)
05	End of file	1AH

Exhibit 4b: Records That End with a Control Symbol

BYTE VALUE	CONTROL SYMBOL	SUBSEQUENT BYTES
06	Title defined	Record is title text (1 to 80 characters).
07	Page length/ line length	One byte for page length followed by one byte for line length.

For all record types, the byte value represents an alternate (Alt) character, as follows:

01	Alt-A
02	Alt-B
04	Alt-D
05	Alt-E
06	Alt-F
07	Alt-G

The control symbols are defined as follows:

- Reference symbol: Record contains the name of a symbol that is referenced. The name can 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 can 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 Alt-D).
- End of file: Record is the end-of-file character (1AH).
- Title defined: ASCII characters of the title to be printed at the top of each listing page. The title can be from 1 to 80 characters long. Additional characters are truncated.

  The last title definition record encountered is used for the title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the listing is left blank.

- Page length/line length: The first byte of the record contains the number of lines to be printed per page (range is from 1 to 255 lines). The second byte contains the number of characters to be printed per page (range is from 1 to 132 characters). The default page length is 58 lines. The default line length is 80 characters.

Summary of CRF file record contents:

BYTE CONTENTS	LENGTH OF RECORD
|01|symbol_name|	2-81 bytes
|02|symbol_name|	2-81 bytes
|04|	1 byte
|05|1A|	2 bytes
|title_text|06|	2-81 bytes
|PL|LL|07|	3 bytes

MACRO-86

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing February, 1983.

ISBN 0-88182-016-4 Printed in U.S.A.

CONTENTS

1. Introduction 1-1
   1.1 General Description 1-1
   1.2 Overview of MACRO-86 Operation 1-5

2. Creating a MACRO-86 Source File 2-1
   2.1 General Facts About Source Files 2-1
      Naming Your Source File 2-1
      Legal Characters 2-2
      Numeric Notation 2-2
      Source File Contents 2-3
   2.2 Statement Line Format 2-4
      Names 2-4
      Comments 2-5
   2.3 The Action Field 2-6
   2.4 The Expression Field 2-7

3. Names: Labels, Variables, and Symbols 3-1
   3.1 Labels 3-1
      Segment 3-3
      Offset 3-3
      Type 3-3
   3.2 Variables 3-4
   3.3 Symbols 3-6

4. Expressions: Operands and Operators 4-1
   4.1 Memory Organization 4-1
      Segments and Groups 4-1
      Segment and Group References 4-2
      Reference Definition During Assembly 4-5

4.2 Operands 4-7
Immediate Operands 4-8
Register Operands 4-10
Memory Operands 4-12
4.3 Operators 4-15
Attribute Operators 4-15
Arithmetic Operators 4-26
Relational Operators 4-28
Logical Operators 4-28
Expression Evaluation: Precedence of Operators 4-29

5. Action: Instructions and Directives 5-1
5.1 Instructions 5-1
5.2 Directives 5-2
Memory Directives 5-3
Conditional Directives 5-27
Macro Directives 5-30
Listing Directives 5-43

6. Assembling a MACRO-86 Source File 6-1
6.1 Invoking MACRO-86 6-1
Method 1: MASM 6-1
Method 2: <filenames> [/switches] 6-3
6.2 MACRO-86 Command Prompts 6-4
Command Prompt Descriptions 6-5
6.3 MACRO-86 Command Switches 6-6
6.4 Formats of Listings and Symbol Tables 6-9
Program Listing 6-9
Symbol Table Format 6-14

7. MACRO-86 Messages 7-1
7.1 Operating Messages 7-1
7.2 Error Messages 7-2
Assembler Errors 7-2
I/O Handler Errors 7-10
7.3 Numerical List of Error Messages 7-12

EXHIBITS

1a: MACRO-86 Features 1-2
1b: Macro Call Statement 1-3
1c: Conditional Assembly Facility 1-4
1d: Overview of MACRO-86 Operation 1-6
1e: Source File Assembly 1-7
1f: Cross-Reference File 1-8
2a: Special Notation and Numeric Values 2-3
2b: Operators and Operands Legal in Expression Field 2-8
3a: Define Directive and Variable Type Sizes 3-5
4a: Diagram of MACRO-86 Program Statements 4-4
4b: Contents of Operand Types 4-8
4c: Format of Data Types Contained in Operands 4-9
4d: Register/Memory Field Encoding 4-11
4e: Use of Structure Operand in a Stack Operation 4-14
4f: MACRO-86 Attribute Operators 4-15
6a: MACRO-86 Command Prompts 6-2
6b: MACRO-86 Command Switches 6-2
6c: Combining Conditional Listing Directives with the /X Switch 6-8

CHAPTERS

1. Introduction 1

2. Creating a MACRO-86 Source File 2

3. Names: Labels, Variables, and Symbols 3

4. Expressions: Operands and Operators 4

5. Action: Instructions and Directives 5

6. Assembling a MACRO-86 Source File 6

7. MACRO-86 Messages 7

1

INTRODUCTION

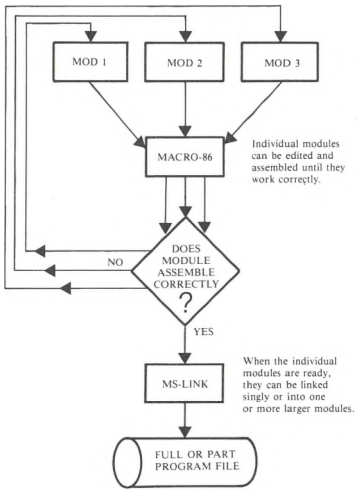
GENERAL DESCRIPTION 1.1

The MACRO-86 Macro Assembler is a very powerful assembler for 8086-based computers. MACRO-86 has many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives give you all the tools you need to get full use and power from an 8086 or 8088 microprocessor. Even though MACRO-86 is more complex than other microcomputer assemblers, it is still easy to use.

MACRO-86 produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. Then, the assembled code can be linked (using the MS-LINK linker utility) to produce relocatable, executable object code. Relocatable code can be loaded anywhere in memory. So, MACRO-86 can execute where it is most efficient, not just in a fixed range of memory addresses.

Relocatable code lets you create programs in modules that can be assembled, tested, and perfected individually. This saves recoding time because testing and assembly is done on smaller pieces of program code. All modules can be error-free before being linked together into larger modules or into the entire program.

Exhibit 1a: MACRO-86 Features



The macro facility lets you write "macros" — blocks of code that represent sets of instructions you use often. So you don't have to recode these instructions each time you use them.

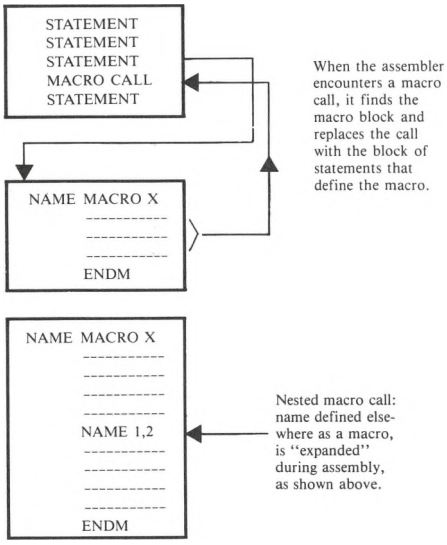
Macro definitions are the instructions that a macro contains. Each time you need the instructions, you place a call to the macro in the source file (instead of recoding the set of instructions). MACRO-86 expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of source modules because the macro definitions are given only once; each subsequent call takes only one line.

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

The macro facility includes repeat, indefinite-repeat, and indefinite-repeat-character directives to help program repeat-block operations. You can also use the Macro directive to change the action of any instruction or directive; just use the instruction or directive name as the macro name. When you put an instruction or directive statement into your program, MACRO-86 first checks

the symbol table it created to see if the instruction or directive is a macro name. If it is, MACRO-86 replaces the macro call statement with the body of instructions in the macro definition. If the name is not defined as a macro, MACRO-86 tries to match the name with an instruction or directive. The Macro directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.

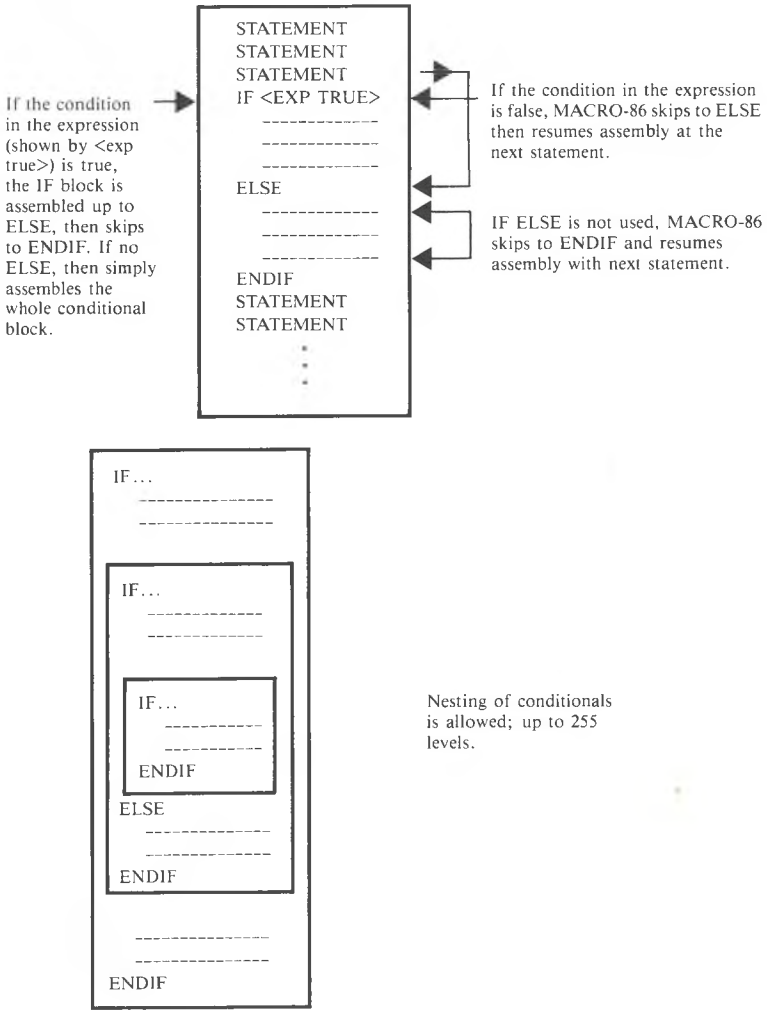
Exhibit 1b: Macro Call Statement



MACRO-86 supports an expanded set of conditional directives. Directives that evaluate a variety of assembly conditions can test assembly results and branch when required. Unneeded or unwanted code portions are left unassembled. MACRO-86 tests for blank or nonblank, for defined or not-defined symbols, for equivalence, and for first or second assembly pass. MACRO-86 also compares strings for identity or difference. Conditional directives simplify the evaluation of assembly results, and make it easier to program condition-testing code (as well as making that code more powerful).

The MACRO-86 conditional assembly facility also supports conditionals inside conditionals ("nesting"). Conditional assembly blocks can be nested up to 255 levels.

Exhibit 1c: Conditional Assembly Facility



MACRO-86 is upward-compatible with MACRO-80 and with Intel's ASM86, except for Intel codemacros and macros.

Some 8086 instructions use only one operand type. If you give a typeless operand for an instruction that accepts only one type of operand (e.g., in PUSH [BX], [BX] has no size, but PUSH only takes a word), MACRO-86 does not return an error. When the wrong type-choice is made, MACRO-86 returns an error message and also tells you the "correct" code. For example, if you enter:


```
MOV     AL,WORDLBL
```

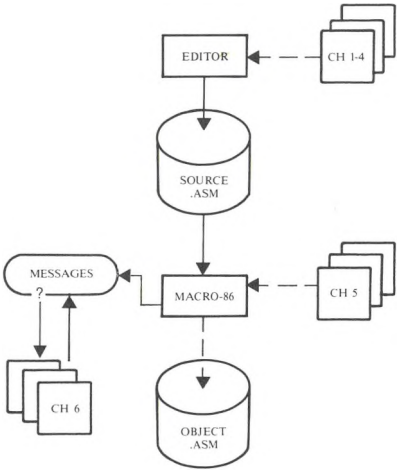
you may mean (1) MOV AX,WORDLBL, (2) MOV AL,BYTE PTR WORDLBL, or (3) MOV AL,<other>. MACRO-86 generates the second instruction because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size." MACRO-86 accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This modification eliminates a lot of debugging work. MACRO-86 still returns an error message, however, because you may have misstated the operand the MACRO-86 assumes is "correct."

OVERVIEW OF MACRO-86 OPERATION 1.2

The first task is creating a source file. Use PMATE or any other 8086 editor to create the MACRO-86 source file. MACRO-86 assumes a default file extension of .ASM for the source file. Creating the source file involves creating instruction and directive statements that follow the rules and constraints described in Chapters 2-5 in this manual.

When the source file is ready, run MACRO-86 as described in Chapter 6. Refer to Chapter 7 for explanations of messages displayed during or immediately after assembly.

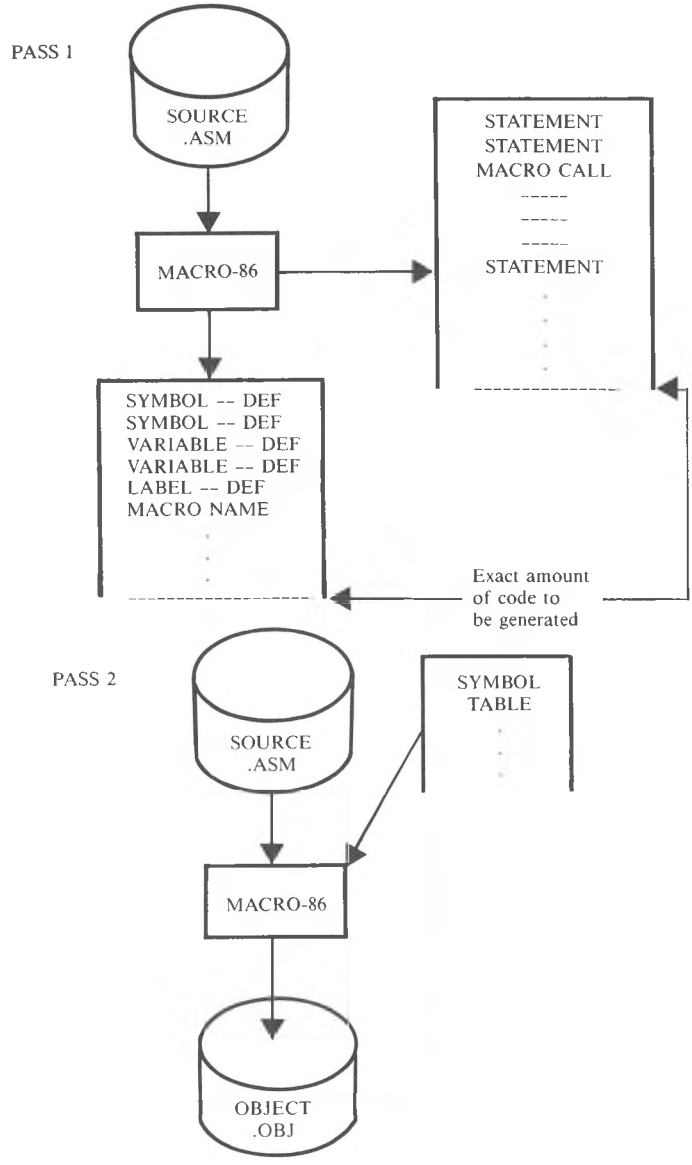
Exhibit 1d: Overview of MACRO-86 Operation



MACRO-86 is a two-pass assembler. The source file is assembled twice; slightly different actions occur during each pass. On the first pass, MACRO-86 evaluates the statements and expands macro call statements, calculates the amount of code it must generate, and builds a symbol table where all symbols, variables, labels, and macros are assigned values. During the second pass, MACRO-86 uses the symbol table to fill in the symbol, variable, label, and expression values. It also expands macro call statements and puts the relocatable object code into a file with the default file extension .OBJ. The .OBJ file can be processed with MS-LINK. The .OBJ file can be stored as part of your library of object programs for later linking with one or more .OBJ modules. .OBJ modules can also be processed with the MS-LIB library manager.

The source file can also be assembled without creating an .OBJ file. All the other assembly steps are performed, but the object code is not sent to disk. Only erroneous source statements are displayed on the terminal screen. This method helps you check the source code for errors. It is faster than creating an .OBJ file because no file-creating or writing is performed. Modules are test-assembled quickly and errors are corrected before the object code is put on disk. Modules that assemble with errors do not clutter the diskette.

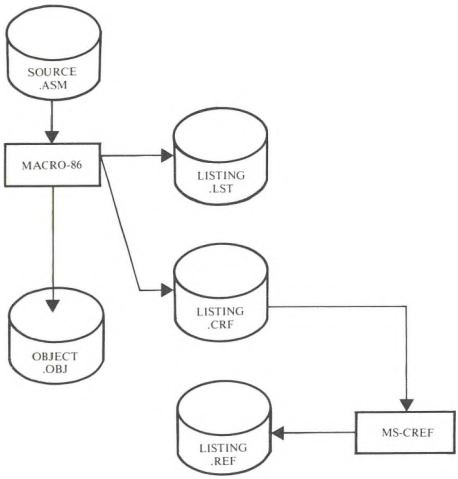
Exhibit 1e: Source File Assembly



On command, MACRO-86 creates a listing file and a cross-reference file. The listing file contains the initial relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal), and the statement itself. The listing also contains a symbol table showing the values of all symbols, labels, and variables, plus the names of all macros. The listing file has the default file extension .LST.

The cross-reference file is a compact representation of variables, labels, and symbols. It has the default file extension .CRF. When a cross-reference file is processed by MS-CREF, the file is converted into an expanded symbol table that lists all variables, labels, and symbols in alphabetical order. The table is followed by the line number of the source program where each symbol is defined, and by the line numbers where each symbol is used. The final cross-reference listing has the file extension .REF. (See the MS-CREF Section of the *Programmer's Tool Kit, Volume II* for further explanation and instructions.)

Exhibit 1f: Cross-Reference File



2

CREATING A MACRO-86 SOURCE FILE

To create a source file for MACRO-86, you first need to use an editor program (such as PMATE) to create a program file as you would for any assembly or high-level programming language. The information and descriptions in this and the next three chapters will help you create this file.

This chapter discusses statement format and introduces its components. Chapter 2 describes names: variables, labels, and symbols. Chapter 3 describes expressions and their components, operands and operators. Chapter 4 describes the assembler directives.

GENERAL FACTS ABOUT SOURCE FILES 2.1

NAMING YOUR SOURCE FILE

You need to give a name to any source file you create. You can use any name that is legal for your operating system. MACRO-86 expects, however, a specific three-character file extension: .ASM. If you want an extension other than .ASM, you must specify that extension when you begin running the assembler. If you don't specify, MACRO-86 assumes that your file has an .ASM extension. MACRO-86 will either find and assemble the wrong file, or display an error message telling you that the requested file can't be found.

MACRO-86 gives the default extension .OBJ to any object file it outputs. Consequently, you should never give this extension to your source file because it would be destroyed. For similar reasons, you should also avoid the extensions .EXE, .LST, .CRF, and .REF.

LEGAL CHARACTERS

You can use any of these characters in your symbol names:

A-Z **0-9** **?** **@** **_** **$**

You cannot use a numeral as the first character of a name.

MACRO-86 also uses these special characters as operators or delimiters:

Colon (:)	Segment override operator.
Period (.)	Operator for field name of a record or structure. A period can be used in a file name only if it is the first character.
Square brackets ([ ])	Placed around register names to indicate the address value in register, as opposed to the value of the data in the register.
Parentheses ( )	Used as operator in DUP expressions and to change precedence of operator evaluation.
Angle brackets (<>)	Operators placed around initialization values for records or structure or around parameters in IRP macro blocks. Also used to indicate literals.

Square brackets and angle brackets are also used for syntax notation in assembler directives.

NUMERIC NOTATION

Any numeric value has a decimal input radix. In listings, the output radix is hexadecimal for code and data items, and decimal for line numbers. You can change the output radix to octal radix by using the /O switch when you run MACRO-86 (see Section 6.3, "Command Switches"). The input radix is changed by using the .RADIX directive, or by appending special notation to a numeric value (see Exhibit 2a).

Exhibit 2a: Special Notation and Numeric Values

RADIX	RANGE	NOTATION	EXAMPLE
Binary	0-1	B	01110100B
Octal	0-7	Q or O	735Q, 621O
Decimal	0-9	None or D	9384 (default) or 8149D*
Hexadecimal	0-9, A-F	H	0FFH, 80H**

* When .RADIX directive changes default radix to not-decimal.
** First character must be a numeral in range 0-9.

SOURCE FILE CONTENTS

A MACRO-86 source file contains instruction statements and directive statements. Instruction statements consist of 8086 instruction mnemonics and operands; they tell the 8086 processor to perform specific tasks. Directive statements tell MACRO-86 to prepare data for use in and by instructions.

Statements are usually put into blocks of code assigned to a specific segment (code, data, stack, extra). The segments can appear in any order in the source file. Within the segments, statements can appear in any order that creates a valid program. Some exceptions to random ordering do exist; these are discussed under the affected assembler directives.

Each segment must end with an end-segment statement (ENDS); each procedure must end with an end-procedure statement (ENDP); and each structure must end with an end-structure statement (ENDS). The source file must end with an END statement, telling MACRO-86 where program execution should start.

2.2 STATEMENT LINE FORMAT

Statements in source files follow a strict format (although some variations are allowed).

Directive statements consist of four fields: name, action, expression, comment. For example:


```
FOO     DB      0D5EH       ;create variable FOO
                            ;containing the value 0D5EH
Name    Action  Expression  ;Comment
```

Instruction statements usually consist of three fields: action, expression, comment. For example:

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

Under some conditions, an instruction statement can also have a name field.

NAMES

When present, a name field is the first entry on the statement line. Names can begin in any column, although they are usually started in column 1. Names can be any length you choose. MACRO-86 recognizes, however, only the first 31 characters when your source file is assembled.

When placed in a statement line, you can use names for three purposes: (1) to represent code, (2) to represent data, and (3) to represent constants.

Any of these formats can make a name represent code:

- NAME: (By itself or followed by a directive or 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)
- EXTRN NAME:FAR (For use outside its own module and segment)

For a name to represent data:

- NAME LABEL <type>
- NAME Dx <exp>
- EXTRN NAME:<type>

For a name to represent a constant:

- NAME EQU <constant>
- NAME = <constant>
- NAME SEGMENT <attributes>
- NAME GROUP <segment-names>

COMMENTS

Comments explain the processing necessary at any point in a program. These comments are useful for debugging, for altering code, or for updating code. You don't need to include comments for your assembly language program to operate successfully; however, we strongly recommend that you use them. You should consider putting comments at the beginning of each segment, procedure, structure, and module; and after each code line that begins a step in the processing.

If you use comments in your program, each one must be preceded by a semicolon. If your comment runs onto a second or third line, each of those lines must also be preceded by a semicolon. If you want to place a very long comment in your program, the COMMENT directive frees you from entering a semicolon on every line (see COMMENT in Section 5.2).

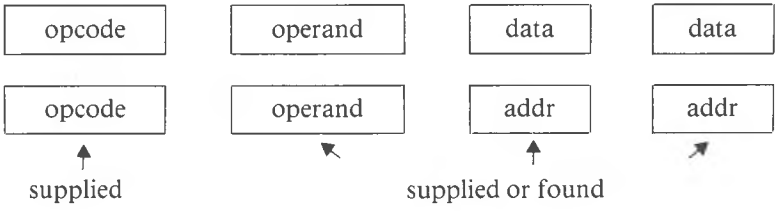
Comments are ignored by MACRO-86. They do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code.

2.3 THE ACTION FIELD

The action field contains either an 8086 instruction mnemonic or a MACRO-86 assembler directive. If the name field is blank, the action field is the first entry in the statement format. In this case, the action field can start anywhere from column 1 to the last column of the maximum line length.

The entry in the action field tells the processor or assembler to perform a specific function. The action field can contain instructions or directives.

Instructions command processor actions. You can build the necessary data or addresses into an instruction or they can be found in the expression part of an instruction. For example:



Supplied: Part of the instruction

Found: Assembler inserts data and/or address from the information provided by expression in instruction statements.

(The 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.

THE EXPRESSION FIELD 2.4

The expression field contains operands and/or combinations of operands and operators. Some instructions use no operands, some use one, and some use two. One-operand instructions must contain either a source operand or a destination operand, depending on the instruction. If you want two operand instructions, the expression field must contain a destination operand and a source operand (in that order) separated by a comma.

If one or both of the operands is omitted, the instruction carries that information in its internal coding.

Source operands can be immediate operands, register operands, memory operands or attribute operands. Destination operands can be register operands and memory operands.

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

A directive operand is a data operand, a code (addressing) operand or a constant, depending on the directive. In many instructions and directives, operands are connected with operators to form complex operands — longer operands that look like mathematical expressions. Complex operands allow you to specify addresses or data derived from several places. For example:

```
MOV     FOO[BX] ,AL
```

is a destination operand that results from adding the address represented by FOO and the address found in register BX. The processor moves the value in register AL to the destination calculated from these two operand elements.

Another example:

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

In this case, the source operand results from adding the value of FOO plus 5 to the value found in the BX register.

MACRO-86 supports the following operands and operators in the expression field. (They are shown in order of precedence.)

Exhibit 2b : Operators and Operands Legal in Expression Field

OPERANDS	OPERATORS
Immediate	LENGTH, SIZE, WIDTH, MASK, FIELD
(incl. symbols)	[ ], ( ), < >
Register	
Memory	segment override(:)
label	
variables	PTR, OFFSET, SEG, TYPE, THIS
simple	
indexed	HIGH, LOW
structures	
Attribute	*, /, MOD, SHL, SHR
override	
PTR	+, −(unary), −(binary)
:(seg)	
SHORT	EQ, NE, LT, LE, GT, GE
HIGH	
LOW	NOT
value returning	
OFFSET	AND
SEG	
THIS	OR, XOR
TYPE	
.TYPE	SHORT,.TYPE
LENGTH	
SIZE	
record specifying	
FIELD	
MASK	
WIDTH	

NOTE: Some operators can be used as operands or as part of an operand expression. Refer to Sections 4.2 and 4.3 for details on operands and operators.

3

NAMES: LABELS, VARIABLES, AND SYMBOLS

Names are symbolic representations of values used for several functions by MACRO-86, whenever naming is allowed or required. The values represented by names can be addresses, data, or constants.

Names can have any length you choose. MACRO-86 truncates, however, names longer than 31 characters when assembling your source file.

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

LABELS 3.1

Labels are targets for JMP, CALL, and LOOP instructions. MACRO-86 assigns an address to each label as it is defined. When you use a label as an operand for JMP, CALL, or LOOP, MACRO-86 substitutes the attributes of that label for the label name, and sends processing to the appropriate place.

Labels are defined in one of four ways:

1. **<name>:**

   A name followed immediately by a colon defines the name as a NEAR label. <name>: can be placed ahead of any instruction and all directives that allow a name field.<name>: can also be placed on a line by itself.

   Examples:

```
CLEAR_SCREEN:     MOV     AL,20H
FOO:    DB      0FH
SUBROUTINE3:
```

2. **<name> LABEL NEAR**
   <name> LABEL FAR

   Use the LABEL directive. Refer to the discussion of LABEL in Section 5.2.

   NEAR and FAR are discussed under the following type attribute.

   Examples:

```
FOO     LABEL     NEAR
GOO     LABEL     FAR
```

3. **<name> PROC NEAR**
   <name> PROC FAR

   Use the PROC directive. Refer to the discussion of PROC in Section 5.2.

   NEAR is optional. It is the default if you enter only <name> PROC. NEAR and FAR are discussed under the following type attribute.

   Examples:

```
REPEAT     PROC     NEAR
CHECKING   PROC
FIND_CHR   PROC     FAR
```

4. **EXTRN <name>:NEAR**
   EXTRN <name>:FAR

   Use the EXTRN directive. Refer to the discussion of EXTRN in Section 5.2.

   NEAR and FAR are discussed under the following type attribute.

   Example:

```
EXTRN   FOO:NEAR
```

A label has four attributes: segment, offset, type, and the CS ASSUME in effect when the label is defined. Segment is the segment where the label is defined. Offset is the distance from the beginning of the segment to the label's location. Type is either NEAR or FAR.

SEGMENT

Labels are defined inside segments. The segment must be assigned to the CS segment register to be addressable. (The segment can be assigned to a group addressable through CS.) 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 a segment to the location where the label is defined. The offset is a 16-bit unsigned number.

TYPE

Labels are one of two types: NEAR or FAR. NEAR labels are used for references from within the segment where the label is defined. NEAR labels can be referenced from more than one module, as long as the references are from a segment with the same name, attributes, and CS ASSUME. FAR labels are used for references from segments with a different CS ASSUME, or when there is more than 64K bytes between the label reference and the label definition.

NEAR and FAR cause MACRO-86 to generate slightly different code. NEAR labels supply their offset attribute only (a 2-byte pointer); FAR labels supply both their segment and offset attributes (a 4-byte pointer).

3.2 VARIABLES

Variables are names used in expressions (as operands for instructions and directives) to represent an address where a specified value can be found. Variables look much like labels and are similar in some ways; however, the differences are important.

Variables are defined in three ways:

1. **<name><define-dir>**
   <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'>
```

HORSE has the same size as the structure CORRAL.

```
GARAGE RECORD       CAR:8 = 'P'
SMALL  GARAGE       10 DUP (<'Z'>)
```

SMALL has the same size as the record GARAGE.

See the Define, STRUC, and RECORD directives in Section 5.2.

2. **<name> LABEL <size>**

Use the LABEL directive with one of the size specifiers.

<size> is one of the following size specifiers:

- BYTE specifies 1 byte.
- WORD specifies 2 bytes.
- DWORD specifies 4 bytes.
- QWORD specifies 8 bytes.
- TBYTE specifies 10 bytes.

Example:

```
CURSOR    LABEL    WORD
```

See LABEL in Section 5.2.

3. **EXTRN <name>:<size>**

   Use EXTRN with one of the size specifiers described above. See EXTRN in Section 5.2.

   Example:

```
EXTRN  FOO:DWORD
```

Like labels, variables have three attributes: segment, offset, and type. Segment and offset are used as they are with labels; type is used differently.

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 and which size specifier was used to define the variable.

Exhibit 3a: Define Directives and Variable Type Sizes

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

3.3 SYMBOLS

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

Symbols are defined three ways:

1. **<name> EQU <expression>**

   Use the EQU directive. See EQU in Section 5.2.

   <expression> is another symbol, an instruction mnemonic, a valid expression, or any other entry (such as text or indexed references).

   Examples:

```
FOO     EQU     7H
ZOO     EQU     FOO
```

2. **<name> = <expression>**

   Use the Equal Sign directive. See Equal Sign in Section 5.2.

   <expression> is any valid expression.

   Examples:

```
GOO     =       0FH
GOO     =       $+2
GOO     =       GOO+FOO
```

3. **EXTRN <name>:ABS**

   Use the EXTRN directive with type ABS. See EXTRN in Section 5.2.

   Example:

```
EXTRN   BAZ:ABS
```

   An EQU or = directive must define BAZ to a valid expression.

4

EXPRESSIONS: OPERANDS OPERATORS

An expression indicates the values on which an instruction or directive performs its functions. Each expression contains at least one operand (a value), but expressions can contain two or more. Multiple operands are joined by operators, resulting in a series of elements that look like a mathematical expression. This chapter describes the types of operands and operators supported by MACRO-86.

MEMORY ORGANIZATION 4.1

SEGMENTS AND GROUPS

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 and ends with an ENDS directive. In an assembled and linked file, a segment is any block of code addressed through the same segment register and less than 64K bytes long.

MACRO-86 does not do any segment operations; these are left to MS-LINK. MACRO-86 does not check whether your references are entered with the correct distance type. Values such as offset are also left for MS-LINK to resolve.

As long as you observe the 64K limit, you can divide a segment among two or more modules. (However, the SEGMENT statements in each module must be identical.) When the modules are linked, the segments become one. Any references to labels, variables, and symbols within each module take on the offset from the beginning of the whole segment, not just from the beginning of their portion of the segment.

You can use the GROUP directive to place several segments into a group. By doing this, you tell MACRO-86 that you want 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 accessible. It does make value relative to a group base.) Grouping lets you refer to data items without worrying about segment overrides or having to frequently change segment registers.

SEGMENT AND GROUP REFERENCES

References within segments or groups are relative to a segment register, and the final offset of a reference is relocatable until linking is completed. Consequently, the OFFSET operator does not return a constant. Instead, OFFSET causes MACRO-86 to generate an immediate instruction; that is, to use the address of the value instead of the value itself.

There are two kinds of references in a program:

1. Code references (JMP, CALL, LOOPxx): These are relative to the address in the CS register. You cannot override this assignment.

2. Data references (all other references): These are usually relative to the DS register, but this assignment can be overridden.

Suppose you give this forward reference in a program statement:

```
MOV AX,<ref>
```

MACRO-86 looks first for the segment of the reference, then scans the segment registers for the SEGMENT of the reference. Lastly, MACRO-86 looks for the GROUP (if any) of the reference.

If you use the OFFSET operator, however, it always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must use the GROUP name and the colon operator, as in this example:

```
MOV AX,OFFSET<group-name>:<ref>
```

If you use the ASSUME directive to set a segment register to a group, then you can also override the restriction on OFFSET by using the register name.

MOV AX,OFFSET DS:<ref>

The result of both of these statements is the same.

Code labels have four attributes:

1. Segment: The segment that 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 used when the label was coded.

When you enter a NEAR JMP or NEAR CALL, you change the offset (IP) in CS. MACRO-86 compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they differ, MACRO-86 returns an error. (In this case, you must use a FAR JMP or CALL.)

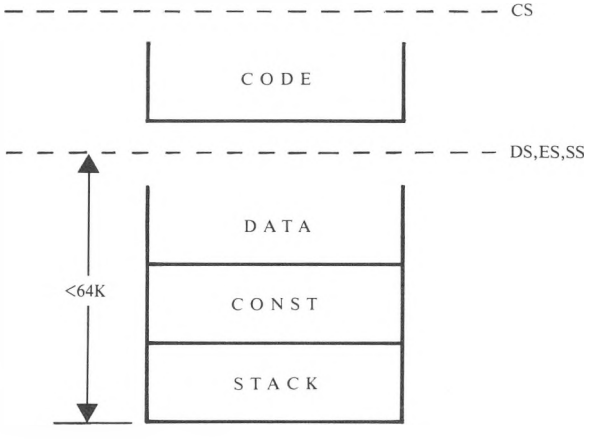
When you enter a FAR JMP or FAR CALL, you change both the offset (IP) in CS and the paragraph number. The paragraph number changes to the CS ASSUME of the target address.

Let's look at a common case: a segment (called CODE) and a group (DGROUP) that contains three segments (DATA, CONST, and STACK). The program statements are:

```
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
                .
                .
                .
```

This arrangement is represented by the following diagram.

Exhibit 4a: Diagram of MACRO-86 Program Statements



Given this arrangement, a statement like:

MOV AX,<variable>

makes MACRO-86 find the best segment register to reach this variable. (The "best" register is the one that requires no segment overrides.)

This statement:

MOV AX,OFFSET<variable>

tells the MACRO-86 to return the offset of the variable relative to the beginning of the segment.

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

MOV AX,OFFSET DGROUP:<variable>

REFERENCE DEFINITION DURING ASSEMBLY

MACRO-86 makes two assembly passes. During the first, it builds a symbol table and calculates how much code is generated; however, MACRO-86 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. Only those errors involving items that must be defined on the first pass are displayed. No listing is produced unless you give a /D switch when you run the assembler. (The /D switch produces a listing for both passes.)

On the second pass, MACRO-86 uses the values defined during the first pass to generate the object code. References defined in the second pass are checked against the pass 1 value in the symbol table. The amounts of code generated during each pass must be the same. If they differ, MACRO-86 returns a phase error.

Because the first pass must keep track of the relative offset, some references must be known. If they are not known, the relative offset is incorrect. These references must be known on the first pass:

- **IF/IFE<expression>**

  If<expression>is not known, MACRO-86 cannot assemble the conditional block (or which part of the block to assemble if ELSE is used). The conditional block will be assembled on the second pass, resulting in a phase error.

- **<expression>DUP(. . .)**

  Since this operand changes the relative offset, the value of the expression must be known on the first pass. The value in parentheses need not be known because it doesn't affect the number of bytes generated.

- **.RADIX<expression>**

  Since this directive changes the input radix, constants could have a different value. This can cause MACRO-86 to evaluate IF or DUP statements incorrectly.

Assembler Operators

MACRO-86 has to solve a major problem during its two passes: how to know the kind of references it's working with even though it has not seen their definitions. Unless the statement containing the forward reference tells the size, distance, or other attribute of the reference, MACRO-86 must take the safe route and generate the largest possible instruction. (Segment overrides and FAR are exceptions to this pattern.) But the result is an extra code that does nothing. Even though MACRO-86 figures this out by the second pass, it cannot reduce the size of the instructions without causing an error.

For this reason, MACRO-86 includes several operators that help the assembler. These operators tell MACRO-86 the size of the instruction to generate when it has to make a choice without sufficient data. You can also use these operators to change the nature of the instruction arguments and reduce the size of your program.

For example:

```
MOV AX,FOO ;FOO = forward constant
```

tells MACRO-86 to generate a move from memory instruction on the first pass. If you use the OFFSET operator, MACRO-86 generates an immediate operand instruction.

```
MOV AX,OFFSET FOO ;OFFSET
```

tells MACRO-86 to use the address FOO. In this case, the assembler knows that the value is immediate (saving a byte of code).

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

CALL FAR PTR <forward-label >

On the other hand, you may have a JMP forward that is less than 127 bytes. You can save a byte if you use the SHORT operator.

```
JMP SHORT <forward-label >
```

Be sure that the target is within 127 bytes; otherwise, MACRO-86 can't find it.

The PTR operator is also used to save a byte when using forward references. If you defined FOO as a forward constant, entering the statement:

```
MOV [BX],FOO
```

If you want FOO to be a byte immediate, you enter either of these statements (they are equivalent):

```
MOV BYTE PTR [BX],FOO
MOV [BX],BYTE PTR FOO
```

Both statements tell MACRO-86 that FOO is a byte immediate, and a smaller instruction is generated.

OPERANDS 4.2

There are three types of operands: immediate, register, and memory. There are no restrictions on combining the various types of operands.

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

Exhibit 4b: Contents of Operand Types

OPERAND TYPE	ITEMS CONTAINED IN OPERAND
Immediate	Data items, symbols
Register	
Memory:	
Direct	Labels, variables, offset (field name)
Indexed	Base register, index register, [constant], displacement (plus/minus)
Structure	

IMMEDIATE OPERANDS

Immediate operands are constant values that you supply when entering a statement line. The value is entered either as a data item or as a symbol.

If an instruction takes two operands, you can use an immediate operand only as a source operand (the second operand in an instruction statement). For example:

```
MOV AX,9
```

Data Items

The default input radix is decimal. If you enter a numeric value without appending numeric notation, MACRO-86 treats it as a decimal value. Nondecimal values are recognized when special notation is appended; these values include ASCII characters and numeric values.

Exhibit 4c: Format of Data Types Contained in Operands

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 (first 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 (The first digit must be 0-9. The total number of digits must be 8, 16 or 20; or 9, 17, 21 if first digit is 0.)

Symbols

Symbols are names representing constants. They can be used as immediate operands. In a statement, you can use a symbol constant in the same way you would use a numeric constant. If you continue with the sample statement we've used in the last few examples, you can enter:

```
MOV AX,FOO
```

if FOO is defined as a symbol constant. For example:

```
FOO EQU 9
```

REGISTER OPERANDS

The 8086 processor contains a number of registers; each is identified by two-letter symbols recognized by the parser. Each register has a different task. There are general registers, pointer registers, counter registers, index registers, segment registers, and a flag register. You can use any of these (except segment registers and flags) as an operand in arithmetic and logical operations.

General Registers

The general registers are both 8-bit and 16-bit. All other registers are 16-bit. The 16-bit general registers consist of a pair of 8-bit registers: one for the low byte (bits 0-7) and one for the high byte (bits 8-15). Each 8-bit general register, however, contains bits 0-7 and can be used independently from its mate.

Segment Registers

Segment registers contain segment base values that you initialize. You can use segment register names (CS, DS, SS, ES) with the colon (a segment override operator) to tell MACRO-86 that an operand is not in the segment specified in an ASSUME statement.

Flag Register

The flag register is a single 16-bit register that contains nine 1-bit flags (six arithmetic flags and three control flags).

Exhibit 4d: Register/Memory Field Encoding

REGISTER MODE (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

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 is a byte value; D16 is a word value.

OTHER REGISTERS:

CS = Code segment
DS = Data segment
SS = Stack segment
ES = Extra segment

FLAGS:

ARITHMETIC FLAGS	CONTROL FLAGS
CF = Carry flag	DF = Direction flag
PF = Parity flag	IF = Interrupt-enable
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. When these registers are enclosed in square brackets, they are memory operands; otherwise, they are register operands.

MEMORY OPERANDS

A memory operand represents an address in memory. When you use a memory operand, MACRO-86 goes to a particular address to find data or instructions. A memory operand always consists of an offset from a base address.

Memory operands fit into three categories:

- Direct memory operands that do not use a register.
- Indexed memory operands that use a base or index register.
- Structure operands.

Direct Memory Operands

Direct memory operands do not use registers and they consist of a single offset value. Direct memory operands are labels, simple variables, and offsets.

You can use memory operands as destination operands or as source operands in instructions that take two operands. For example:

```
MOV   AX,FOO
MOV   FOO,CX
```

Indexed Memory Operands

Indexed memory operands use base and index registers, constants, displacement values, and variables — often in combination. Each time 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. The square brackets are treated like plus signs. So, FOO[5] is the same as FOO + 5, and 5[FOO] is equivalent to 5 + FOO.

The only difference between square brackets and plus signs is when a register name appears inside the square brackets. In this case, 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], [FOO].
- + Displacement: 8-bit or 16-bit value. Used only with another indexed operand.

You can combine these elements in any order. The only restriction is that no two base registers or indexed registers can be combined.

Some examples of indexed memory operand combinations:

```
[BP + 8]
[SI + BX][4]
16[DI + BP + 3]
8[FOO] - 8
```


More examples of equivalent forms:

```
5[BX][SI]
[BX + 5][SI]
[BX + SI + 5]
[BX]5[SI]
```

Structure Operands

Structure operands have this form:

<variable>.<field>.

where:

<variable>is a name that initializes a structure field when you are coding a statement line. The variable can be an anonymous variable (such as an indexed memory operand).

<field>is a name defined by a Define directive within a STRUC block. The field is a typed constant.

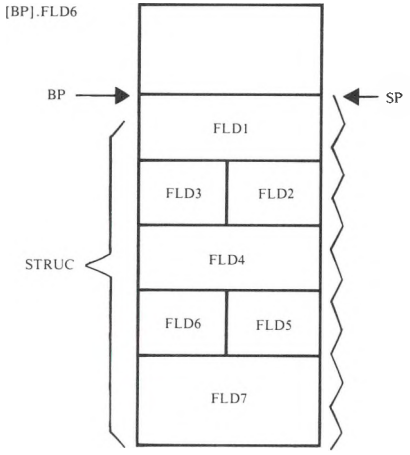
You must include the period between the elements of a structure operand.

Example:

```
ZOO          STRUC
GIRAFFE      DB  ?
ZOO          ENDS

LONG_NECK    ZOO  <16>

MOV AL,LONG_NECK.GIRAFFE

MOV AL,[BX].GIRAFFE   ;anonymous variable
```

Structure operands are helpful in stack operations. If you set BP to the top of the stack (BP = SP), then you can access any value in the stack structure by a field name indexed through BP.

Exhibit 4e: Use of Structure Operand in a Stack Operation



With this method, all values on the stack are available at all times, not just the value at the top. The stack, then, is a handy place for passing parameters to subroutines.

OPERATORS 4.3

There are four types of operators: attribute, arithmetic, relational, and logical. Attribute operators are used with operands to override their attributes, return the value of the attributes, or isolate fields of records. Arithmetic, relational, and logical operators are used to combine or compare operands.

ATTRIBUTE OPERATORS

An attribute operator used as an operand performs one of three functions. It can:

- 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.

Exhibit 4f: MACRO-86 Attribute Operators

OVERRIDE OPERATORS	VALUE-RETURNING OPERATORS	RECORD-SPECIFIC OPERATORS
PTR	SEG	Shift count
: (segment override)	OFFSET	(Field name)
SHORT	TYPE	WIDTH
THIS	.TYPE	MASK
HIGH	LENGTH	
LOW	SIZE	

Override Operators

These operators override the segment, offset, type, or distance of variables and labels.

POINTER (PTR):

<attribute> PTR <expression>

where:

<attribute> is the new type or new distance.

<expression> is the operand whose attribute is to be overridden.

4

PTR overrides the type(BYTE, WORD, DWORD) or the distance (NEAR, FAR) of an operator. PTR is commonly used to ensure that MACRO-86 understands which attribute an expression should have (especially for the type attribute). If your program contains forward references, PTR makes clear the distance or type of the expression and helps you avoid phase errors.

PTR is also used to access data by a type other than that in the variable definition. This usually occurs in structures. For example, you could use PTR if a structure is defined as WORD but you want to access an item as a byte. (A much easier method is to enter a second statement that defines the structure in bytes, which eliminates the need to use PTR for every reference to the structure.)

Examples:

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

ADD   BYTE  PTR  FOO,9
```

SEGMENT OVERRIDE (:):

<segment-register>:<address-expression>
<segment-name>:<address-expression>
<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.

This operator overrides the assumed segment of an address expression (which can be a label, variable, or other memory operand). It tells MACRO-86 the segment, group, or segment register to which a reference is relative.

MACRO-86 assumes that labels are addressable through the CS register and that variables are addressable through the DS register or the ES register (by default). If you have not used ASSUME to tell MACRO-86 that the operand is in another segment, you need to use a segment override operator. You also need to use a segment override operator for forward references, if you want to use a nondefault relative base (i.e., one other than default segment register).

Examples:

```
MOV  EX,ES:[BX + SI]
MOV  CSEG:FAR _ LABEL,AX
MOV  AX,OFFSET DGROUP:VARIABLE
```

SHORT:

SHORT <label>

SHORT overrides the NEAR distance attribute of labels used as targets for the HMP instruction. SHORT tells MACRO-86 that the distance between the JMP statement and the specified label is 127 bytes or less in either direction.

SHORT is helpful if you need to save a byte in your program. Normally, the label carries a 2-byte segment offset pointer. Since SHORT handles a range of 256 bytes in a single byte, it eliminates the need for the second byte.

Example:

```
        JMP SHORT REPEAT
          .
          .
          .
REPEAT:
```

THIS:

THIS has two forms:

THIS <distance>

creates an operand with the distance attribute you specify, an offset equal to the location counter, and the same segment attribute (segment base address) as the enclosing segment.

THIS <type>

creates an operand with the type attribute you specify, an offset equal to the location counter, and the same segment attribute (segment base address) as the enclosing segment.

Each of the following pairs are equivalent:

```
TAG EQU THIS BYTE
TAG LABEL BYTE

SPOT_CHECK = THIS NEAR
SPOT_CHECK LABEL NEAR
```

HIGH, LOW:

HIGH and LOW are byte-isolation operators that provide 8080 assembly language compatibility.

HIGH <expression>

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

LOW <expression>

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

Since MACRO-86 variables have three attributes, you need value returning operators to isolate single attributes. These operators are available:

- SEG isolates the segment base address.
- OFFSET isolates the offset value.
- TYPE isolates either type or distance.
- LENGTH and SIZE isolate the memory allocation.

These operators return the attribute values of the operands that follow them but do not override the attributes. All of them take labels and variables as their arguments.

SEG:

SEG returns the segment value (segment base address) of the segment enclosing the label or variable. It has two forms:

SEG <label>
SEG <variable>

Example:

```
MOV AX,SEG VARIABLE_NAME
MOV AX,<segment-variable>:<variable>
```

OFFSET:

OFFSET returns the segment offset value (the number of bytes between the segment base address and the address where a label or variable is defined) of a variable or label. It has these forms:

OFFSET <label>
OFFSET <variable>

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

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. MACRO-86 applies an implicit OFFSET to variables in address expressions following DW and DD.

Example:

```
MOV BX,OFFSET FOO
```

If you use an ASSUME to GROUP, OFFSET does not automatically return the offset of a variable from the base address of the group. Unless you use the segment override operator, OFFSET returns the segment offset. For example, if you want to get the offset of the variable GOB that is defined in a segment placed in DGROUP, enter a statement such as:

MOV BX,OFFSET DGROUP:GOB

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

TYPE:

This operand has two forms:

TYPE <label>

returns a value equal to the number of bytes of the variable type, as follows:

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

TYPE <variable>

returns NEAR (EFFFH) or FAR (FFFEH).

Examples:

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

.TYPE:

.TYPE <variable>

The .TYPE operator returns a byte that describes two characteristics of the variable: (1) the mode, and (2) whether or not the variable is External. You can use any expression (string, numeric, logical) as an argument. If the expression is invalid, .TYPE returns zero.

.TYPE returns a byte configured as follows:

- The lower two bits are the mode. If 0, the mode is absolute; if 1, the mode is program related; if 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.

The .TYPE operator is usually used inside macros where you may need to test an argument type to make a decision about program flow (i.e., in conditional assembly).

Example:

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


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

LENGTH:

LENGTH <variable>

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for a variable. LENGTH accepts only one variable as its argument.

If a variable is defined by a DUP expression, LENGTH returns the number of type units duplicated (that is, the number that precedes the first DUP in the expression). If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples:

```
FOO DW 100 DUP(1)

MOV CX,LENGTH FOO   ;get number of elements
                    ;in array
                    ;LENGTH returns 100
```

In this example:

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

LENGTH BAZ is still 100, regardless of the expression following DUP. In this example, however:

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE:

SIZE <variable>

SIZE is the product of the value of LENGTH times the value of TYPE. It tells you the total number of bytes allocated for a variable.

Example:

```
FOO DW 100 DUP(1)
MOV BX,SIZE FOO ;get total bytes in array

SIZE = LENGTH X TYPE
SIZE = 100 X WORD
SIZE = 100 X 2
SIZE = 200
```

Record-Specific Operators

Records are defined by the RECORD directive; each can be up to 16 bits long. Each record is defined by fields, which range from one to 16 bits long. To isolate one of the three characteristics of a record field, you need one of these record-specific operators:

- Shift count: The number of bits from low end of record to low end of field.
- WIDTH: The width of a field or record, expressed in bits.
- MASK: The value of record if field contains its maximum value and all other fields are zero.

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

- FOO: A record defined by the RECORD directive:

  ```
  FOO RECORD FIELD1:3,FIELD2:6,FIELD3:7
  ```

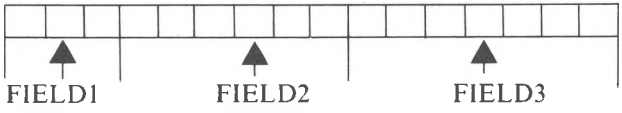
- BAZ: A variable used to allocate FOO.
- FIELD1, FIELD2, FIELD3: The fields of the record FOO.

SHIFT-COUNT:

<record-fieldname>

This shift count is derived from the record field name to be isolated. The shift count is the number of bits the field must be right-shifted in order to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (FOO) contains three fields (FIELD1, FIELD2, and FIELD3), the record is diagrammed as follows:



FIELD1 has a shift count of 13. FIELD2 has a shift count of 7. FIELD3 has a shift count of 0.

When you want to isolate the value in one of these fields, you enter its name as an operand.

Example:

```
MOV DX,BAZ
MOV CL,FIELD2
SHR DX,CL
```

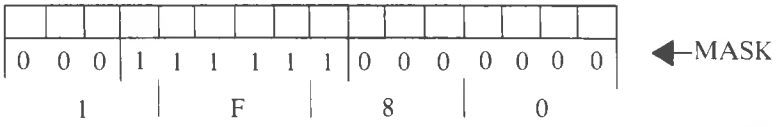
FIELD2 is now right-shifted and ready for access.

MASK:

MASK <record-fieldname>

MASK accepts a field name as its only argument. It returns a bit-mask defined by 1 for those bit positions included by the field, and 0 for bit positions not included. The value returned is the maximum value for the record when the field is masked.

Using the diagram for shift count, MASK is diagrammed as:



The MASK of FIELD2 equals 1F80H.

Example:

```
MOV DX,BAZ
AND DX,MASK FIELD2
```

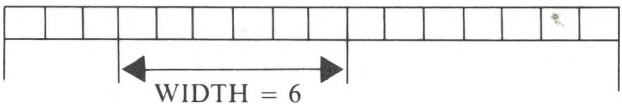
FIELD2 is now isolated.

WIDTH:

WIDTH <record-fieldname>
WIDTH <record>

When <record-fieldname> is given as the argument, WIDTH returns the width of a record field expressed in bits. If <record> is given as the argument, WIDTH returns the width of a record, expressed in bits.

Using the diagram for shift count, WIDTH is 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.

ARITHMETIC OPERATORS

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

The arithmetic operators combine operands to form an expression that results in a data item or an address. These restrictions must be observed:

- Operands must be constants, except for + and − (binary).
- For plus (+), one operand must be a constant.
- For minus (−), the first (left) operand can be a nonconstant or both operands can be nonconstants. However, the right operand cannot be a nonconstant if the left is a constant.

Here is a list of MACRO-86 arithmetic operators:

*	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 right-shifted.

Example:

```
MOV AX,1100000B SHR 5
```

The value moved into AX is 11B (03).

SHL Shift Left. SHL is followed by an integer that specifies the number of bit positions the value is to be left-shifted.

Example:

```
MOV AX,0110B SHL 5
```

The value moved into AX is 011000000B (0C0H).

– Unary Minus. Indicates that following value is negative, as a negative integer.

+ Add. One operand must be a constant. The other can be a nonconstant.

– Subtract the right operand from the left operand. The first (left) operand can be a nonconstant or both operands can be nonconstants. However, the right operand can be a nonconstant only if the left is another nonconstant 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. A zero is returned if the relationship between the two operands does not match the operator.

Relational operators are usually used with conditional directives and conditional instructions to direct program control.

These relational operators are available for use:

- 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 the binary values of corresponding bit positions in each operand. Logical operators are used in two ways:

1. To combine operands in a local relationship. All bits in the operands have the same value (either 0000 or FFFFH).

2. In bitwise operations. In this case, the bits are different and the logical operators act like the instructions of the same name.

These are the MACRO-86 logical operators:

- NOT: Logical NOT. Unary operator which returns false if operand is true, returns true if operand is 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 one 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

When expressions are evaluated, the higher-precedence operators are evaluated first. Equal-precedence operators are evaluated from left to right. 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 * have equal precedence. Their functions are performed in the order in which the operators are encountered (left to right).

In the following list, all operators in a single item have the same precedence, regardless of their order within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: Parentheses ( )
   angle brackets < >
   square brackets [ ]
   structure variable operand: <variable>.<field>
2. Segment override operator (:)
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

5

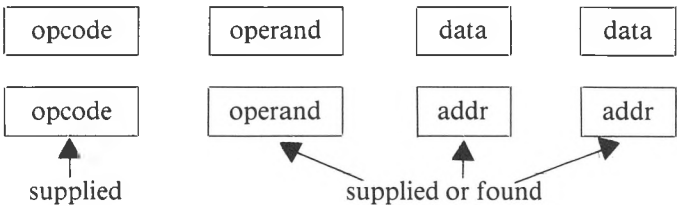
ACTION: INSTRUCTIONS AND DIRECTIVES

MACRO-86 receives your instructions in the action field. The field contains either an 8086 instruction mnemonic that tells the processor to perform a specific function or a MACRO-86 assembler directive.

Action field entries can begin in any column, following a name field entry (if any). Specific spacing is not required; consistent spacing only helps make your program more readable.

INSTRUCTIONS 5.1

Instructions tell the processor what to do. An instruction can have the data and/or addresses it needs built into it, or that data and/or addresses can be found in the expression portion of an instruction. For example:



Supplied: Part of the instruction.

Found: Assembler inserts data and/or address using information provided by expression in instruction statements.

(Opcode is the binary code for the action of an instruction.)

This manual does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this, you can consult the following texts:

- Morse, Stephen P. *The 8086 Primer.* Rochelle Park, NJ: Hayden Publishing Co., 1980.
- Rector, Russell, and George Alexy. *The 8086 Book.* Berkeley, CA: Osbourne/McGraw-Hill, 1980.
- *The 8086 Family User's Manual.* Santa Clara, CA: Intel Corporation, 1980.

Appendix C gives the instruction mnemonics. An alphabetical listing shows the full name of the instruction. Following the alphabetical list, a second listing groups the instruction mnemonics by the number and type of arguments they take.

5.2 DIRECTIVES

Directives give MACRO-86 directions for input and output, memory organization, conditional assembly, listing and cross reference control, and definitions. In this section, the directives are divided into groups by function. The directives are listed alphabetically within each group.

The groups are:

- Memory directives: These organize memory. This group also contains directives (such as COMMENT) that do not organize memory.
- Conditional directives: These 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: These create blocks of code called macros. This group also includes special operators and directives used only inside macro blocks. The repeat directives are considered Macro directives for descriptive purposes.
- Listing directives: These directives control the format and, to some extent, the content of listings produced by the assembler.

Here is an alphabetical list of the directives supported by MACRO-86.

ASSUME	EVEN	LABEL	.RADIX
	EXITM	.LALL	RECORD
COMMENT	EXTERN	.LFCOND	REPT
.CREF		.LIST	
	GROUP		.SALL
DB		MACRO	SEGMENT
DD	IF		.SFCOND
DQ	IFB	NAME	STRUC
DT	IFDEF		SUBTTL
DW	IFDIF	ORG	
	IFE	%OUT	.TFCOND
ELSE	IFIDN		TITLE
END	IFNB	PAGE	
ENDIF	IFNDEF	PROC	.XALL
ENDM	IF1	PUBLIC	.XCREF
ENDP	IF2	PURGE	.XLIST
ENDS	IRP		
EQU	IRPC		

MEMORY DIRECTIVES

ASSUME

ASSUME <seg-reg>:<seg-name>[,...]

or

ASSUME NOTHING

ASSUME tells MACRO-86 that symbols in a segment or group can be accessed with a particular segment register. When MACRO-86 encounters a variable, it automatically assembles the variable reference under the proper segment register. You can use up to four arguments with ASSUME.

The valid <seg-reg> entries are CS, DS, ES, and SS.

The possible entries for <seg-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> (see SEG operator in Chapter 4).
- The key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement.

Unless you use ASSUME (or if NOTHING is entered as the ASSUME segment name), each reference to variables, symbols, labels, and so forth in a particular segment must be prefixed by a segment register. For example, you'd have to use DF:FOO instead of FOO.

Example:

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

COMMENT

COMMENT<delim><text><delim>

COMMENT lets you enter comments about your program without entering a semicolon (;) before each line. If you use COMMENT inside a macro block, the comment does not appear on your listing unless you also put the .LALL directive in your source file.

The first non-blank character encountered after COMMENT is the delimiter. The following text is a comment block that continues until the next occurrence of the specified delimiter.

If you use an asterisk as a delimiter, the format of the comment block is:

```
COMMENT   *
this is the comment block.
you can enter any amount
of text between
the two delimiters
   .
   .
   .     *       ; return to normal mode
```

DEFINE

```
<varname>    DB    <exp>[,<exp>,...]
<varname>    DW    <exp>[,<exp>,...]
<varname>    DD    <exp>[,<exp>,...]
<varname>    DQ    <exp>[,<exp>,...]
<varname>    DT    <exp>[,<exp>,...]
```

The Define directives define variables or initialize portions of memory. They allocate memory in units specified by the second letter of the directive:

- 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.

If a variable name is entered, the Define directives define the name as a variable. If <varname> contains a colon, it becomes a NEAR label instead of a variable. (See the sections on Labels and Variables in Chapter 3.)

The expression used by Define can be one or more of the following:

- A constant expression.
- A question mark (?). This is usually used to reserve space without placing any particular value into it. (It equals the DS pseudo-operator in MACRO-80.)
- An address expression (for DW and DD only).
- An ASCII string. Except with DB, it cannot be longer than 2 characters.
- <exp> DUP (?). When this type of expression is the only argument to a Define, Define produces an uninitialized data block. If used with a question mark, this expression results in a smaller object file because only the segment offset is changed to reserve space.
- <exp> DUP (<exp> [, . . .]). Like the last item, this expression produces a data block, but initializes it with the value of the second expression. The first expression must be a constant greater than zero and cannot be a forward reference.

Here are examples of how the various Define directives are used.

Define Byte (DB):

```
NUM_BASE    DB   16
FILLER      DB   ?                    ;initialize with
                                      ;indeterminate value
ONE_CHAR    DB   'M'
MULT_CHAR   DB   'MARC MIKE ZIBO PAUL BILL'
MSG         DB   'MSGTEST', 13, 10    ;message, carriage return,
                                      ;and linefeed
BUFFER      DB   10 DUP (?)           ;indeterminate block
TABLE       DB   100 DUP (5 DUP (4), 7)
                                      ;100 copies of bytes
                                      ;with values 4,4,4,4,4,7
NEW_PAGE    DB   0CH                  ;form feed character
ARRAY       DB   1,2,3,4,5,6,7
```

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 MACRO-86
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 (?)             ;uninitialized data
FILLER      DQ   1 DUP (?,?)           ;initialized with
                                       ;indeterminate value
HEX_REAL    DQ   0FDCBA9A98765432105R
```

Define Tenbytes (DT):

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

END

END [<exp>]

END specifies the end of the program. Any expression present is the start address of the program. If several modules are to be linked, only the main module can use END (exp) to specify the start of the program.

If no expression is used, then no start address is passed to MS-LINK for that program or module.

Example:

```
END START ;START is a label somewhere in the program
```

EQU

<name> EQU <exp>

EQU assigns the value of the expression to the specified name. EQU is often used like a macro as a primitive text substitution. (If you want to be able to redefine a name in your program, use the Equal-sign directive instead.)

An error is generated if the expression is an external symbol or if the specified name already has a value. The expression used can be any 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:

```
FOO     EQU     BAZ             ;must be defined in this
                                ;module or an error results
B       EQU     [BP+8]          ;index reference (Text)
P8      EQU     DS:
                                ;segment prefix
                                ;and operand (Text)
CBD     EQU     AAD             ;an instruction name (Opcode)
ALL     EQU     DEFREC<2,3,4>   ;DEFREC = record name
                                ;<2,3,4> = initial values
                                ;for fields of record
EMP     EQU     6               ;constant value
FPV     EQU     6.3E7           ;floating point (Text)
```


EQUAL-SIGN

<name> = <exp>

<exp> must be a valid expression. (It is shown as a number or L (label) in the symbol table.) The equal sign lets you set and redefine symbols. The equal sign is used much like the EQU directive, except that you can redefine the symbol without generating an error. You can redefine the symbol as many times as you like, even to a definition that you have already used.

Example:

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

EVEN

EVEN

EVEN causes the program counter to go to an even boundary — that is, to go to an address that begins a word. If the program counter is not already at an even boundary, EVEN tells MACRO-86 to add an NOP instruction so that the counter will reach an even boundary.

Suppose the PC in your program points to 0019 hex (25 decimal). Using EVEN makes the PC point to 1A hex (26 decimal), and the 0019 hex will contain an NOP instruction.

An error results if EVEN is used with a byte-aligned segment.

EXTRN

EXTRN <name>:<type>[,...]

where:

<name> is a symbol defined in another module. <name> must have been declared PUBLIC in the module where it is defined.

<type> can be any one of the following, as long as it is 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).

The placement of the EXTRN directive is significant. If the directive is given with a segment, MACRO-86 assumes that the symbol is located within that segment. If the segment is not known, you should place the directive outside all segments and use either:

```
ASSUME <seg-reg>:SEG <name>
```

or an explicit segment prefix.

NOTE: If you don't place the symbol in the segment, MS-LINK takes the offset relative to the given segment, if possible. If the correct segment is less than 64K bytes from the reference, MS-LINK may find the definition. If the correct segment is more than 64K bytes away, MS-LINK can't make the link between the reference and the definition. An error message is returned.

Example:

In same segment:

In Module 1:

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

In Module 2:

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

In another segment:

In Module 1:

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

In Module 2:

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

GROUP

<name> GROUP <seg-name>[,...]

This directive gives the segments named a single name so that MS-LINK loads them together. (The order in which the segments are named does not affect the order in which they are loaded. This is handled by the CLASS designation of the SEGMENT directive or by the order in which you name object modules when responding to the MS-LINK object module prompt.)

All segments in a Group must fit into 64K bytes of memory.

The segments named in the GROUP directive are one of the following:

- A segment name assigned by a SEGMENT directive. The name can be a forward reference.
- An expression: either SEG<var>or SEG<label>. Both of these resolve themselves to a segment name.

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

- 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
```

  You can use the DS register to reach any symbol in any segment of the group.

- As an operand prefix (for segment override):

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

  DGROUP: makes the offset relative to DGROUP, instead of to the segment where FOO is defined.

In this example, GROUP is used to combine segments.

In Module A:

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

In Module B:

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


INCLUDE

INCLUDE <filename>

INCLUDE takes source code from an alternate assembly language source file and inserts it into the source file during assembly. The INCLUDE directive eliminates the need to repeat an often-used sequence of statements in the source file.

The file name you use can be any valid file specification. Unless you use the default device, you must include a device or drive designation in the file name. (The default device designation is the logged drive or device.)

An INCLUDEd file is opened and assembled into the source file immediately following the INCLUDE statement. When end-of-file is reached, assembly resumes with the next statement.

A file inserted with an INCLUDE statement can contain an INCLUDE directive. However, this can cause problems if you have only a limited amount of memory left.

The file specified in the INCLUDE statement must exist. If the file is not found, MACRO-86 returns an error and aborts the assembly.

On a MACRO-86 listing, the letter C is printed between the assembled code and the source line on each line that is assembled from an INCLUDEd file. (See Chapter 6 for a description of listing file formats.)

Example:

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


LABEL

<name> LABEL <type>

By using LABEL to define a name, you tell MACRO-86 to associate the current segment offset with the name you have defined. The item is assigned a length of 1.

The type varies depending on the use of the defined name.

The name you define can be used for code or for data.

FOR CODE: If you use LABEL for code (for example, as a JMP or CALL operand), the type can be NEAR or FAR. The name cannot be used in data manipulation instructions without using a type override.

You can define a NEAR label using the <name>: form. If you are defining a BYTE or WORD NEAR label, you can place the <name>: in front of a Define directive. When you use a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

Example:

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

FOR DATA: The type can be BYTE, WORD, DWORD, STRUC <name> or RECORD <name>. When STRUC <name> or RECORD <name> is used, <name> is the size of the structure or record.

Example:

```
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
```


Since you defined the array in two ways, you can access entries by byte or by word. (You can also use this method for STRUC.) This lets you put your data in memory as a table and to access it without the offset of the STRUC.

By defining the array in two ways you can avoid using the PTR operator. This is especially effective if you access the data in different ways; giving the array a second name is easier than remembering PTR.

NAME

NAME **<module-name>**

where:

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

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

- A valid NAME directive statement.

► The first six characters of a TITLE directive statement, if the module does not contain a NAME statement. These characters must be legal as a name.

NAME passes the module name to MS-LINK, but otherwise does not affect MACRO-86. MACRO-86 does check if more than one module name has been declared.

Example:

```
NAME CURSOR
```

ORG

ORG **<exp>**

The location counter is set to the value of the expression. MACRO-86 assigns generated code starting with that value.

All names in the expression must be known on the first pass. The value of the expression must evaluate to an absolute or else it must be in the same segment as the location counter.

Example:

```
ORG     120H        ;2-byte absolute value
                    ;maximum = OFFFFH
ORG     $+2         ;skip two bytes
```

To 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
```

PROC

```
<procname>      PROC      [NEAR]
or
<procname>      PROC      [FAR]
                .
                .
                .
                RET
<procname>  ENDP
```

PROC is a structuring device that makes your programs more understandable. Using the NEAR/FAR option, PROC tells CALLs which procedure to use to generate a NEAR or FAR CALL. PROC also tells RETs to generate a NEAR or FAR RET, eliminating the need for you to assign NEAR or FAR to CALLs and RETs.

If no operand is specified, the default is NEAR. Use FAR if the procedure name is an operating system entry point or if the procedure is called from code that has another ASSUME CS value. Each PROC block should contain a RET statement. PROCs can be nested.

5

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.

If you combine the PUBLIC directive with a PROC statement (both NEAR and FAR), you can make external CALLs to the procedure or 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 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 must call to the first subroutine, which then calls the second (an indirect call):

```
CALL FAR_NAME
```

PUBLIC

PUBLIC <symbol>[,...]

where:

<symbol> is a number, a variable or a label (including PROC labels). <symbol> cannot be a register name or a symbol defined by floating point numbers (with EQU) or by integers larger than 2 bytes.

Put a PUBLIC directive statement in any module containing a symbol you want to use in other modules without redefining it. PUBLIC makes the listed symbol(s) available to other modules when they are linked to the module that defines the symbol(s). This information is passed to MS-LINK.

Example:

```
            PUBLIC  GETINFO
GETINFO     PROC    FAR
            PUSH    BP          ;save caller's register
            MOV     BP,SP       ;get address parameters
                                ;body of subroutine
            POP     BP          ;restore caller's reg
            RET                 ;return to caller
GETINFO     ENDP
```

This is an illegal use of PUBLIC:

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

.RADIX

.RADIX **<exp>**

where:

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

This directive lets you change the input radix to any base in the range from 2 to 16. The default input base (or radix) for all constants is decimal.

The two MOVs in this example are identical.


```
MOV     BX,0FFH
.RADIX  16
MOV     BX,0FF
```

The .RADIX directive does not affect the generated code values in the .OBJ, .LST, or .CRF output files. It does not affect the DD, DQ, or DT directives. Expressions entered in these directives are always evaluated as decimal numeric values unless a data-type suffix is appended.

Example:

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

RECORD

<recordname> RECORD <fieldname>:<width>[=<exp>] [,...]

where:

<fieldname> is the name of the field. When you use <fieldname> in an expression, its value is the shift count for moving the field to the far right. If you use MASK with <fieldname>, it returns a bit mask for that field.

<width> is a number from 1 to 16 returned by the WIDTH operator. It specifies the number of bits in the field defined by <fieldname>. If the total width of all declared fields is larger than 8 bits, then MACRO-86 uses two bytes. Otherwise, only one byte is used.

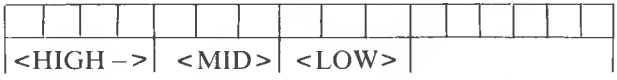
<exp> contains the initial (or default) value for the field. Forward references are not allowed in a RECORD statement.

The first field you declare goes into the most significant bits of the record. Successive fields are placed in the bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is right-shifted so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

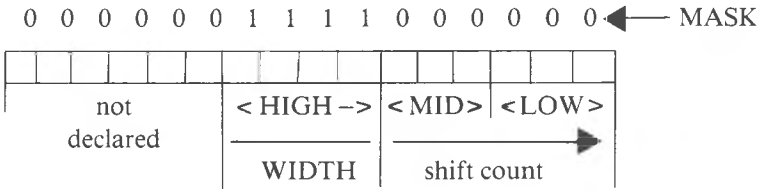
If you enter:

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

Initially, the bit map is:



If the bit total is less than 8, MACRO-86 uses a word; if the total is less than 16, MACRO-86 shifts them right and places undeclared bits at the high end of the word.



<exp> contains the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character as the <exp>.

For example:

```
HIGH: 7 = 'Q'
```

To initialize records, use the same method used for DB. The format is:

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

5

The name is optional. When given, the name is a label for the first byte or word of the record storage area. The record name is the name used as a label by the RECORD directive.

In both forms, the expression contains the values you want placed into the fields of the record. In the second form, parentheses and angle brackets are required only around the second expression. If [exp] is left blank, either the default value applies (the value given in the original record definition) or the value is determinant (not initialized in the original record definition). If a field is initialized to a value you want, enter consecutive commas (,,) so that MACRO-86 uses the default values of those fields.

For example:

```
FOO <,,7 >
```

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

Records are used in expressions (as an operand) in the form:

recordname<[value[,...]]>

The value entry is an optional value placed into a field of the record. Angle brackets must be entered as shown, even if you don't give the optional values. If a value already has the value you want, enter consecutive commas so that MACRO-86 uses the default values.

Example:

```
FOO     RECORD  HIGH:5,MID:3,LOW:3
        .
        .
        .
BAX     FOO     <  > ;leave undeterminate here
JANE    FOO     10 DUP(<16,8>)   ;HIGH = 16,MID = 8,
                ;LOW = ?
        .
        .
        .
        MOV     DX,OFFSET JANE [2]
                ;get beginning record address
        AND     DX,MASK MID
        MOV     CL,MID
        SHR     DX,CL
        MOV     CL,WIDTH MID
```


SEGMENT

<segname> SEGMENT [<align>] [<combine>] [<'class '>]

.
.
.

<segname> ENDS

where:

<segname> is a unique, legal name. The segment name must not be a reserved word.

<align>is PARA (paragraph-default), BYTE, WORD, or PAGE.

<combine> is PUBLIC, COMMON, AT <exp>, STACK, MEMORY, or no entry (which defaults to not combinable).

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

All three operands are passed to MS-LINK.

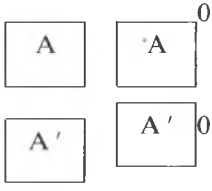
At run time, all instructions that generate code and data are in separate segments. Your program can be a segment, part of a segment, several segments, parts of several segments or a combination. If your program has no SEGMENT statement, an MS-LINK error (invalid object) results at link time.

The alignment lets the linker know the kind of boundary where you want the segment to begin. The first address of the segment is (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 xxxeH (e = even number; low bit is 0).
  Bit map: |x|x|x|x|x|x|x|0|
- BYTE: Address is xxxxxH (place anywhere).

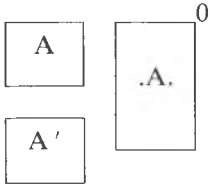
The combine type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

- None (not combinable or Private)



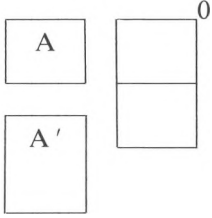
Private segments are loaded separately and remain separate. They can be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

► Public and Stack



Public segments of the same name and class name name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is 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 encountered is the highest segment in memory. Subsequent segments are treated the same as Common segments.

NOTE: This combine type 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 cannot be a forward reference. Also, the AT type cannot be used to force loading at fixed addresses. Instead, AT lets you define labels and variables at fixed offsets within fixed areas of storage (such as ROM or the vector space in low memory). This restriction is imposed by MS-LINK and MS-DOS.

Class names must be enclosed in quotation marks. Class names can be any legal name. See MS-LINK in this volume for more information.

Segment definitions can be nested. When this is done, segments are handled sequentially by appending the second part of the split segment to the first. When MACRO-86 reaches a nested segment, it treats that segment as a new segment. MACRO-86 completes it, and then moves on to the remaining portion of the surrounding segment. Overlapping segments are not permitted.

These two segment arrangements are legal:

```
A  SEGMENT          A  SEGMENT
     .                   .
     .                   .
     .                   .
B  SEGMENT          A  ENDS
     .              B  SEGMENT
     .                   .
     .                   .
B  ENDS                  .
     .              B  ENDS
     .              A  SEGMENT
     .                   .
A  ENDS                  .
                         .
                    A  ENDS
```

The following arrangement is not allowed:

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

Here is another legal use of SEGMENT:

In Module A:

```
SEGA    SEGMENT   PUBLIC 'CODE'
        ASSUME    CS:SEGA
          .
          .
          .
SEGA    ENDS
        END
```

In Module B:

```
SEGA    SEGMENT   PUBLIC 'CODE'
        ASSUME    CS:SEGA
          .       ;MS-LINK adds this segment to same
          .       ;named segment in module A (and
          .       ;others) if class name is the same.
SEGA    ENDS
        END
```


STRUC

```
<structurename>    STRUC
                     .
                     .
                     .
<structurename>    ENDS
```

The STRUC directive is much like RECORD, except that it has a multiple byte capability. The allocation and initialization of a STRUC block is the same as for RECORDs.

The Define directives (DB, DW, DD, DQ, DT) are used to allocate space inside the STRUC block. The Define directives and comments are the only statement entries allowed.

Any Define directive label inside a STRUC block becomes a field name of the structure (which is how structure field names are defined). Initial values given to field names in the STRUC block are default values for the various fields.

These field values are either overridable or not overridable. A field with only one entry (but not a DUP expression) is overridable. A field with more than one entry is not overridable.

For example:

```
FOO   DB   1,2          ;is not overridable
BAZ   DB   10 DUP (?)   ;is not overridable
ZOO   DB   5            ;is overridable
```

If the expression following the Define directive contains a string, it can be overridden by another string. If the overriding string is shorter than the initial string, the assembler pads it with spaces. If the overriding string is longer, MACRO-86 truncates the extra characters.

Structure fields are usually used as operands in an expression. The format for a reference to a structure field is:

<variable>. field>

where:

<variable>is an anonymous variable, usually set up when the structure is allocated.

.<field>is a label given to a Define directive inside a STRUC/ENDS. The value of the field is the offset within the addressed structure.

If you want to allocate a structure, use the structure name as a directive with a label and enclose any override values in angle brackets:

```
FOO   STRUCTURE
      .
      .
      .
FOO   ENDS
GOO   FOO <,7,, 'JOE '>
```

To define a structure:

```
S       STRUC
FIELD1  DB      1,2             ;not overridable
FIELD2  DB      10 DUP (?)      ;not overridable
FIELD3  DB      5               ;overridable
FIELD4  DB      'DOBOSKY'       ;overridable
S       ENDS
```

In this example, the Define directives define the fields of the structure. The order corresponds to the values given in the initialization list when the structure is allocated. Each Define directive 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

With conditional directives, you can design blocks of code which test for specific conditions and then proceed accordingly.

All conditionals have this format:

```
IFxxxx [argument]
    .
    .
    .
[ELSE
    .
    .
    .  ]
ENDIF
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Otherwise, an "Unterminated conditional" message appears at the end of each pass. If you enter an ENDIF without a matching IF, you'll get a "Code 8, Not in conditional block" error.

Each conditional block can include the optional ELSE directive. This directive allows alternate code to be generated when the opposite condition exists. An ELSE is bound to the most recent, open IF; only one ELSE is allowed for a given IF.

A conditional with more than one ELSE or an ELSE without a conditional causes a "Code 7, Already had ELSE clause" error.

Conditionals can be nested up to 255 levels. An argument to a conditional must be known on the first pass to avoid phase errors and incorrect evaluation. For IF and IFE conditionals, values in the expression must have been previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, MACRO-86 considers the name undefined on the first pass, but defined on the second pass.

MACRO-86 evaluates a conditional statement to TRUE (which equals any non-zero value) or to FALSE (which equals 0000H). If this evaluation matches the condition defined in the conditional statement, MACRO-86 assembles the whole conditional block or (if the block contains an ELSE directive) assembles from IF to ELSE. The ELSE-to-ENDIF portion of the block is ignored. If the evaluation does not match, MACRO-86 ignores the conditional block completely or (if the block contains an ELSE directive) assembles only the ELSE to ENDIF portion. The IF-to-ELSE portion is ignored.

These are the MACRO-86 conditional directives:

IF <exp>

If the value of the expression is non-zero, statements within the conditional block are assembled.

IFE <exp>

If the value of the expression is zero, statements in the conditional block are assembled.

IF1

Pass 1 conditional. If MACRO-86 is in the first pass, statements in the conditional block are assembled. IF1 takes no expression. For IF1 use, refer to %OUT.

IF2

Pass 2 conditional. If MACRO-86 is in the second pass, statements in the conditional block are assembled. IF2 takes no expression. For IF2 use, refer to %OUT.

IFDEF <symbol>

Statements in the conditional block are assembled if <symbol> is defined or has been declared external.

IFNDEF <symbol>

Statements in the conditional block are assembled if <symbol> is not defined or not declared external.

IFB <arg>

Statements in the conditional block are assembled if <arg> is blank (none given) or null (two angle brackets with nothing between). The angle brackets around <arg> are required.

IFB is usually used inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy symbol is replaced by a parameter passed by the macro call. If no parameter is specified, the expression is blank and the block is assembled.

IFNB <arg>

If <arg> is not blank, the statements in the conditional block are assembled. The angle brackets around <arg> are required.

IFNB is normally used inside macro blocks. The expression following the IFNB directive is generally a dummy symbol. When the macro is called, the dummy is replaced by a parameter passed by the macro call. When this happens, the expression is not blank and the block will be assembled.

IFIDN <arg1>,<arg2>

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled. The angle brackets around <arg1> and <arg2> are required.

IFIDN is generally used inside macro blocks. The expression after IFIDN is typically two dummy symbols. When the macro is called, the dummies are replaced by parameters passed by the macro call. If two identical parameters are specified, the block is assembled.

IFDIF <arg1>,<arg2>

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled. The angle brackets around <arg1> and <arg2> are required.

IFDIF usually occurs inside macro blocks. The expression IFDIF is typically two dummy symbols. When the macro is called, the dummies are replaced by parameters passed by the macro call. If two different parameters are specified, the block is assembled.

ELSE

ELSE lets you generate alternate code when the opposite condition exists. It can be used with any of the conditional directives; however, only one ELSE is allowed for each IFxxxx conditional directive. ELSE takes no expression.

ENDIF

ENDIF terminates a conditional block by closing the most recent unterminated IF. An ENDIF directive must be given for every IFxxxx directive used. ENDIF takes no expression.

MACRO DIRECTIVES

Macro directives let you write blocks of code which can be repeated without recoding. These blocks begin with the Macro definition directive or one of the repetition directives and end with the ENDM directive. All Macro directives can be used inside a macro block. Nesting of macros is limited only by memory. MACRO-86 has Macro directives for:

- Macro definition: MACRO.
- Termination: ENDM, EXITM.
- Unique symbols within macro blocks: LOCAL.
- Undefining a macro: PURGE.

► Repetitions: REPT (repeat), IRP (indefinite repeat), and IRPC (indefinite repeat character).

The Macro directives also include these special macro operators:

& ;; ! %

Macro Definition

<name> MACRO [<dummy>,...]

.
.
.

ENDM

where:

<name> is like a LABEL and conforms to the rules for forming symbols. After the macro is defined, <name> is used to invoke the macro.

< dummy > is a place holder that's replaced by a parameter when the macro block is used. All dummies inside the macro should be included on the same line. A dummy is formed in the same way as any other name.

The block of statements from the macro statement line to the ENDM statement line is the body of the macro (the macro's definition).

Macro is a very powerful directive. With it, you can change the value and effect of any instruction, directive, label, variable, or symbol. When MACRO-86 evaluates a statement, it first looks at the macro table it builds during the first pass. If it sees a name that matches an entry in a statement, it replaces that entry with the macro it found in the table. (Remember: MACRO-86 evaluates macros first, then goes on to instructions and directives.)

The number of dummies you can use with a Macro directive is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. MACRO-86 interprets a series of dummies the same way it interprets any list of symbol names.

NOTE: A dummy is always recognized only as a dummy. Even if you use a register name (such as AX or BH) as a dummy, MACRO-86 replaces it with a parameter during expansion.

Another way to use the Macro directive is to list no dummies:

<name>MACRO

This lets you call the block repeatedly, even if you don't need to pass parameters to the block. In this case, of course, the block contains no dummies.

A macro block is not assembled when it is encountered. Instead, MACRO-86 "expands" the macro call statement by bringing in and assembling the appropriate macro block.

If you want to use the TITLE, SUBTTL, or NAME directives as the portion of your program where a macro block appears, you should be careful about the form of the statement. For example, if you enter:

SUBTTL MACRO DEFINITIONS

MACRO-86 assembles the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, change the word macro in some way — use macroe, macros, and so on.

CALLING A MACRO: To use a macro, enter a macro call statement:

<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 put angle brackets around parameters separated by commas, the assembler passes the items inside the angle brackets as a single parameter.

For example:

```
FOO 1,2,3,4,5
```

passes five parameters to the macro, but:

```
FOO <1,2,3,4,5>
```

passes only one.

You don't need to use the same number of parameters in a macro call statement and the macro definition. If there are more parameters than dummies, MACRO-86 ignores the extras. If there are fewer, the extra dummies are null. The assembled code includes the macro block after each macro call statement.

Suppose you enter the following:

```
GEN   MACRO  XX,YY,ZZ
      MOV    AX,XX
      ADD    AX,YY
      MOV    ZZ,AX
      ENDM
```

If you enter a macro call statement:

```
GEN   DUCK,DON,FOO
```

assembly generates the statements:

```
MOV   AX,DUCK
ADD   AX,DON
MOV   FOO,AX
```

On your program listing, these statements are preceded by a plus sign to show that they came from a macro block.

END MACRO

ENDM

ENDM tells MACRO-86 that the macro or repeat block is ended. Each 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 want to exit from a macro or repeat block before expansion is completed, use EXITM.

EXIT MACRO

EXITM

EXITM is used inside a macro or repeat block to stop an expansion when continuing becomes unnecessary or undesirable.

When an EXITM is assembled, the expansion is halted immediately. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

EXITM is usually used with a conditional directive.

Example:

```
FOO   MACRO   X
X     =       0
      REPT    X
X     =       X+1
      IFE     X-OFFH   ;test X
      EXITM   ;if true, exit REPT
      ENDIF
      DB      X
      ENDM
      ENDM
```

LOCAL

LOCAL <dummy>[,<dummy>...]

The LOCAL directive is allowed only inside a macro definition block. A LOCAL statement must precede all other types of statements in the macro definition.

When LOCAL is executed, MACRO-86 creates a unique symbol for each <dummy> and substitutes that symbol for each occurrence of the <dummy>

during expansion. These unique symbols usually define labels within a macro and eliminate multiple-defined labels on successive expansions of the macro. You should avoid creating symbols with the form ??nnnn, since MACRO-86 uses the same form when it creates its own symbols.

Example:

```
0000                    FUN     SEGMENT
                                ASSUME  CS:FUN,DS:FUN
                        FOO     MACRO   NUM,Y
                                LOCAL   A,B,C,D,E
                        A:      DB      7
                        B:      DB      8
                        C:      DB      Y
                        D:      DW      Y+1
                        E:      DW      NUM+1
                                JMP     A
                                ENDM
                                FOO     0C00H,0BEH
0000  07        +       ??0000: DB      7
0001  08        +       ??0001: DB      8
0002  BE        +       ??0002: DB      0BEH
0003  00BF      +       ??0003: DW      0BEH+1
0005  0C01      +       ??0004: DW      0C00H+1
0007  EB F7      +               JMP     ??0000
                                FOO     03C0H,0FFH
0009  07        +       ??0005: DB      7
000A  08        +       ??0006: DB      8
000B  FF        +       ??0007: DB      0FFH
000C  0100      +       ??0008: DW      0FFH+1
000E  03C1      +       ??0009: DW      03C0H+1
0010  EB F7                     JMP     ??0005
0012                    FUN     ENDS
                                END
```

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

PURGE

PURGE <macro-name> [,...]

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

PURGE provides three benefits:

1. It frees text space in the macro body.

2. Any instruction or directive redefined by a macro is returned to its original function.

3. It edits out macros from a macro library file. This lets you use macros repeatedly with easy access to their definitions. Typically, you place an INCLUDE statement in your program file. After the INCLUDE, place a PURGE statement to delete any macros you won't use in your program.

You don't need to PURGE a macro before you redefine it. All you need to do is 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 let you repeat an operation for as many times as you specify. They are convenient when you know in advance that a parameter won't change while your program is executing. Unlike with the Macro directive, you don't have to call in the parameter each time it is needed.

Repeat directive parameters must be assigned as a part of the code block. Each Repeat directive must be matched with the ENDM directive to terminate the repeat block.

REPEAT:

```
REPT <exp>
.
.
.
ENDM
```

Repeat statements between REPT and ENDM <exp> times. <exp> is evaluated as a 16-bit unsigned number. An error is generated if <exp> contains an External symbol or undefined operands.

The following example:

```
X   =       0
    REPT    10      ;generates DB 1 - DB 10
X   =       X+1
    DB      X
    ENDM
```


assembles as:

```
0000          X     =       0
                    REPT    10      ;generates DB 1 - DB 10
                    X     =       X+1
                          DB      X
                          ENDM
0000'  01     +     DB      X
0001'  02     +     DB      X
0002'  03     +     DB      X
0003'  04     +     DB      X
0004'  05     +     DB      X
0005'  06     +     DB      X
0006'  07     +     DB      X
0007'  08     +     DB      X
0008'  09     +     DB      X
0009'  0A     +     DB      X
                          END
```

INDEFINITE REPEAT:

IRP<dummy>,<parameters inside angle brackets>
.
.
.
ENDM

Parameters must be enclosed in angle brackets. They can be any legal symbol, string, numeric, or character constant.

The statement block is repeated for each parameter. With each repetition, the next parameter is substituted for every occurrence of <dummy> in the block. If a parameter is null, 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 generates the same bytes (DB 1 – DB 10) as the REPT example.

When you use IRP inside a macro definition block, the angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. The next example generates the same code as the one above, but shows the removal of one level of brackets from the parameters.

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

When the macro call statement:

```
FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion is:

```
IRP     Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

The angle brackets around the parameters are removed and all items are passed as a single parameter.

INDEFINITE REPEAT CHARACTER:

```
IRPC <dummy>,<string>
.
.
.
ENDM
```

The statements in the block are repeated once for each character in the string. With each repetition, the next character in the string is substituted for every occurrence of <dummy> in the block.

Example:

```
IRPC    X,0123456789
DB      X + 1
ENDM
```

This generates the same code (DB 1 – DB 10) as the two previous examples.

Special Macro Operators

Several special operators are used in a macro block to select additional assembly functions.

& Concatenates text or symbols. (Ampersands cannot be used in macro call statements.) A dummy parameter in a quoted string is not substituted in expansion unless preceded by an ampersand. Put an ampersand between text and a dummy to form a symbol.

If you enter:

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

the call ERRGEN A generates:

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

The ampersand does not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. Extra ampersands may be needed for complex macros, where nesting is involved. You need to supply as many ampersands as there are levels of nesting.

For example:

CORRECT FORM

```
FOO     MACRO   X
        IRP     Z,<1,2,3>
X&&&Z   DB      Z
        ENDM
        ENDM
```

INCORRECT FORM

```
FOO     MACRO   X
        IRP     Z,<1,2,3>
X&&Z    DB      Z
        ENDM
        ENDM
```

When the previous example is expanded (when called by FOO BAZ), the expansion follows these steps. (As shown, the correct form is expanded in the left column; the incorrect in the right.)

1. Macro build, find dummies and change to dl

```
        IRP     Z,<1,2,3>           IRP     Z,<1,2,3>
dl&Z    DB      Z           dlZ     DB      Z
        ENDM                        ENDM
```

2. Macro expansion, substitute parameter text for dl

```
        IRP     Z,<1,2,3>           IRP     Z,<1,2,3>
BAZ&Z   DB      Z           BAZZ    DB      Z
        ENDM                        ENDM
```

3. IRP build, find dummies and change to dl

```
BAZ&dl  DB      dl          BAZZ    DB      dl
```

4. IRP expansion, substitute parameter text for dl

```
BAZ1    DB      1           BAZZ    DB      1
BAZ2    DB      2           BAZZ    DB      2 ; error
BAZ3    DB      3           BAZZ    DB      3
```

The error is due to a multi-defined symbol.

<text> The angle brackets tell MACRO-86 to treat the enclosed text as a single literal. If you use them to enclose parameters to a macro call or the list of parameters following the IRP directive inside angle brackets, there can be two results:

1. All text within the angle brackets is seen as a single parameter, even if commas are used.
2. Characters with special functions are taken as literal characters. For example, a semicolon inside angle brackets becomes a character, not an indicator that a comment follows.

One set of angle brackets is removed each time you use a parameter in a macro. If you're using nested macros, you need to supply as many sets of angle brackets as there are levels of nesting.

;; Used in a macro or repeat block. A comment preceded by two semicolons is not saved in the expansion.

The default listing condition for macros is .XALL (see "Listing Directives"). Under .XALL, comments in macro blocks are not listed because they do not generate code.

If you put the .LALL listing directive in your program, comments inside macro and repeat blocks are saved and listed. However, this can cause an out-of-memory error. To avoid this, put double semicolons before comments inside macro and repeat blocks unless you want a particular comment to be retained.

!

An exclamation point in an argument indicates that the next character is to be taken literally. So, !; is the same as <;>.

%

Used only in macro arguments. Converts the following expression (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 % operator allows a macro call by value.

The expression following the % must be 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 substitutes the string (SYM1 + SYM2) for the dummy N. The result is:

```
%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 let you insert page breaks and page headers. Listing control directives turn on and off the listing of any part of the assembled file (or the entire file).

PAGE

PAGE [<length>] [,<width>]
PAGE [+]

where:

<length> is the new page length (measured in lines per page). The length must be in the range 10 to 255. The default page length is 50 lines per page.

<width> is the new page width (measured in characters). It must be in the range 60 to 132. The default page width is 80 characters.

[+] tells MACRO-86 that there is a major and minor page number and resets the minor page number to 1. (In the page number 2-1, for example, the 2 is the major page number and the 1 is the minor page number.) If the plus sign is not present, only the minor portion of the page number is incremented.

If used without an argument or with the optional [, +] argument, PAGE tells MACRO-86 to start a new output page. MACRO-86 puts a form-feed character in the listing file at the end of the page.

If used with the length or width argument, PAGE does not start a new listing page.

Example:

```
        .
        .
        .
PAGE +    ;increment Major, set minor to 1
        .
        .
        .
PAGE 58, 60    ;page length = 58 lines,
               ;width = 60 characters
```

TITLE

TITLE <text>

TITLE sets the title listed on the first line of each page. The text can be up to 60 characters long. If you give more than one TITLE, 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 appears at the top of every page of the listing.

SUBTITLE

SUBTTL<text>

SUBTTL sets the subtitle listed in each page heading on the line below the title. The text is truncated after 60 characters.

You can use any number of subtitles in a program. Each time MACRO-86 encounters a SUBTTL directive, it replaces the text of the previous subtitle with that of the latest subtitle. If you want to turn off SUBTTL for part of the output, enter SUBTTL followed by a null string.

Example:

```
SUBTTL SPECIAL I/O ROUTINE
      .
      .
      .
SUBTTL
      .
      .
      .
```

The first SUBTTL causes the subtitle "SPECIAL I/O ROUTINE" to print at the top of every page. The second SUBTTL turns off the subtitle (the subtitle line is left blank).

%OUT

%OUT< text >

The text is listed on the terminal during assembly. Use %OUT when you want to display progress through a long assembly or see the value of conditional assembly switches.

%OUT outputs on both passes. If you want only one printout, use the IF1 or IF2 directive, depending on the pass you want to see.

When MACRO-86 encounters the following:

```
%OUT *Assembly half done*
```

this message is sent to the screen:

```
%OUT*Pass 1 started*
ENDIF
IF2
%OUT*Pass 2 started*
ENDIF
```

.LIST, .XLIST

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

If you specify a listing file after the Listing prompt, a listing file including all source statements is listed.

An .XLIST overrides all other listing directives. When .XLIST is encountered in the source file, source and object code are not listed. .XLIST stays in effect until a .LIST is encountered.

5

Example:

```
    .
    .
    .
.XLIST    ;listing suspended here
    .
    .
    .
.LIST     ;listing resumes here
```

There are several other associated directives that you can use to control a listing.

- .SFCOND: Suppresses any part of the listing containing conditional expressions that evaluate as false.
- .LFCOND: Ensures the listing of conditional expressions that evaluate false. This is the default condition.
- .TFCOND: Operates independently of .LFCOND and .SFCOND to toggle the current setting. The default setting is set by the presence or

absence of the /X switch when running MACRO-86. When /X is used, .TFCOND causes false conditionals to list. When /X is not used, .TFCOND suppresses false conditionals.

- .XALL: The default. It lists source code and object code produced by a macro. Source lines which do not generate code are not listed.
- .LALL: Lists the complete macro text for all expansions, including lines that do not generate code. Comments preceded by two semicolons (;;) are not listed.
- .SALL: Suppresses listing of all text and object code produced by macros.

.CREF, .XCREF

.CREF
.XCREF [<variable list>]

The .CREF directive is the default condition. .CREF remains in effect until MACRO-86 encounters .XCREF.

Used without an argument, .XCREF turns off the .CREF (default) directive. It remains in effect until MACRO-86 encounters another .CREF. Use .XCREF to suppress 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 after .XCREF, they don't appear in the listing or cross-reference file. No other cross-referencing is affected. (Separate the variables with commas.)

When used without arguments, neither .CREF nor .XCREF takes effect unless you specify a cross-reference file when running MACRO-86. If you use .XCREF <variable list>, it suppresses the variables from the symbol table listing regardless of whether a cross-reference file is being created.

Example:

```
.XCREF CURSOR,FOO,GOO,BAZ,ZOO ;these variables won't
                               ;appear in the listing
                               ;or cross-reference file
```

6

ASSEMBLING A MACRO-86 SOURCE FILE

There are two types of commands used when assembling with MACRO-86: a command that invokes MACRO-86 and others that result from your answers to command prompts. In addition, there are three switches that control alternate MACRO-86 features and a set of command characters that help you enter assembler commands.

Usually, you'll enter all MACRO-86 commands at the keyboard. As an option, answers to the command prompts and any switches can be put into a batch file.

INVOKING MACRO-86 6.1

MACRO-86 is invoked in two ways. With the first method, you enter commands as answers to individual prompts. With the second method, you enter all commands on the line used to invoke MACRO-86.

METHOD 1: MASM

Enter:

```
MASM
```

MACRO-86 is loaded into memory and a series of four text prompts appear on your screen one at a time. Your answers to the prompts tell MACRO-86 to perform specific tasks.

At the end of each line, you can enter one or more switches, each of which must be preceded by a slash mark. If you don't enter a switch, MACRO-86 does not carry out the function controlled by that switch.

The command prompts are summarized here and described in more detail in Section 6.2. Following the summary of prompts is a summary of switches. These are described in Section 6.3.

Exhibit 6a: MACRO-86 Command Prompts

PROMPT	RESPONSES
Source filename [.ASM]:	Lists .ASM file to be assembled. No default value; you must supply a file name.
Object filename [source.OBJ]:	Lists file name for relocatable object code. Default: source filename .OBJ.
Source listing [NUL.LST]:	Lists file name for listing. Default: no listing file.
Cross reference [NUL.CRF]:	Lists file name for cross-reference file (used with MS-CREF to create a cross-reference listing). Default: no cross-reference file.

Exhibit 6b: MACRO-86 Command Switches

SWITCH	ACTION
/D	Produces a listing on both assembler passes.
/O	Shows generated object code and offsets in octal radix on listing.
/X	Suppresses the listing of false conditionals. Also used with the .TFCOND directive.

Command Characters

MACRO-86 has two command characters.

; This can be used any time after you respond to the first prompt. Use a semicolon followed immediately by a carriage return to

select default responses to the remaining prompts. This saves time and ends the need to enter a series of carriage returns.

Do not use the semicolon to skip over some prompts and not others. Once the semicolon has been entered, you can no longer respond to any of the prompts for that assembly. Use the Return key to skip single prompts.

Example:

Source filename [.ASM]: FUN ↵

Object filename [FUN.OBJ]: ; ↵

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

You get the same result by entering:

Source filename [.ASM]: FUN; ↵

Alt-C Alt-C aborts the assembly at any time. If you make an incorrect response, enter Alt-C to exit MACRO-86. Then reinvoke MACRO-86 and start over.

METHOD 2: MASM <filenames>[/switches]

Enter:

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

where:

<source> is the name of the source file.

<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 placed following any of the response entries (just before any of the commas or after <cross-ref>, as shown).

MACRO-86 is loaded into memory and 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.

To select the default for a field, enter a second comma without space in between. For example:

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

loads MACRO-86 and causes the source file FUN.ASM to be assembled. MACRO-86 then outputs the relocatable object code to a file named FUN.OBJ (a default caused by the two commas in a row). Then, it 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 are not given for listing and cross-reference files, those files are not created. If listing file switches are given without a file name, the switches are ignored.

6.2 MACRO-86 COMMAND PROMPTS

MACRO-86 is commanded by the responses you give to four text prompts. These ask you for the names of source, object, listing, and cross-reference files. Each time you respond to a prompt, the next one appears. When the last prompt has been answered, MACRO-86 begins assembly automatically. When it finishes assembly, MACRO-86 exits to the operating system. When you see the operating system prompt, you know that MACRO-86 has finished successfully. If the assembly is not successful, MACRO-86 returns an appropriate error message.

All command prompts accept a file specification as a response. You can enter:

- A file name only.
- A device designation only.

- A file name and an extension.
- A device designation and file name.
- A device designation, file name, and extension.

You cannot enter only a filename extension.

COMMAND PROMPT DESCRIPTIONS

Source filename [.ASM]:

Enter the file name of your source program. MACRO-86 assumes that its extension is .ASM, as shown in square brackets in the prompt text. If your source program has another extension, you must enter it along with the file name. Otherwise, the extension can be omitted.

Object filename [source.OBJ]:

Enter the name of the file to receive the generated object code. If you press Return, the object file is given the same name as the source file, but with the extension .OBJ. If you want a different name or extension, you must enter your choice(s). If you want to change only the file name, enter the file name only. To change the extension only, you must enter both the file name and the extension.

Source listing [NUL.LST]:

Enter the name of the file you want to receive the source listing. If you press Return, MACRO-86 does not produce a listing file. If you enter a file name only, the listing is created with the name you chose and extension .LST. You can also enter your own extension.

The source listing file lists all the statements in your source program and shows the code and offsets generated for each statement. The listing also shows any error messages generated during the session.

Cross reference [NUL.CRF]:

Enter the name of the file to receive the cross-reference file. If you press Return, MACRO-86 does not produce a cross-reference file. If you enter a file

name only, the cross-reference file is created with the name you chose and the extension .CRF. You can also choose your own extension.

The cross-reference file is the source file for the MS-CREF cross-reference utility. MS-CREF converts this file into a cross-reference listing that you can use 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 each occurrence of every symbol in your program. The occurrence that defines the symbol is marked.

6.3 MACRO-86 COMMAND SWITCHES

Three switches control alternate assembler functions. They must be entered at the end of a prompt response, regardless of the method used to invoke MACRO-86. You can group switches at the end of a single response or they can be scattered at the end of several. If you enter more than one switch at the end of a response, each switch must be preceded by the slash mark (/). You cannot enter a switch by itself in response to a command prompt.

These switches are available when using the assembler:

/D Produces a source listing on both assembler passes. When compared, the listings show where errors occur and, possibly, give you a clue as to why the errors occur. The /D switch does not take effect until you tell MACRO-86 to create a source listing.

/O Outputs the listing file in octal radix. The generated code and the offsets shown on the listing are all given in octal. The actual code in the object file will be the same as if the /O switch were not given. The /O switch affects only the listing file.

/X Suppresses the listing of false conditionals. If your program contains conditional blocks, the listing file shows 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 these blocks from your listing.

The /X switch does not affect any block of code that is controlled by the .SFCOND or .LFCOND directives.

If your source program contains the .TFCOND directive, the /X switch has the opposite effect. Normally, the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND suppresses false conditionals; the second restores listing of false conditionals, and so on. When you use /X, false conditionals are suppressed. When MACRO-86 reaches the first .TFCOND directive, listing of false conditionals is restored. When MACRO-86 reaches the second .TFCOND, false conditionals are again suppressed from the listing.

Of course, the /X switch has no effect if a listing is not created. See additional discussion under the .TFCOND directive in Chapter 5.

The following chart shows the effects of the conditional listing directives when combined with the /X switch.

Exhibit 6c: Combining Conditional Listing Directives with the /X Switch

DIRECTIVE	/X OFF	/X ON
—	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

FORMATS OF LISTINGS AND SYMBOL TABLES 6.4

The source listing produced by MACRO-86 (created when you use a file name 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.
- The 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 contains your source program file with the line numbers, offsets, and generated code. Where applicable, it also contains plus signs to show that the source statements are part of a macro block; or a C to show that the source statements come from a file input by the INCLUDE directive. If errors occur during assembly, an error message is printed directly below the statement where the error occurred.

Following this section you'll see part of a listing file, with notes explaining what the various entries represent. The comments have been moved down one line because of format restrictions. If you print your listing on 132-column paper, the comments shown here will easily fit on the same line as the rest of the statement.

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

Summary of listing symbols:

R	Linker resolves entry to left of R.
E	External.
—	Segment name, group name, or segment variable used in MOV: AX,↔ , DD ↔ , JMP ↔ , and so on.
=	Statement has an EQU or = directive.
nn:	Statement contains a segment override.
nn/	REPxx or LOCK prefix instruction.
[xx]	DUP expression; xx is the value in parentheses following DUP. For example, DUP(?) puts ?? where xx is shown here.
+	Line comes from a macro expansion.
C	Line comes from file named in INCLUDE directive statement.

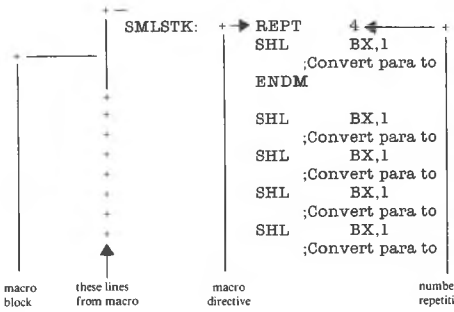
ENTX PASCAL entry for initializing programs

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

                                                              ;Base of heap before init
done
0000    14 [                                DB      20 DUP(?)
                                ?? shows value in parentheses
                                ]
                                + Indicates DUP expression

= 0014                          SKTOP       EQU     THIS BYTE
0014                          STACK         ENDS


0000                            MAINSTARTUP SEGMENT   'MEMORY'
                                DGROUP      GROUP     DATA,STACK,CONST,HEAP,MEMORY
                                            ASSUME    CS:MAINSTARTUP,DS:DGROUP,
                                                      ES:DGROUP,SS:DGROUP
                                            PUBLIC    BEGXQQ ;Main entry

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

0005  8C 06 0022 R                          MOV       CESXQQ,ES
offset  generated code          name        action    expression   comment

                                                      .
                                                      .
                                                      .
000C  26: 8B 1E 0002                        MOV       BX,ES:2     ;Highest paragraph
          segment override
```

6

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

0024  8B  E3                            MOV     SP,BX
                                        ;Set stack to top of memory
                                        .
                                        .
                                        .
0069  EA  0000  ---- R                  JMP     FAR PTR STARTmain

006E                      BEGXQQ        ENDP

                                        .
                                        .
                                        .
007E                      MAINSTARTUP   ENDS

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



+ -signal to linker

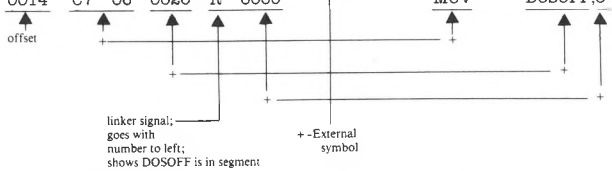
+ -segment variable

+ linker resolves: indicates segment name, group name, or segment variable used in MOV AX, <---->; DD <---->; JMP <---->, etc. (See other examples in this listing.)

6

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
```




```
00  2E  0020  R                    JMP     DWORD PTR DOSOFF
                                               ;return to DOS
001E                            STARTmain  ENDP
                                :
                                :
0037                            ENTXCM     ENDS
                                           END     BEGXQQ
```

6

Differences Between Pass 1 Listing and Pass 2 Listing

If you give the /D switch when you use MACRO-86 to assemble a file, the assembler produces a listing for both passes. This option is especially helpful when looking for the cause 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 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 — a jump of 3 bytes.

The same amount of code was produced during both passes, so there was no phase error. The only difference is in the contents, not in the size.

SYMBOL TABLE FORMAT

The symbol table portion of a listing separates all symbols into their respective categories, with appropriate descriptive data. This data gives you an idea how your program is using various symbolic values. Use this information to help you debug.

You can also use a cross-reference listing (produced by MS-CREF) to help you locate uses of the various symbols in your program.

A complete symbol table listing is on the next page. Following the complete listing, sections from different symbol tables are shown with explanatory notes.

This rule applies to all sections of symbol tables: If your program doesn't contain symbolic values for a particular category, the heading for that category is left out of the symbol table listing. If you don't use macros in your program, you won't see a macro section in the symbol table.

Macros:

Name	Length
BIOSCALL	0002
DISPLAY	0005
DOSCALL	0002
KEYBOARD	0003
LOCATE	0003
SCROLL	0004

Structures and records:

Name	Width / Shift	# fields / Width	Mask	Initial
PARMLIST	001C	0004		
BUFSIZE	0000			
NAMESIZE	0001			
NAMETEXT	0002			
TERMINATOR	001B			

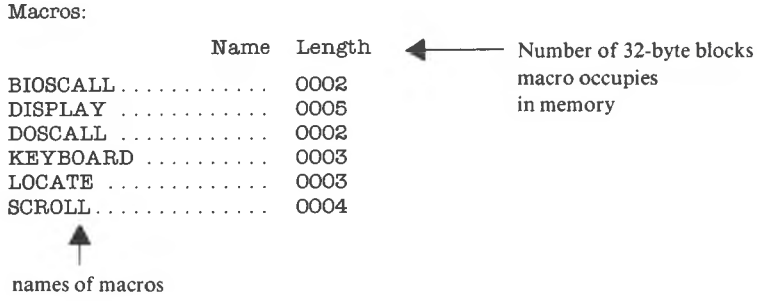
Segments and groups:

Name	Size	align	combine	class
CSEG	0044	PARA	PUBLIC	'CODE'
STACK	0200	PARA	STACK	'STACK'
WORKAREA	0031	PARA	PUBLIC	'DATA'

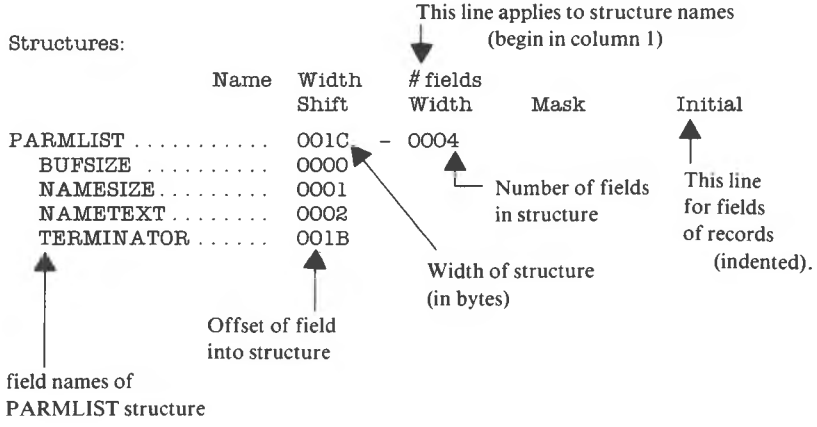
Symbols:

Name	type	Value	Attr	
CLS	N PROC	0036	CSEG	Length =000E
MAXCHAR	Number	0019		
MESSG	L BYTE	001C	WORKAREA	
PARMS	L 001C	0000	WORKAREA	
RECEIVR	L FAR	0000		External
START	F PROC	0000	CSEG	Length =0036

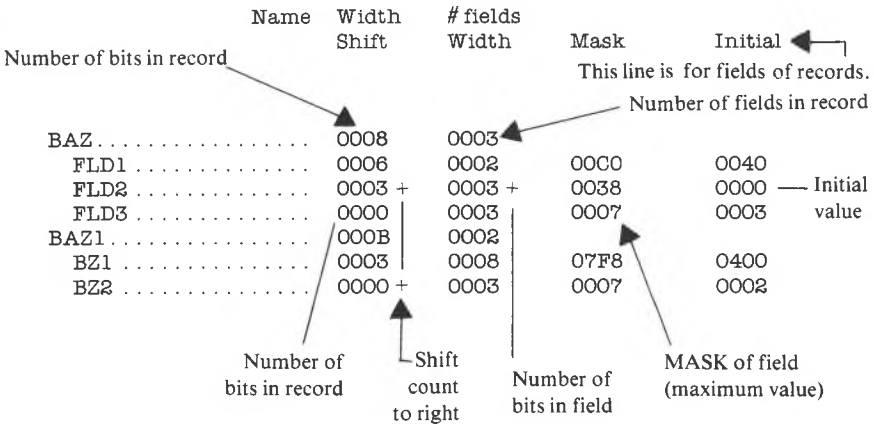
Warning Errors	Severe Errors
0	0



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 × 32 bytes =) 160 bytes long.



Records:



The preceding 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.

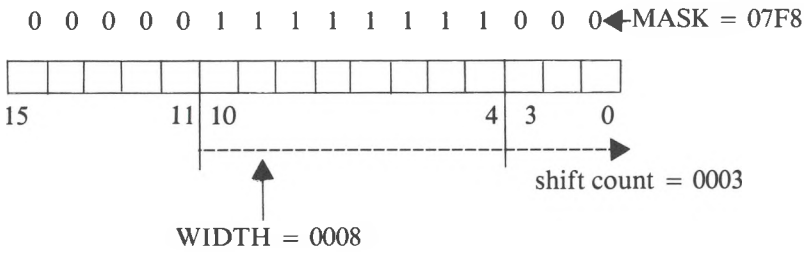
For structures, Width (upper line) shows the number of bytes your structure occupies in memory. # fields shows how many fields are in your structure.

For records, Width (upper line) shows the number of bits the record occupies. # fields shows how many fields are in your record.

For fields of structures, Shift shows the number of bytes the fields is offset into the structure. The other columns are not used.

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

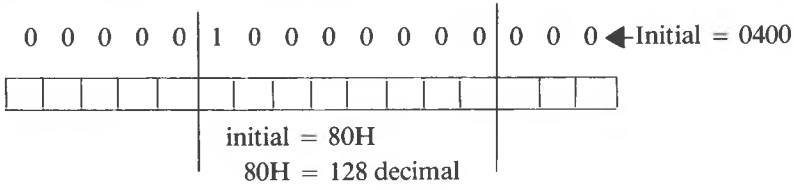
Using field BZ1 of the record BAZ1 above to illustrate:

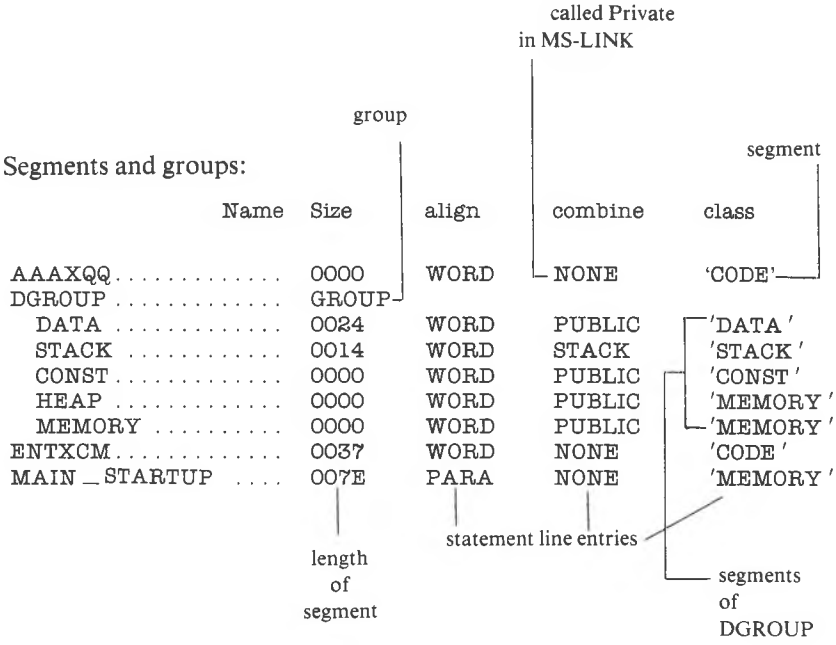


Initial shows the value specified as the initial value for the field, if any.

When naming the field, you specified: fieldname:# = value

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




called Private
in MS-LINK
group
segment
Segments and groups:
Name Size align combine class
AAAXQQ 0000 WORD NONE 'CODE'
DGROUP GROUP
DATA 0024 WORD PUBLIC 'DATA'
STACK 0014 WORD STACK 'STACK'
CONST 0000 WORD PUBLIC 'CONST'
HEAP 0000 WORD PUBLIC 'MEMORY'
MEMORY 0000 WORD PUBLIC 'MEMORY'
ENTXCM 0037 WORD NONE 'CODE'
MAIN_STARTUP 007E PARA NONE 'MEMORY'
length
of
segment
statement line entries
segments
of
DGROUP

6

For groups, the name of the group appears in 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 appear in column 1 (as here) if you do not declare them part of a group. If you do declare a group, the segment names 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 For example:

  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 MS-LINK combines the various segments.
- Class is the class name under which MS-LINK combines segments in memory.

Symbols:

Name	Type	Value	Attr
F00	Number	0005	
F001	Text	1.234	
F002	Number	0008	
F003	Alias	F00	
F004	Text	5[BP][DI]	
F005	Opcode		

all formed by EQU or = directive

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
CRCXQQ . . . . . .      L WORD    001C    DATA      Global
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
HEAPLOW . . . . . .     BYTE      0000    HEAP
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
```

length of PROC

+ EQU statements

+ showing segment

+ – If MACRO-86 knows this length as one of the type lengths (BYTE, WORD, DWORD, QWORD, TBYTE), it shows that type name here.

The preceding section lists other symbolic values in your program that do not fit 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 can be combined to form the various types shown in the example.

For all procedures, the length of the procedure is given after its attribute (segment). You may also see an entry under type like:

```
L 0031
```

This entry results from code such as:

```
BAZ LABEL FOO
```

where FOO is a STRUC that is 31 bytes long. BAZ is shown in the symbol table with the L 0031 entry. Basically, Number (and some other similar entries) indicates that the symbol was defined by an EQU or = directive. Value (usually) shows the numeric value the symbol represents. (In some cases, the Value column will show some text — when the symbol was defined by EQU or = directive.) Attr always shows the segment of the symbol, if known. Otherwise, the Attr column is blank.

Following the segment name, the table shows either External, Global or a blank (which means not declared with either the EXTRN or PUBLIC directive). The last entry applies to PROC types only. This is a length = entry, which is the length of the procedure.

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

Each of the four types shows a value as follows:

- Number shows a constant numeric value.
- Opcode shows a blank. The symbol is an alias for an instruction mnemonic. Sample directive statement:

```
FOO EQU ADD
```

- Alias shows a symbol name equal to the named symbol. Sample directive statement:

```
FOO EQU BAX
```

- Text shows the "text" the symbol represents. "Text" is any EQU directive operand that does not fit one of the three categories above. Sample directive statements:

```
GOO EQU 'WOW'
BAZ EQU DS:8[BX]
ZOO EQU 1.234
```

7

MACRO-86 MESSAGES

MACRO-86 outputs two kinds of messages. Most are error messages. These messages are classified as assembler errors, I/O handler errors and run-time errors. The non-error messages output by MACRO-86 are the banner displayed when MACRO-86 is first invoked, the command prompt messages, and the end of (successful) assembly message. These non-error messages are classified here as operating messages.

OPERATING MESSAGES 7.1

Banner Message and Command Prompts:

```
MACRO-86 v1.0 Copyright (C) Microsoft, Inc.

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

End of Assembly Message:

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

7.2 ERROR MESSAGES

MACRO-86 outputs error messages when it encounters errors. It tells you the numbers of warning and fatal errors and returns control to your operating system. The error message is sent either to your screen or to the listing file (if you have created one).

In the following listing, error messages are divided into three categories: assembler errors, I/O handler errors, and run-time errors. In each category, messages are listed in alphabetical order, along with a short explanation when necessary. At the end of this chapter, the error messages are listed in a single numerical order list without explanations.

ASSEMBLER ERRORS

Already defined locally (Code 23)

You tried to define a symbol as external when it had already been defined locally.

Already had ELSE clause (Code 7)

You attempted to define an ELSE clause within an existing ELSE clause. (You cannot nest ELSE without nesting IF...ENDIF.)

Already have base register (Code 46)

You tried to double base register.

Already have index register (Code 47)

You tried to double index address.

Block nesting error (Code 0)

You have not properly terminated nested procedures, segments, structures, macros, IRC, IRP, or REPT. An example of this error is closing an outer level of nesting when inner level(s) are still open.

Byte register is illegal (Code 58)

You've used one of the byte registers in context where it is illegal. For example: PUSH AL.

Can't override ES segment (Code 67)

You've tried 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)

You attempted to use EVEN on a segment that was declared to be byte segment.

Circular chain of EQU aliases (Code 83)

An alias EQU eventually points to itself.

Constant was expected (Code 42)

MACRO-86 expected a constant and received something else.

CS register illegal usage (Code 59)

You're trying to use the CS register illegally. For example: XCHG CS,AX.

Directive illegal in STRUC (Code 78)

All statements within STRUC blocks must be Define directives or comments preceded by a semicolon (;).

Division by 0 or overflow (Code 29)

An expression is given that results in a divide by 0.

DUP is too large for linker (Code 74)

Nesting of DUPs was such that the record created was too large for the linker.

Extra characters on line (Code 1)

This occurs when MACRO-86 has not received enough information on a line to define the instruction directive. Superfluous characters are received.

Field cannot be overridden (Code 80)

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Forward needs override (Code 71)

This message not currently used.

Forward reference is illegal (Code 17)

You've attempted to forward reference something that must be defined in the first pass.

Illegal register value (Code 55)

The register value specified does not fit into the register field (the field takes values up to 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)

You've used an external in some illegal manner. For example: DB M DUP (?) where M is declared external.

Illegal use of register (Code 49)

You've used a register with an instruction where there is no 8086 or 8088 instruction possible.

Illegal value for DUP count (Code 72)

DUP counts must be a non-negative constant other than 0.

Improper operand type (Code 52)

You've used an operand in a way such that an opcode cannot be generated.

Improper use of segment reg (Code 61)

You've specified a segment register where this is illegal. For example: an immediate move to a segment register.

Index displ. must be constant (Code 54)

Label can't have seg. override (Code 65)

Illegal use of segment override.

Left operand must have segment (Code 38)

You've used something in the right operand that required a segment in the left operand. (A colon, for example.)

More values than defined with (Code 76)

Too many fields are given in REC or STRUC allocation.

Must be associated with code (Code 45)

You've used a data-related item where a code item was expected.

Must be associated with data (Code 44)

You've used a code-related item where a data-related item was expected. For example: MOV AX,<code-label>.

Must be AX or AL (Code 60)

You've specified a register other than AX or AL where only these are acceptable.

Must be index or base register (Code 48)

The instruction requires a base or index register. Some other register was specified in square brackets.

Must be declared in pass 1 (Code 13)

MACRO-86 was expecting a constant value but got something else. An example of this is using a vector size as a forward reference.

Must be in segment block (Code 69)

You've attempted to generate code when not in a segment.

Must be record field name (Code 33)

MACRO-86 was expecting a record field name but got something else.

Must be record or field name (Code 34)

MACRO-86 was expecting a record name or field name and received something else.

Must be register (Code 18)

Register unexpected as operand but user furnished symbol — was not a register.

Must be segment or group (Code 20)

MACRO-86 expected a segment or group but something else was specified.

Must be structure field name (Code 37)

MACRO-86 expected a structure field name but received something else.

Must be symbol type (Code 22)

MACRO-86 needed WORD, DW, QW, BYTE, or TB but received something else.

Must be var, label or constant (Code 36)

MACRO-86 expected a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)

You have used a prefix instruction without specifying an opcode.

Near JMP/CALL to different CS (Code 64)

You have attempted to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)

No immediate mode has been specified, or you've specified an opcode that cannot accept the immediate. For example: PUSH.

No or unreachable CS (Code 62)

You've tried to jump to a label that is unreachable.

Normal type operand expected (Code 41)

MACRO-86 received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)

You've specified an ENDIF or ELSE 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)

You've attempted to use STRUC name without angle brackets.

Operand combination illegal (Code 63)

You've specified a two-operand instruction where that combination 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)

MACRO-86 expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)

Operand cannot be accessed because it is not in the current IP segment.

Operand types must match (Code 31)

MACRO-86 gets arguments of different kinds or sizes in a case where the arguments must match. For example: MOV.

Operand was expected (Code 27)

MACRO-86 is expecting an operand but has received an operator.

Operator was expected (Code 28)

MACRO-86 was expecting an operator but received an operand.

Override is of wrong type (Code 81)

You've tried to use the wrong size on an override in a STRUC initialization statement. For example: 'HELLO' for DW field.

Override with DUP is illegal (Code 79)

You've tried to use DUP in an override in a STRUC initialization statement.

Phase error between passes (Code 6)

MACRO-86 has received ambiguous instruction directives. These caused the location of a label to change in value between the first and second assembler passes. One example of this is a forward reference coded without a segment override where one is required. There is an additional byte (the code segment override) generated in the second pass, causing the next label to change. You can use the /D switch to produce a listing to help you resolve phase errors between passes (see Section 5.3).

Redefinition of symbol (Code 4)

This error occurs on the second pass and on 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 occurs only if MACRO-86 has internal logic errors. Report this problem to your dealer.

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. You've tried to jump beyond this range.

Segment parameters are changed (Code 24)

The list of SEGMENT arguments is not identical to that at the first time this segment was used.

Shift count is negative (Code 30)

The shift expression generated results in a negative shift count.

Should have been group name (Code 12)

MACRO-86 expects a group name but something else was given.

Symbol already different kind (Code 15)

You've attempted to define a symbol differently than in a previous definition.

Symbol already external (Code 73)

You've attempted define a symbol as local when it has already been defined as external.

Symbol has no segment (Code 21)

You're trying to use a variable with SEG when the variable has no known segment.

Symbol is multi-defined (Code 5)

This error occurs when MACRO-86 encounters a symbol that is later redefined.

Symbol is reserved word (Code 16)

You've attempted to use an assembler reserved word illegally. (For example: declaring MOV as a variable.)

Symbol not defined (Code 9)

You've used a symbol 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 has an unacceptable size.

Unknown symbol type (Code 3)

A symbol statement has something in its type field that MACRO-86 cannot recognize.

Usage of ? (indeterminate) bad (Code 75)

You've used "?" incorrectly. For example: ? + 5.

Value is out of range (Code 50)

Value is too large for the expected use. For example: MOV AL,5000.

Wrong type of register (Code 19)

The 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. They have a different format from that used by other error messages.

Assembler Errors

The general format for assembler errors is:

MASM Error — error-message-text
in: filename

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)

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)

Run-Time Errors

These messages are displayed while your assembled program is being executed.

Internal Error

Usually caused by an arithmetic check. Notify your dealer if this error occurs.

Out of Memory

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

7.3 NUMERICAL 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

CODE	MESSAGE
36	Must be var, label or constant
37	Must be structure field name
38	Left operand must have segment
39	One operand must be const
40	Operands must be same or 1 abs
41	Normal type operand expected
42	Constant was expected
43	Operand must have segment
44	Must be associated with data
45	Must be associated with code
46	Already have base register
47	Already have index register
48	Must be index or base register
49	Illegal use of register
50	Value is out of range
51	Operand not in IP segment
52	Improper operand type
53	Relative jump out of range
54	Index displ. must be constant
55	Illegal register value
56	No immediate mode
57	Illegal size for item
58	Byte register is illegal
59	CS register illegal usage
60	Must be AX or AL
61	Improper use of segment reg
62	No or unreachable CS
63	Operand combination illegal
64	Near JMP/CALL to different CS
65	Label can't have seg. override
66	Must have opcode after prefix
67	Can't override ES segment
68	Can't reach with segment reg
69	Must be in segment block
70	Can't use EVEN on BYTE segment
71	Forward needs override
72	Illegal value for DUP count
73	Symbol already external

CODE	MESSAGE
74	DUP is too large for linker
75	Usage of ? (indeterminate) bad (Code 75)
76	More values than defined with
77	Only initialize list legal
78	Directive illegal in STRUC
79	Override with DUP is illegal
80	Field cannot be overridden
81	Override is of wrong type
82	Register can't be forward ref
83	Circular chain of EQU aliases
101	Hard data
102	Device name
103	Operation
104	File system
105	Device offline
106	Lost file
107	File name
108	Device full
109	Unknown device
110	File not found
111	Protected file
112	File in use
113	File not open
114	Data format
115	Line too long

SYSELECT

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing February, 1983.

ISBN 0-88182-017-2 Printed in U.S.A.

CONTENTS

1. Operating System Generation 1-1
1.1 Introduction 1-1
1.2 Diskette Space 1-1
1.3 Using SYSELECT 1-2
1.4 Selection Menus 1-3
Character Set Selection 1-3
Alternate Character Set 1-3
Keyboard Selection Set 1-3
Primary Printer Selection 1-4
Secondary Printer 1-4
Serial Port Configuration 1-4
Logo Selection 1-4
Banner Skeleton Selection 1-5
Current Configuration 1-5
Writing the Operating System Out 1-5

2. System Operation 2-1
2.1 SYSELECT Batch Files 2-1
2.2 System Selection Files 2-2
Character Set Files 2-3
Keyboard Table File 2-4
Banner Skeleton File 2-4
Logo Files 2-5
2.3 Files Generated by SYSELECT 2-5
2.4 Instruction Files 2-6

EXHIBITS

2a: System Selection File Extensions 2-3
2b: Information Displayed by Character Set Tables 2-4

CHAPTERS

1. Operating System Generation 1

2. System Operation 2

1

OPERATING SYSTEM GENERATION

INTRODUCTION 1.1

SYSELECT is a system selection program that lets you generate a custom operating system for MS-DOS. Operating system components are configured for International, British, French, Italian and German variations. Configurable system components include character set, alternate character set, keyboard, logo and banner selection. User-configured I/O components include primary printer, secondary printer and serial communications port.

SYSELECT produces three intermediate program files that describe the system being generated. The operating system is generated by BIN2REL.EXE and LINK.EXE. BIN2REL.EXE converts binary image files to relocatable Intel object module format files. LINK.EXE combines component system files into an operating system. SYSLOC.EXE reformats the output of LINK.EXE to the appropriate form for SYSCOPY.EXE. SYSCOPY.EXE writes the operating system to diskette or hard disk.

DISKETTE SPACE 1.2

The configuration diskette contains a large number of files, reducing the available user space. The linker and SYSLOC create the files MSDOS.EXE and MSDOS.BIN which SYSCOPY.EXE copies onto the boot tracks. These files can be deleted before starting another SYSELECT session. Space can also be increased by creating a separate configuration diskette without unneeded .CHR, .KB, and .LGO files.

1.3 USING SYSELECT

1. Before beginning system selection and generation, use DCOPY to duplicate the system generation diskette. Store the original system diskette in a safe place.

1

2. Make sure you have a formatted diskette (or a hard disk volume that has already been set up using HDSETUP) in the destination drive. Place the SYSELECT diskette in drive A (or in the floppy drive of a hard disk system) and press the reset button to boot the computer. MS-DOS signs on and asks you to set the time and date. Then SYSELECT prompts you for the type of system you want to generate.

3. System parameters are selected from menus. The first SYSELECT menu gives you three choices:

   - Generate a New Operating System
   - Modify an Existing Operating System
   - Help — Display Instructions

   The first choice ("Generate a New Operating System") is highlighted. To generate a new system, press the Return key. To modify an existing operating system, press the space bar to advance to that selection. When "Modify an Existing Operating System" is highlighted, press the Return key. To get Help, press the space bar again and then press the Return key when "Help — Display Instructions" is highlighted.

   The "Generate a New Operating System" option takes you through the entire process of creating a new configuration. If you make an error during this process, the configuration acceptance display at the end of the program lets you correct that mistake.

   The "Modify an Existing Operating System" option lets you change an existing configuration. When you select an existing control file for modification, SYSELECT displays the configuration specified by that control file. Any part of that configuration can then be changed without going through an entire reselection process.

SELECTION MENUS 1.4

CHARACTER SET SELECTION

The available character sets are displayed. Each set is described by banner name, display class, descriptive comment and file name.

- The banner name is the name of the character set (including version number) displayed in the banner.

- The display class describes the graphic subset (hex 21 through hex 7E) of the character set and helps you avoid incompatible combinations of character set and keyboards. For example, the International display class defines hex 23 as the crosshatch (#) and the British class defines the same hex as the British monetary sign.

- The file name is the name of the file containing the character set.

ALTERNATE CHARACTER SET

An alternate character set lets application programs display an entirely different character set of 128 or 256 characters. Including an alternate character set in a configuration decreases the available memory by up to 8K bytes.

KEYBOARD SELECTION

This menu lets you select a new keyboard table. (The keyboard table defines the codes generated or keyboard functions done when a key is pressed.) The standard keyboard tables are provided on your SYSELECT diskette; however, it is possible to generate your own keyboard tables using the KEYGEN utility.

The Keyboard Selection menu has the same format as the Character Set Selection menu.

PRIMARY PRINTER SELECTION

The Primary Printer Selection menu sets the name of the default printer of the logical device LST:. The choices available are serial printer (UL1:) or parallel printer (LPT:). You can change these values at run time by using the SETIO utility program.

If a serial printer is selected, SYSELECT displays the menu for serial port configuration. (Refer to the Serial Port Configuration menu for more details.)

SECONDARY PRINTER

You can select a secondary printer as well as a primary printer. If a serial printer is chosen as the secondary printer, the next menu to appear is the Serial Port Configuration menu. The primary and secondary printers cannot both be parallel printers because the system supports only one parallel port.

SERIAL PORT CONFIGURATION

This menu lets you set the baud rate, stop bits and parity of the two serial ports. Information on these settings is contained in the user menu for each device you want to connect to a serial port.

The available baud rate choices are 50, 75, 110, 134.5, 150, 200, 300, 600, 1200, 1800, 2000, 2400, 3600 and 4800. The choices for stop bits are 1, 1.5 and 2. The choices for parity are even, odd and none. (The parity bit can be set by software for transmission; parity is not checked on incoming characters.)

LOGO SELECTION

The logo is a unique set of graphics characters that form the logo display. Normally, the logo is displayed as part of the banner when the operating system is loaded. If you've generated a system without a logo character, then you must select a banner without a logo.

BANNER SKELETON SELECTION

The banner skeleton is a framework that holds the logo and configuration information displayed in the banner. For each configuration, the banner skeleton contains different character set, keyboard, and other information.

CURRENT CONFIGURATION

This menu displays the current configuration. The first selection in the upper box starts the process of writing the intermediate files onto disk. The second choice starts the selection process from the beginning. The items displayed in the lower box are the values of the current configuration. Any of these values can be changed by moving the highlight to the item to be changed and pressing the Return key. The menu for the specified selection is displayed for modification.

After the modification is made, the updated Current Configuration menu is displayed. If you choose "Accept the Current Configuration", SYSELECT displays the menu for writing the operating system out.

WRITING THE OPERATING SYSTEM OUT

This menu displays the current intermediate files. You can select an existing file, or you can enter a new file name by selecting the "User Entered File" option. After you make your choice, you'll be asked to confirm it. If you answer No, control returns to the Current Configuration menu. If you choose to write the operating system to the specified file, SYSELECT does this job (which can take several minutes).

2

SYSTEM OPERATION

This chapter describes the operation of the system selection program, including SYSELECT programs, system selection files and files generated by SYSELECT.

SYSELECT BATCH FILES 2.1

There are four SYSELECT batch files: FL.BAT, HD.BAT, FLIEEE.BAT and HDIEEE.BAT. Each batch file runs the same five programs: SYSELECT.EXE, BIN2REL.EXE, LINK.EXE, SYSLOC.EXE and SYSCOPY.EXE. The difference between the batch files is that each tells the linker to link different object files, depending on the configuration.

For example, if you have a hard disk system and you want to use the IEEE 488 driver in your operating system, then choose the HDIEEE.BAT file. This file tells the linker to link the appropriate object modules for your system. To invoke HDIEEE.BAT, type HDIEEE after the A> prompt.

SYSELECT.EXE

SYSELECT.EXE makes the system component selection. Available options are:

- Generate a new system, or modify an existing one.
- Select a keyboard table.
- Select a display character set.
- Select the printer type.
- Select the serial ports options (baud rate, stop bits, parity).
- Select the logo file.
- Select the banner file.

BIN2REL.EXE

BIN2REL.EXE converts binary image files to relocatable Intel object module format files.

LINK.EXE

LINK.EXE collects the files selected and created by SYSELECT and links them into a single file called MSDOS.EXE.

SYSLOC.EXE

SYSLOC.EXE reformats the MSDOS.EXE file into the proper format for SYSCOPY.EXE.

SYSCOPY.EXE

The SYSCOPY utility makes a diskette or hard disk volume into a bootable diskette or volume. It is also used to replace a system image on a diskette or on a hard disk volume.

In order to boot the system on a hard disk volume, you must use HDSETUP to select that volume as the primary boot volume (in addition to making the volume bootable with SYSCOPY.EXE). If a hard disk volume is the current primary boot volume, the newly copied system is used on the next boot from the hard disk.

Similarly, a diskette must first be formatted in order for SYSCOPY.EXE to work.

2.2 SYSTEM SELECTION FILES

The directory has information on keyboards, character sets, translation tables, banner skeletons and the logo. These files can be found by searching the directory for their file extensions. These extensions are listed in Exhibit 2a.

Exhibit 2a: System Selection File Extensions

FILE TYPE	EXTENSION
Keyboard	.KB
Character set	.CHR
Banner skeleton	.BNR
Translation table	.XLT
Logo	.LGO

SYSELECT expects a particular format for each file type. Errors occur if other file types use any of the extensions in Exhibit 2a, or if the format of a file type is modified.

CHARACTER SET FILES

Files with the .CHR extension contain character set tables. These tables contain data corresponding to the dot matrix displayed by each character on the keyboard, and information on the character set name, version number, origin and date of origin, and display class. Most of this information is displayed by SYSELECT to help you select the correct character set. The format of the information in the first sector of these files is shown in Exhibit 2b.

The banner name and version are the name and version number of the character set placed in the banner. The banner is displayed when the system is booted.

Exhibit 2b: Information Displayed by Character Set Tables

TYPE OF INFORMATION	LENGTH (BYTES)
File type (K = keyboard; C = character)	1
Format version	1
Display class	12
Banner name	8
Filler (a space)	1
Comment	35*
Originator**	16
Date (yy/mm/dd)**	8
Length**	4
Unused**	51

* 31 bytes are displayed by SYSELECT
** Not displayed by SYSELECT

KEYBOARD TABLE FILE

Files with the .KB extension are keyboard table files. These files contain information on the keyboard code sent to the processor when a key is pressed, and information on the keyboard table name, version number, origin, date of origin, and display class displayed by the system selection program.

BANNER SKELETON FILE

Files with the .BAN extension are banner skeleton files. (The banner is information displayed during system boot, including the logo and configuration information.) The banner skeleton is a set of ASCII strings containing the escape sequences and characters needed to display the logo and configuration information.

SYSELECT displays the banner files available for selection. A copy of the banner you specify is made with the names of the chosen keyboards and

character sets placed in the correct locations. The first sector of the banner contains the location of these fields. If the first byte is not zero, SYSELECT does not customize the banner. If data in the first sector is not "recognizable," default locations are used during custom banner generation.

If a custom banner is written, the first sector has this format: The first byte is zero, followed by 0DH 0AH. This is followed by the length of the file in decimal (with a leading and a trailing space), followed by 0DH 0AH.

The location of the keyboard name and character set name follow the same format as the file length. If the file length is 639 characters, the keyboard name is at byte 502, and the character set name is at 541. The first 24 bytes of the banner file are shown below (in hex):


```
00 0D 0A 20 36 33 39 20 0D 0A 20 35 30 32 20 0D 0A 20
35 34 31 20 0D 0A
```

LOGO FILES

Like the character set, the logo contains data that corresponds to a set of special characters. These characters represent the set of dots in the logo. If the size of the logo is nonstandard, the first byte must contain its length in sectors. Sixteen-sector logo files are supported.

FILES GENERATED BY SYSELECT 2.3

*.CTL Files

The primary output of SYSELECT is a control file containing the specifications of the operating system. Existing control files can be modified by using the "Modify an Existing Operating System" option.

*.SPR Files

An *.SPR file is generated for each operating system selected by SYSELECT. This file contains system parameter data to be loaded into the operating system.

*.BNR Files

A *.BNR file (banner file) is generated each time you select an operating system. This file is a customized version of the selected banner skeleton file.

2.4 INSTRUCTION FILES

The file in the SYSELECT.HLP program contains information that tells you how to use SYSELECT.