Z-DOS™

Volume II

593-0051
CONSISTS OF

MANUAL
595-2835
FLYSHEET
597-2869

Printed in the
United States of America

ZENITH data systems

HEATH

NOTICE

This software is licensed (not sold). It is licensed to sublicensees, including end-users, without either express or implied warranties of any kind on an "as is" basis.

The owner and distributors make no express or implied warranties to sublicensees, including end-users, with regard to this software, including merchantability, fitness for any purpose or non-infringement of patents, copyrights or other proprietary rights of others. Neither of them shall have any liability or responsibility to sublicensees, including end-users, for damages of any kind, including special, indirect or consequential damages, arising out of or resulting from any program, services or materials made available hereunder or the use or modification thereof.

Technical consultation is available for any problems you encounter in verifying the proper operation of these products. Sorry, but we are not able to evaluate or assist in the debugging of any programs you may develop. For technical assistance, call:

(616) 982-3884 Application Software/Softstuff Products
(616) 983-3860 Operating System/Language Software/Utilites

Consultation is available from 8:00 AM to 4:30 PM (Eastern Time Zone) on regular business days.

Zenith Data Systems
Software Consultation
Hilltop Road
St. Joseph, Michigan 49085

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS
ST. JOSEPH, MICHIGAN 49085

TABLE OF CONTENTS

PART 4: Appendices and Index

Appendices

Appendix Q Procedure to Change Disk Parameters
Memory Checking

Introduction to MACRO-86

INTRODUCTION

MACRO-86 produces relocatable, linkable code for maximum efficiency in memory assignment, library maintenance, and modular program development.

MACRO-86 fully supports macro assembly, conditional assembly, and an extensive set of assembler directives.

Source code blocks used repeatedly within a program can be entered once as a "macro definition". A one line "macro call" causes the assembler to insert the code at any desired point in the program. A macro call may occur within the definition of another macro. Such nesting is limited only by memory size.

Conditional assembly allows portions of source code to be either assembled or ignored as a result of tests on conditions chosen by the programmer. Conditional statements may be nested to a maximum level of 255.

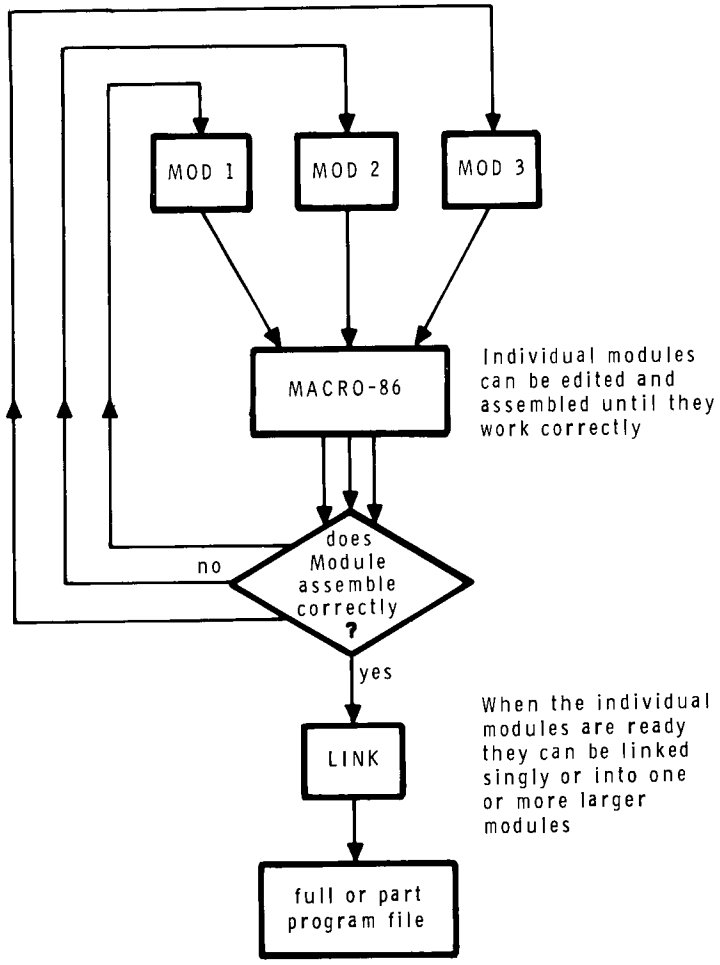
Directives are statements included in the source code to control the functions of the assembler. MACRO-86 has the ability to correct some errors made during source code entry. This feature relies on an evaluation of the programmer's likely intent while making the error. Such corrections are flagged as errors to ensure a recheck.

Features and Benefits of MACRO-86

Zenith's MACRO-86 Assembler is a powerful assembler for 8088 based computers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from an 8088 or 8086 microprocesser.

MACRO-86 produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked using Zenith's LINK utility to produce relocatable, executable object code. Relocatable code can be loaded anywhere in memory. Thus, the program can execute where it is most efficient, not only in some fixed range of memory addresses.

In addition, relocatable code means that programs can be created in modules, each of which can be assembled, tested, and perfected individually. This saves recoding time because testing and assembly is performed on smaller pieces of program code. Also, all modules can be error free before being linked together into larger modules or into the whole program. The program is not a huge monolith of code.



Module Assembly

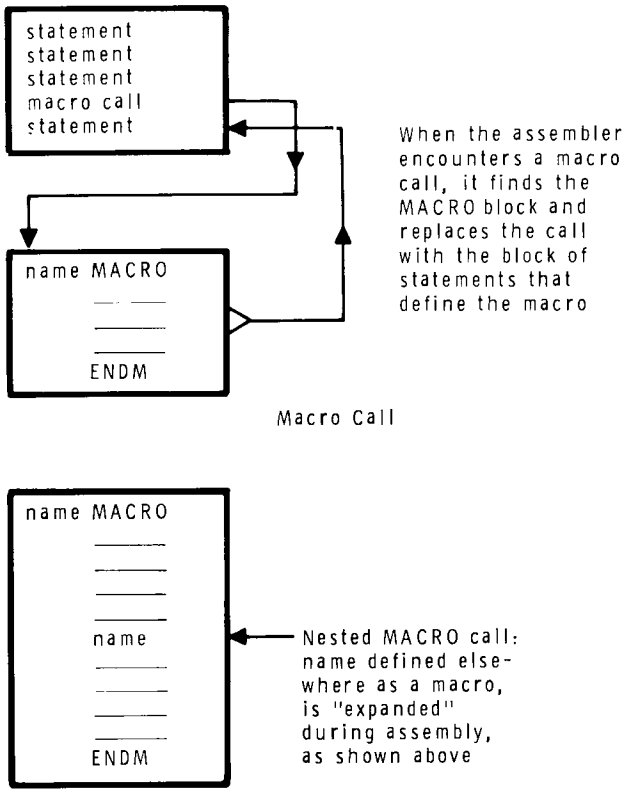
Introduction to MACRO-86

MACRO-86 supports Zenith Data System's complete 8080 macro facility, which is the Intel 8080 standard. The macro facility permits writing blocks of code for a set of instructions used frequently. The need for recoding these instructions each time they are needed is eliminated.

This block of code is given a name: *macro*. The instructions are the *macro definition*. Each time the set of instructions is needed, instead of recoding it, a simple "call" to the macro is placed in the source file. 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 a source module because the macro definitions are given only once, then, other occurrences are one line calls.

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

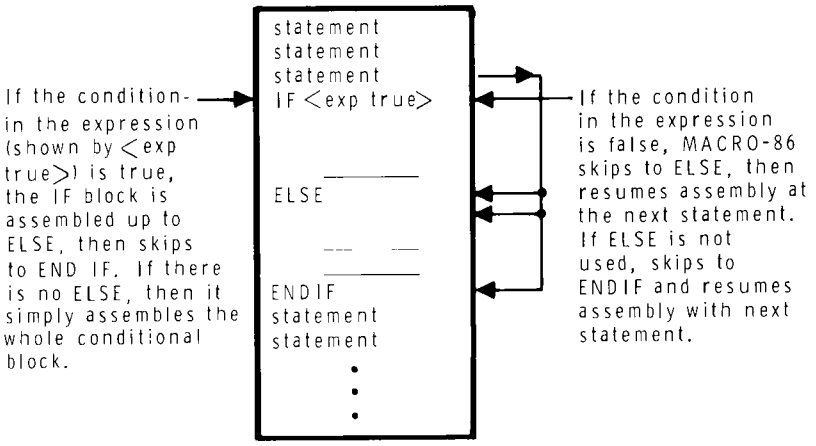
The macro facility includes "repeat", "indefinite repeat", and "indefinite repeat character" directives for programming repeat block operations. The MACRO directive can also be used to alter the action of any instruction or directive by using its name as the macro name. When any instruction or directive statement is placed in the program, MACRO-86 checks first the symbol table it created to see if the instruction or directive is a macro name. If it is, MACRO-86 "expands" the macro call statement by replacing it with the body of instructions in the macro's definition. If the name is not defined as a macro, MACRO-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.



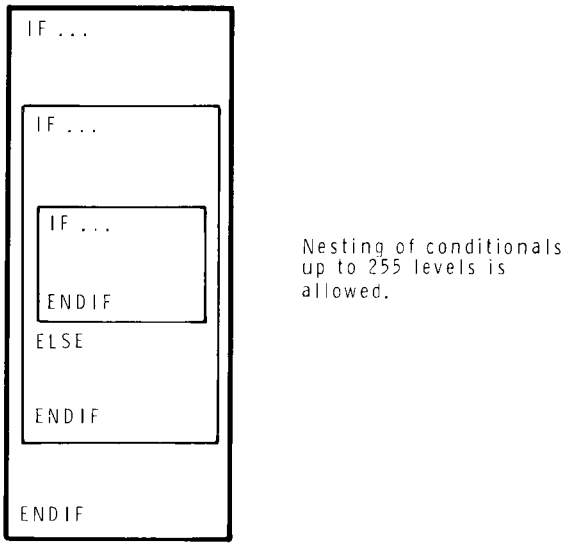
Macro Assembly

MACRO-86 supports an expanded set of conditional directives. Directives for evaluating a variety of assembly conditions can test assembly results and branch where required. Unneeded or unwanted portions of code will be left unassembled. MACRO-86 can test for blank or nonblank arguments, for defined or not-defined symbols, for equivalence, for first assembly pass or second. MACRO-86 can compare strings for identity or difference. The conditional directives simplify the evaluation of assembly results, and make programming the testing code for conditions easier as well as more powerful.

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



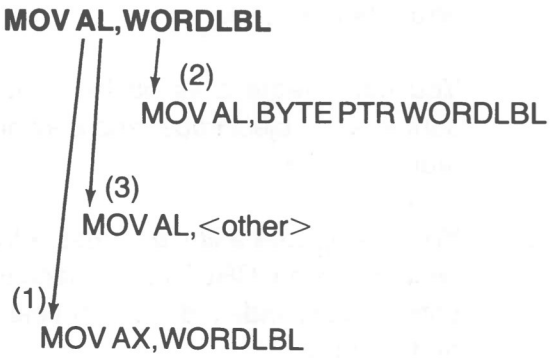
Conditional Assembly



Nesting Conditionals for Assembly

MACRO-86 supports all the major 8080 directives found in Zenith Data System/Heath's MACRO-80 Macro Assembler. This means that any conditional, macro, or repeat blocks programmed under MACRO-80 can be used under MACRO-86. Processor instructions and some directives (e.g., PHASE, CSEG, DSEG) within the blocks, if any, will need to be converted to the 8086 (8088) instruction set. All the major MACRO-80 directives and pseudo-ops, that are supported under MACRO-86 will assemble as is, as long as the expressions to the directives are correct for the processor and the program. The syntax of directives is unchanged. MACRO-86 is upward compatible, with MACRO-80 and with Intel's ASM86, except Intel code macros.

MACRO-86 provides some relaxed typing. Some 8086 (8088) instructions take only one operand. If a typeless operand is entered for an instruction that accepts only one type of operand (e.g., in the instruction PUSH [BX]. [BX] has no size, but PUSH only takes a word), it seems wasteful to return an error for a lapse of memory or a typographical error. When the wrong type choice is given, MACRO-86 returns an error message but generates the "correct" code. That is, it always puts out instructions, not just NOP's. For example, if you enter:



Error Correction During Assembly

MACRO-86 generates instruction two, because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size." MACRO-86 accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some debugging chores. An error message is still returned, however, because you may have misstated an operand that MACRO-86 assumes is "correct."

Overview of MACRO-86 Operation

Brief

The Assembly Process:

1. Create source code file. MACRO-86 expects a default .ASM filename extension.

2. Run MACRO-86 through passes one and two. Error messages are displayed on the terminal. Modify source code as necessary to produce an error-free run. The object filename contains a default .OBJ extension.

3. Use LINK to add the object module to your main program, or LIB to add it to a library.

Options:

1. You may suppress the .OBJ file to speed processing of an error-check run.

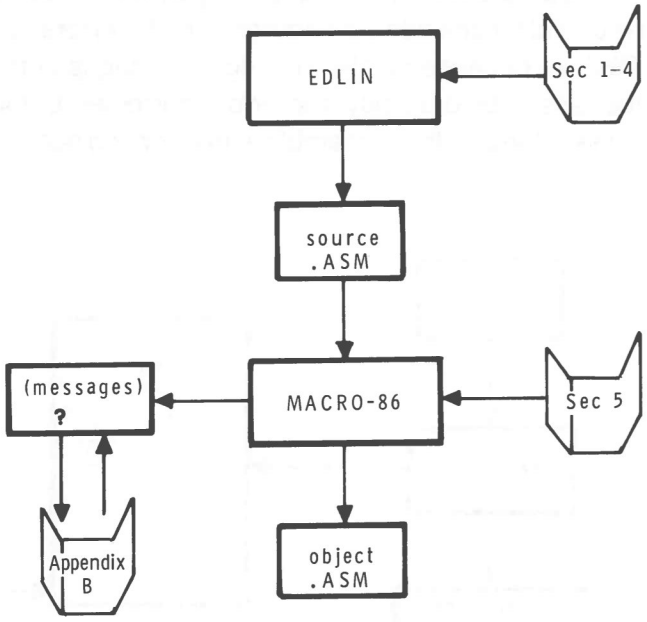
2. You may create a listing file containing relative addresses, source and object code, and a symbol table. It has a .LST default extension.

3. You may create a limited cross reference file with a .CRF default extension. CREF can expand the .CRF file to an .REF file containing an indexed, alphabetical table of all labels, symbols, and variables.

Details

The first task is to create a source file. Use EDLIN (the resident editor in Z-DOS), or other 8088 editors compatible with your operating system, to create the source file. MACRO-86 assumes a default filename extension of .ASM for the source file. Creating the source file involves creating instruction and directive statements that follow the rules and constraints described in the first four sections in this Chapter.

When the source file is ready, run MACRO-86, see Assembling a source File on Page 10.162. Refer to Appendix B, for explanations of any messages displayed during or immediately after assembly.

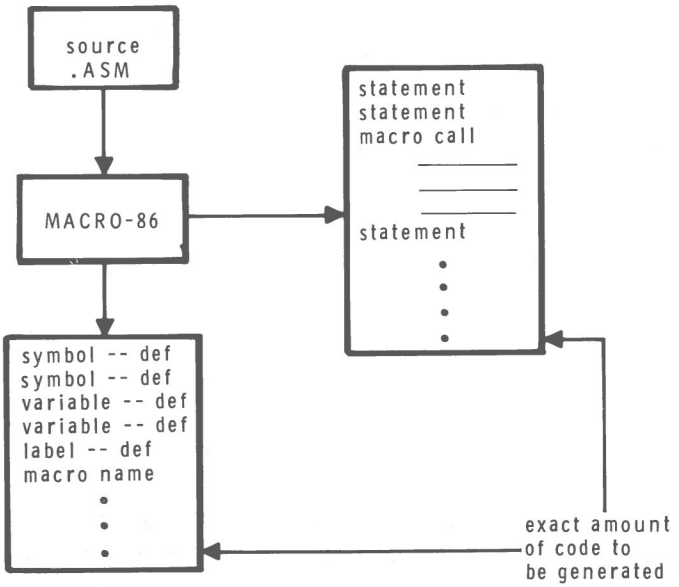


Using This Chapter

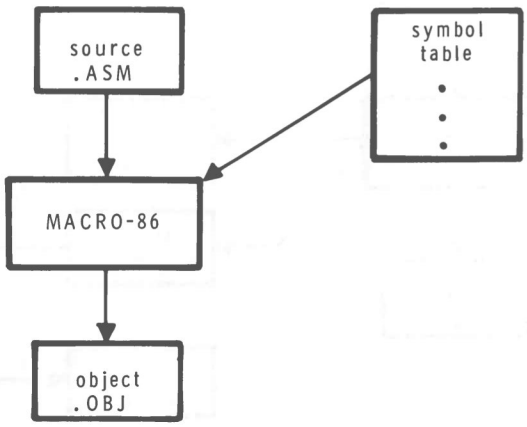
MACRO-86 is a two-pass assembler. This means that the source file is read twice by the assembler. Slightly different actions occur during each pass. During the first pass, the assembler evaluates the statements and expands macro call statements. It calculates the amount of code it will generate, and builds a symbol table where all symbols, variables, labels, and macros are assigned values.

During the second pass, the assembler fills in the symbols, variables, labels, and expression values from the symbol table, expands macro call statements, and sends the relocatable object code into a file with the default filename extension .OBJ. The .OBJ file is suitable for processing with Zenith's LINK utility. The .OBJ file can be stored as part of your library of object programs, which later can be linked with one or more .OBJ modules by LINK. The .OBJ modules can also be processed with Zenith's LIB Library Manager (refer to the LIB Library Manager Chapter, Page 12.1, for further explanation and instructions).

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



Pass One of the Assembly

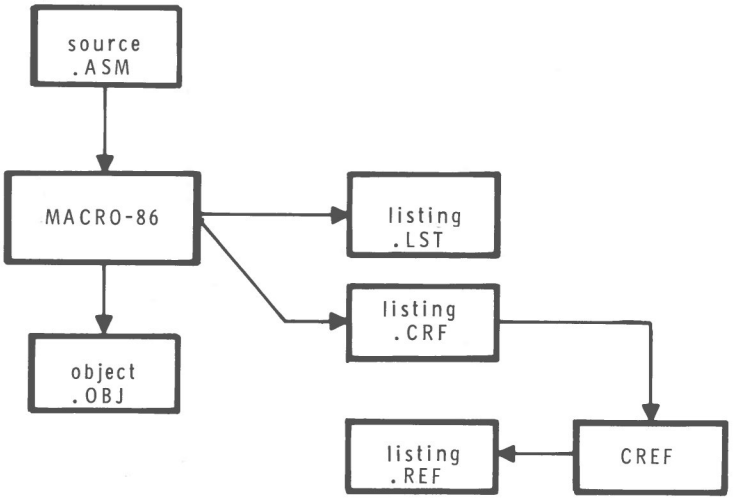


Pass Two of the Assembly

MACRO-86 will create on command, a listing file and a cross-reference file. The listing file contains the beginning relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. The listing contains a symbol table which shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file receives the default .LST filename extension.

The cross reference file contains a compact representation of variables, labels, and symbols. The cross reference file receives the default .CRF filename extension. When this cross reference file is processed by CREF, the file is converted into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order. It is followed by the line number in the source program where each is defined, followed by the line numbers where each is used in the program. The final cross reference listing receives the .REF filename extension. (Refer to the CREF Cross Reference Facility Chapter, Page 13.1, for further explanation and instructions.)

Introduction to MACRO-86



Listing and Cross Referencing

Application

Being aware of the features built into your MACRO- 86 assembler promotes efficiency and accuracy in the development of assembly language programs. Every feature might not be needed in every program. The ability to recognize the usefulness of a feature in a given situation will reduce application development costs by reducing the time required for coding and debugging.

The relocatable code produced by MACRO-86 makes it possible to design large software systems without the need to assign absolute addresses or create a complex memory map. You can develop each program module independently and test it with dummy arguments at any convenient address. When it runs satisfactorily, store it in a system library. At link time, program modules will be selected and structured in whatever way best accommodates the hardware and operating system requirements.

The use of conditional directives provides a great deal of portability and flexibility in source code modules. For example, you might be testing several algorithms in separate modules that are called from the same driver. The driver requires some changes to accommodate the particular module it is assembled with. Place the variable source statements for each module under an IFDEF module name condition line in the driver source code. The driver can now be assembled as is with any of the modules.

Programmers frequently encounter situations where identical, or nearly identical, blocks of code must be repeated within a program. A subroutine structure is inappropriate. The source code for such a block should be entered as a macro definition. You can then include the object code it produces anywhere in the program by invoking the macro's user defined name. Flexibility is increased by passing immediate arguments into each expansion of the macro, by nesting macros, and by including macros within conditionals or conditionals within macros.

Take time to understand the assembler command line structure. Use the various options to speed assembly by not producing unwanted or error-filled files.

Creating a MACRO-86 Source File

GENERAL FACTS ABOUT SOURCE FILES

Brief

.ASM is the preferred source filename extension. MACRO-86 accepts other extensions, provided they are entered with the filename. Problems may arise if .OBJ, .LST, .CRF, .REF, or .EXE are used.

All numeric values must begin with a numeral, i.e., 0FFFFh. The default input radix is decimal. For output the default is decimal for line numbers and hexadecimal for object code. The /O command provides octal code listings. Change the input radix with the .RADIX directive followed by 2, 8, 10, or 16. The radix of a single value may be changed by appending B for binary, Q or O for octal, D for decimal, or H for hexadecimal.

Legal characters in symbol names are:

A–Z 0–9 ? @ _ $

The characters 0–9 may not begin a symbol.

Special character operators and delimiters:

:	(colon) segment override operator.
.	(period) record or structure field name operator. Permitted in filename as first character only.
[]	(square brackets) around a register name define its contents as a pointer.
()	(parentheses) DUP expression operator, also used to set operand evaluation precedence.
<>	(angle brackets) delimit initialization values for records or structure, parameters in IRP macro blocks, and literals.

Details

To create a source file for MACRO-86, you need to use an editor program, such as EDLIN in Z-DOS. You simply create a program file as you would for any other assembly or high-level programming language. Use the general facts and specific descriptions in this and the following sections when creating the file.

In this portion of text, you will find discussions of the statement format and introductory descriptions of its components. Later, you will find full descriptions of names — variables, labels, and symbols. You will also find full descriptions of expressions and their components, operands and operators. Additionally, you will find full descriptions of the assembler directives.

Naming Your Source File

When you create a source file, you will need to name it. A filename may be any name that is legal for your operating system. MACRO-86 expects a specific three character filename extension, .ASM, whenever you run MACRO-86 to assemble your source file. MACRO-86 assumes that your source filename has the filename extension .ASM. This is not required. You may name your source file with any extension you like. However, when you rename a MACRO-86 source file, you must remember to specify the extension. If you use .ASM you will not need to specify the extension.

Because of this default action by MACRO-86, it is impossible to omit the filename extension. When you assemble a source file without a filename extension, MACRO-86 will assume that the source has a .ASM extension because you would not be specifying an extension. When MACRO-86 searches the disk for the file, it will not find the correct file and will either assemble the wrong file or will return an error message stating that the file cannot be found.

MACRO-86 gives the object file it outputs the default extension .OBJ. To avoid confusion or the destruction of your source file, you will want to avoid giving a source file an extension of .OBJ. For similar reasons, you will also want to avoid the .EXE, .LST, .CRF, and .REF extensions.

Legal Characters

The legal characters for your symbol names are:

A-Z 0-9 ? @ _ $

Only numerals 0-9 cannot appear as the first character of a name. A numeral must appear as the first character of a numeric value.

Additional special characters act as operators or delimiters:

:	(colon) segment override operator.
.	(period) operator for fieldname of Record or structure; may be used in a filename only if it is first character.
[ ]	(square brackets) around register names to indicate value in address in register not value (data) in register.
( )	(parentheses) operator in DUP expressions and operator to change precedence of operator evaluation.
< >	(angle brackets) operators used around initialization values for records or structure, around parameters in IRP macro blocks, and to indicate literals.

The square brackets and angle brackets are also used for syntax notation inthe discussions of the assembler directives, Page 10.88, as well as earlier in the manual. When these characters are operators and not syntax notation, you are told explicitly; for example, `angle brackets must be coded as shown.`

Numeric Notation

The default input radix for all numeric values is decimal. The output radix for all listings is hexadecimal for code and decimal for line numbers. The output radix can only be changed to octal radix by giving the /O switch when MACRO-86 is run. The input radix may be changed two ways:

1. The .RADIX directive

2. Special notation appended to a numeric value:

Radix	Range	Notation	Example
Binary	0-1	B	01110100B
Octal	0-7	Q or O (letter)	735Q 621O
Decimal	0-9	(none) or D	9384 (default) 8149D (when .RADIX directive changes default radix to not) decimal.
Hexadecimal	0-9 A-F	H	0FFH 80H (first character must be numeral in range 0-9)

What's in a Source File?

A source file for MACRO-86 consists of instruction statements and directive statements. Instruction statements are made of 8088 (8086) instruction mnemonics and their operands, which command specific processes directly to the 8088 processor. Directive statements are commands to MACRO-86 to prepare data for use in and by instructions.

Statements are usually placed in blocks of code assigned to a specific segment (code, data, stack, extra). The segments may appear in any order in the source file. Within the segments, generally speaking, statements may appear in any order that create a valid program. Some exceptions to random ordering do exist, which will be discussed under the affected assembler directives.

Every segment must end with an end segment statement ENDS. Every procedure must end with an end procedure statement ENDP. Every structure must end with an end structure statement ENDS. Likewise, the source file must end with an END statement that tells MACRO-86 where program execution should begin.

"Memory Organization" on Page 10.38 describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another and to your programming as a whole. This information is important and helpful for developing your programs. The information is presented as a prelude to the discussion of operands and operators.

STATEMENT LINE FORMAT

Brief

Format (typical directive statement):
<**name**><**action**><**expression**>;<**comment**>

Format (typical instruction statement):
<**action**> <**expression**> ;<**comment**>

Details

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

MACRO-86 directive statements consist of four fields — Name, Action, Expression, and Comment. For example:

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

MACRO-86 instruction statements usually consist of three fields: Action Expression, and Comment. For example:

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

An instruction statement may have a Name field under certain circumstances; see the discussion in the next section, on "Names".

Names

Brief

There are three categories of Names: Labels (referencing addresses); Variables (referencing data); and Symbols (referencing constants). The name field, if present, occurs first in a statement line. Name length is not limited. Only the first 31 characters are significant.

Details

The Name field, when present, is the first entry on the statement line. The name may begin in any column, although normally names are started in column one.

Names may be any length you choose. However, MACRO-86 considers only the first 31 characters significant when your source file is assembled.

One other significant use for names is with the MACRO directive. Although, all the rules covering names, described on Page 10.28 apply the same to MACRO names. The discussion of macro names is best left to the section described by the macro facility.

MACRO-86 supports the use of names in a statement line for three purposes: to represent code, to represent data, and to represent constants.

To make a name represent code, use:

NAME:	followed by a directive, instruction, or nothing at all
NAME LABEL NEAR	(for use inside its own segment only)
NAME LABEL FAR	(for use outside its own segment)
EXTRN NAME: NEAR	(for use outside its own module but inside its own segment only)
EXTRN NAME: FAR	(for use outside its own module and segment)

To make a name represent data, use:

NAME LABEL <**size**> (BYTE, WORD, etc.)
NAME Dx <**exp**>
EXTRN NAME: <**size**> (BYTE, WORD, etc.)

To make a name represent a constant, use:

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

Comments

Brief

Comments are always optional. A comment must be the last field in a source line and must be preceded by a semicolon. The COMMENT directive is preferable for multiline comments.

Details

Comments are never required for the successful operation of an assembly language program, but they are strongly recommended.

If you use comments in your program, every comment on every line must be preceded by a semicolon. If you want to place a very long comment in your program, you can use the COMMENT directive. The COMMENT directive releases you from the required semicolon on every line (refer to COMMENT, Page 10.94).

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

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

Action

Brief

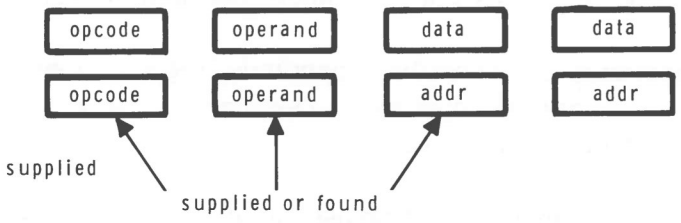
The action field follows the name field, or else occurs first if no name is present. The action field is not optional and may contain either an assembly language mnemonic or a directive to the assembler.

Details

The action field contains either an 8088 (8086) instruction mnemonic or a MACRO-86 assembler directive. Refer to Appendix L for a list of 8088 (8086) instruction mnemonics. The MACRO-86 directives are described in detail on Page 10.90.

If the name field is blank, the action field will be the first entry in the statement format. In this case, the action may appear starting in any column, one through maximum line length (less columns for action and expression).

The entry in the action field either directs the processor to perform a specific function or directs the assembler to perform one of its functions. Instructions command the processor's actions. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements.

Opcode in Action Field

Directives give the assembler directions for I/O, memory organization, conditional assembly, listing and cross reference control, and definitions.

Expressions

Brief

The expression field follows the action field and may contain from zero to two operands plus their associated operators. A comma is required to separate two operands.

Details

The expression field contains entries which are operands and/or combinations of operands and operators.

Some instructions take no operands, some take one, and others take two. For two operand instructions, the expression field consists of a destination operand and a source operand in that order, separated by a comma. For example:

Expression Field Operands

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

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

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

Expression Field Directives

A directive operand is a data operand, a code (addressing) operand, or a constant, depending on the nature of the directive.

For many instructions and directives, operands may be connected with operators to form a longer operand that looks like a mathematical expression. These operands are called "complex". Use of a complex operand permits you to specify addresses or data derived from several places. For example:

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

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

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

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

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

OPERANDS	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	

Precedence of Operands and Operators in the Expression Field

NOTE: Some operators can be used as operands or as part of an operand expressions. Refer to Page 10.37 for details on operands and operators.

Applying the Statement Line Format

Unlike program listings in high-level languages, assembly language listings provide very few immediate clues about the program function. This makes it especially important to understand the purpose and location of each field, as well as the legal entries each field may contain. Without this knowledge, it will be impossible to analyze a source listing, whether it is your own creation or someone else's. Assembling a file containing missing or incorrect delimiters, missing or incorrectly placed fields, or illegal field entries will result in a flood of syntactical errors that mask your program logic.

Names: Labels, Variables and Symbols

Brief

Labels are symbolic addresses used as operands of JMP, CALL, and LOOP instructions. Labels may reference procedures within or outside the segment where they are defined. Labels may also reference procedures in separately assembled modules. A colon separates a label from its following field.

Variables are symbolic addresses where data is stored. Unlike labels, variable names are separated from their following field by a space. A directive that defines the format of the data always occupies the second field.

Symbols are names that reference constant values. When an equal sign or the EQU directive occurs in field two, the expression following it is evaluated and assigned the chosen name. Symbols may also reference values defined in separately assembled modules. A space separates a symbol from the next field.

Details

Names are used in several capacities throughout MACRO-86, wherever any naming is allowed or required.

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

Names may be any length you choose. However, MACRO-86 will truncate names longer than 31 characters when your source file is assembled.

Names may be defined and used in a number of ways. This section introduces you to the basic ways to define and use names. You will discover additional uses as you study the sections on Expressions and Action, and as you use MACRO-86.

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

LABELS

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

Labels are defined one of four ways:

1. **<name>:**

   Use a name followed immediately by a colon. This defines the name as a NEAR label. <name>: may be prefixed to any instruction and to all directives that allow a name field. <name>: may also be placed on a line by itself.

   Examples:

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

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

   Use the LABEL directive. Refer to the discussion of the LABEL directive in "Memory Directives", Page 10.111.

   NEAR and FAR are discussed under the TYPE attribute on Page 10.31.

   Examples:

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

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

   Use the PROC directive. Refer to the discussion of the PROC directive , "Memory Directives", Page 10.115.

   NEAR is optional because it is the default if you enter only **<name> PROC.** NEAR and FAR are discussed under the Type Attribute on Page 10.31.

   Examples:

```
REPEAT    PROC   NEAR
CHECKING  PROC          ;same as CHECKING PROC NEAR
FIND_CHR  PROC   FAR
```

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

   Use the EXTRN directive.

   NEAR and FAR are discussed under the TYPE attribute on Page 10.31.

   Refer to the discussion of the EXTRN directive "Memory Directives", Page 10.104.

   Examples:

```
EXTRN FOO:NEAR
EXTRN ZOO:FAR
```

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

Segment

Labels are defined inside segments. The segment must be assigned to the CS segment register to be addressable. The segment may be assigned to a group, in which case the group must be addressable through CS. MACRO-86 requires that a label be addressable through the CS register. Therefore, the segment (or group) attribute of a symbol is the base address of the segment (or group) where it is defined.

Offset

The offset attribute is the number of bytes from the beginning of the label's segment to where the label is defined. The offset is a 16-bit unsigned number.

Type

There are two types of labels: NEAR or FAR. NEAR labels are used for references from within the segment where the label is defined. NEAR labels may be referenced from more than one module, as long as the references are from a segment with the same name and attributes and that has the same CS ASSUME.

FAR labels are used for references from segments with a different CS ASSUME or that have more than 64K bytes between their label reference and their label definition.

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

VARIABLES

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

A variable represents addresses where a specified value may be found.

Variables look much like labels and are defined alike in some ways. The differences are important.

Variables are defined three ways:

1. **<name> <define-dir>** ;no colon!
   <name> <struc-name> <expression>
   <name> <rec-name> <expression>

   <define-dir> is any of the five define directives: DB, DW, DD, DQ, DT

   Example:

```
START_Move     DW     ?
```

   <struc-name> is a structure name defined by the STRUC directive.

   Examples:

```
CORRAL STRUC

        .
        .
        .
CORRAL ENDS
HORSE  CORRAL    <'SADDLE'>
```

   Note that HORSE will have the same size as the structure CORRAL.

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

Note that SMALL will have the same size as the record GARAGE.

See the DEFINE, STRUC, and RECORD directives on Pages 10.95, 10.126 and 10.122, respectively, under "Memory Directives".

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

Use the LABEL directive with one of the size specifiers.

<size> is one of the following size specifiers:

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

Example:

```
CURSOR   LABEL   WORD
```

See LABEL directive on Page 10.111.

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

Use the EXTRN directive with one of the size specifiers described above. See the EXTRN directive on Page 10.104.

Example:

```
EXTRN FOO:DWORD
```

Variables as well as labels have three attributes: segment, offset, and type.

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

Type

The type attribute is the size of the variable's location, as specified when the variable is defined. The size depends on which Define directive was used or which size specifier was used to define the variable.

Directive	Type	Size
DB	BYTE	1 byte
DW	WORD	2 bytes
DD	DWORD	4 bytes
DQ	QWORD	8 bytes
DT	TBYTE	10 bytes

SYMBOLS

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

Symbols are defined three ways:

1. <**name**> **EQU** <expression>. See EQU directive , "Memory Directives", Page 10.100.

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

   Examples:

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

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

   Use the Equal Sign directive. See "Equal Sign" on Page 10.100.

   <expression> may be any valid expression.

   Examples:

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

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

   Use the EXTRN directive with type ABS. See EXTRN on Page 10.104.

   Example:

```
EXTRN BAZ: ABS
```

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

Application

Names should be defined for labels, variables, and symbols whenever possible. The use of numerals should be limited to transient values, or values which have no extended meaning beyond their numeric value. This provides a program listing with a high density of user-defined words, making its logic easier to follow. It also provides a more effective use of the assembler, which relies heavily on internal symbol tables for its operation.

Expressions: Operands and Operators

INTRODUCTION

Every expression consists of at least one operand (a value). An expression may consist of two or more operands. Multiple operands are joined by operators. The result is a series of elements that look like a mathematical expression.

This portion of the chapter describes the types of operands and operators that MACRO-86 supports. The discussion of memory organization in a MACRO-86 program acts as a preface to the descriptions of operands and operators, and as a link to topics discussed earlier.

MEMORY ORGANIZATION

Brief

Memory segments of up to 64K bytes are defined by the SEGMENT and ENDS directives. Segment types are CODE, DATA, STACK, and EXTRA. They are designated CS, DS, SS, and ES respectively. All address references are relative until LINK sets an absolute base address for each segment. Four segment registers store the base addresses.

A single segment may contain any number of separate code modules, provided the 64K limit is observed. The address of each module is an offset from the common segment base.

The GROUP directive permits referencing several segments to a common base address. The 64K limit then applies to the total size of all the segments in the group.

The CS register cannot be modified by instructions within the current code segment. This protection does not apply to DS, SS, or ES.

Because different functions are performed on passes one and two, only certain types of errors can be detected on each pass. When forward references are processed, special operators may be used to assist the assembler in producing the correct amount of code.

Details

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

You should note that MACRO-86 leaves everything to do with segments to LINK. LINK resolves all references. For that reason, MACRO-86 does not check (because it cannot) if your references are entered with the correct distance type. Values such as OFFSET are also left to the LINK for resolution.

Although a segment may not be more than 64K bytes long, you may, as long as you observe the 64K limit, divide a segment among two or more modules. The SEGMENT statement in each module must be the same in every respect.

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

You have the option of grouping several segments into a group, using the GROUP directive. When you group segments, you tell MACRO-86 that you want to be able to refer to all of these segments as a single entity. This does not eliminate segment identity, nor does it make values within a particular segment less immediately accessible. It does make values relative to a group base. The usefulness of grouping is that you can refer to data items without worrying about segment overrides and about changing segment registers often.

You should note that references within segments or groups are relative to a segment register. Until linking is completed, the final offset of a reference is relocatable. For this reason, the OFFSET operator does not return a constant. The major purpose of OFFSET is to cause MACRO-86 to generate an immediate instruction; that is, to use the address of the value instead of the value itself.

There are two kinds of references in a program:

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

2. *Data references* — all other references — These references are usually relative to the DS register, but this assignment may be overridden.

   When you give a forward reference in a program statement, for example:

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

MACRO-86 first looks for the segment of the reference. MACRO-86 scans the segment registers for the SEGMENT of the reference then the GROUP, if any, of the reference. However, the use of the OFFSET operator always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must override this restriction by using the GROUP name and the colon operator, for example:

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

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

```
MOV AX OFFSET DS:<ref>
```

The result of both of these statements is the same.

Code labels have four attributes:

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

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

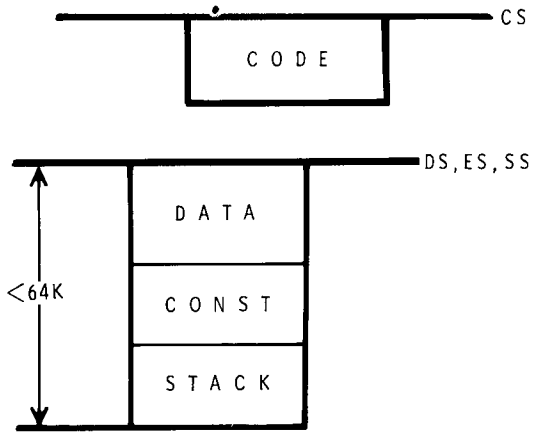
When you enter a FAR JMP or FAR CALL, you are changing both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

Let's take a common case, a segment called CODE; and a group (called DGROUP) that contains three segments (called DATA, CONST, and STACK).

The program statements would be:

```
DGROUP   GROUP    DATA, CONST, STACK
         ASSUME   CS: CODE, DS: DGROUP, SS: DGROUP, ES: DGROUP
         MOV      AX, DGROUP  ;CS initialized by entry'
         MOV      DS, AX      ;you initialize DS, do this
                              ;as soon as possible,
                              ;especially before any DS
                              ;relative references
          .
          .
          .
```

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



Segment Register Relationship to Code Address

Given this arrangement, a statement like:

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

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

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

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

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

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

MACRO-86 is a two-pass assembler. During pass one, it builds a symbol table and calculates how much code is generated but does not produce object code. If undefined items are found (including forward references), assumptions are made about the reference so that the correct number of bytes are generated on pass one. Only certain types of errors are displayed; errors involving items that must be defined on pass one. No listing is produced unless you give a /D switch when you run the assembler. The /D switch produces a listing for both passes.

On pass two, the assembler uses the values defined in pass one to generate the object code. Definitions of references during pass two are checked against the pass one value, which is in the symbol table and also, the amount of code generated during pass two. If either is different, MACRO-86 returns a phase error.

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

The following references must be known on pass one.

```
IF/IFE <expression>
```

If <expression> is not known on pass one, MACRO-86 does not know to assemble the conditional block (or which part to assemble if ELSE is used). On pass two, the assembler would know and would assemble, resulting in a phase error.

For example:

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

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

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

Because OFFSET tells MACRO-86 to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code.

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

CALL FAR PTR <forward-label>

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

JMP SHORT <forward-label>

However, you must be sure that the target is indeed within 127 bytes or MACRO-86 will not find it.

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

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

You may want to use the variable FOO as an immediate operand. In this case, you could enter either of the statements (they are equivalent):

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

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

OPERANDS

Brief

Legal operands:

Immediate class:

- Data items (e.g., 25, 6000, 0FFFH, 412Q, "HI", 11110000B)
- Symbols (e.g., FOO, IOBYT, TTYOUT)

Immediate operands are source operands only.

Register class:

- Register names, except flag or segment registers are not eligible for logical and arithmetic operations.

Memory class:

- Direct (e.g., FOO, OFFSET XTABLE, BEGIN)
- Indexed (e.g., [BX], [BP], [DI], [SI], [DI]XTABLE, [BX]FOO)

There are only four index registers. BP defaults to the SS segment, the others to DS.

- Structure (e.g., ZOO.BEAR, [BX].ZOO, ZOO.KEEPER)

Details

There are three types of operands: Immediate, Register, or Memory operands. There are no restrictions on combining the various types of operands.

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

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

Immediate Operands

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

Instructions that take two operands permit an immediate operand as the source operand only (the second operand in an instruction statement). For example:

```
MOV AX,9
```

Data Items

The default input radix is decimal. Any numeric values entered without numeric notation appended will be treated as a decimal value. MACRO-86 recognizes values in forms other than decimal when special notation is appended. These other values include ASCII characters as well as numeric values.

Data Form	Format	Example
Binary	XXXXXXXXB	01110001B
Octal	XXXO XXXQ	7350 (letter O) 412Q
Decimal	XXXXX XXXXXD	65535 (default) 1000D (when .RADIX changes input radix to nondecimal)
Hexadecimal	XXXXH	0FFFH (first digit must be 0-9)
ASCII	'XX' "xx"	'OM' (more than two with DB only; "OM" both forms are synonomous)
10 real	XX.XXfoXX	25.23E-7 (floating point format)
16 real	X...XR	8F76DEA9R (first digit must be 0-9, the total number of digits must be 8, 16, or 20; or 9, 17, or 21 if first digit is 0)

MASM Recognized Data Item Values

Symbols

Symbol names equated with some form of constant information may be used as immediate operands. Using a symbol constant in a statement is the same as using a numeric constant. Therefore, using a sample statement, you could enter:

MOV AX,FOO

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

```
FOO EQU 9
```

Register Operands

The 8088 processor contains fourteen registers. These registers have two-letter indentifiers that the assembler recognizes. These identifiers are reserved and may not be used for user-defined names.

The registers are appropriated to different tasks: general registers, pointer registers, counter registers, index registers, segment registers, and a flag register.

The general registers are both 8-bit and 16-bit registers. Actually, the 16-bit general registers are composed of a pair of 8-bit registers, one for the low byte (bits 0–7) and one for the high byte (bits 8–15). Note, however, that each 8-bit general register can be used independently from its mate. In this case, each 8-bit register contains bits 0–7.

You initialize segment registers. They contain segment base values. The segment register names CS, DS, SS, ES can be used with the colon segment-override operator to inform MACRO-86 that an operand is in a different segment than specified in an ASSUME statement. (See the Segment Override Operator, Page 10.57.)

The flag register is one 16-bit register containing nine one-bit flags (six arithmetic flags and three control flags).

Each of the registers, except segment registers and flags, can be an operand in arithmetic and logical operations.

MOD=11			Register Mode
R/M	8-bit W=0	16-bit W=1	
000	AL	AX	
001	CL	CX	
010	DL	DX	
011	BL	BX	
100	AH	SP	
101	CH	BP	
110	DH	SI	
111	BH	DI	

Register/Memory Field Encoding

EFFECTIVE ADDRESS CALCULATION			
R/M	MOD=00	MOD=01	MOD=10
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16
011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16
100	[SI]	[SI]+D8	[SI]+D16
101	[DI]	[DI]+D8	[DI]+D16
110	DIRECT ADDRESS	[BP]+D8	[BP]+D16
111	[BX]	[BX]+D8	[BX]+D16

Note: D8 = a byte value; D16 = a word value

Other Registers:

Segment:	CS	code segment
	DS	data segment
	SS	stack segment
	ES	extra segment

Effective Address Calculation

Flags:	six one-bit arithmetic flags		three one-bit control flags	
Flag	CF	carry flag	DF	direction flag
	PF	parity flag	IF	interrupt-enable
	AF	auxiliary flag	TF	trap flag
	ZF	zero flag		
	SF	sign flag		

Flags

NOTE: The BX, BP, SI and DI registers are also used as memory operands. The distinction is: when these registers are enclosed in square brackets [ ], they are memory operands; when they are not enclosed in square brackets, they are register operands. (See "Memory Operands", Page 10.51.)

Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct MACRO-86 to an address to find some data or instruction.

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

Memory operands fit into three categories: 1) those that use a base or index register, indexed memory operands; 2) those that do not use a register, direct memory operands; and 3) structure operands.

Direct Memory Operands

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

Memory operands can be used as destination operands as well as source operands for instructions that take two operands. For example:

```
MOV AX,FOO                    ;FOO is the direct memory operand
MOV FOO,CX                    ;in thse two examples, not
                              ;AX or CX.
```

Indexed Memory Operands

Indexed memory operands use base, index registers, constants, displacement values, and variables, often in combination. When you combine indexed operands, you create an address expression.

Indexed memory operands use square brackets to indicate indexing (by a register or by registers) or subscripting (for example, FOO[5]). The square brackets are treated like plus signs (+). Therefore,

`FOO[5]` is equivalent to `FOO+5`
`5[FOO]` is equivalent to `5+FOO`

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

The types of indexed memory operands are:

Base registers:	[BX] [BP] BP has SS as its default segment register, all others have DS as default.
Index registers:	[DI] [SI]
[constant]:	immediate in square brackets [8]. [FOO]
Displacement:	8-bit or 16-bit value. Used only with another indexed operand.

These elements may be combined in any order. The only restriction is that neither two base registers nor two indexed registers can be combined:

```
[BX+BP] ;illegal
[SI+DI] ;illegal
```

Some examples of indexed memory operand combinations:

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

More examples of equivalent forms:

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

Structure Operands

Structure operands take the form <variable>,<field>.

The variable is any name you give when coding a statement line that initializes a Structure field. The variable may be an anonymous variable, such as an indexed memory operand.

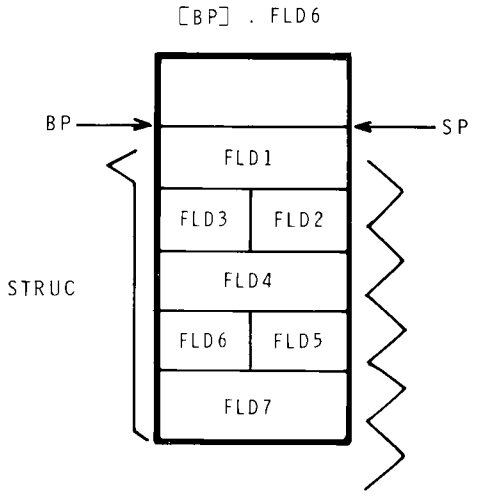
The field is a name defined by a DEFINE directive within a STRUC block. Field is a typed constant.

The period (.) must be included.

Application

```
ZOO      STRUC
GIRAFFE DB  ?
ZOO      ENDS

LONG_NECK   ZOO <16>

MOV AL, LONG_NECK. GIRAFFE

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

The use of structure operands can be helpful in stack operations. If you set up the stack segment as a structure, setting BP to the top of the stack (BP equal to SP), then you can access any value in the stack structure by fieldname indexed through BP1. For example:



Structure Operands in Stack Operation

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

OPERATORS

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 to isolate fields of records.

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

Attribute Operators

Brief

MACRO-86 Attribute Operators:

- Attribute override operators:

  PTR : SHORT THIS HIGH LOW

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

- Value returning operators:

  SEG OFFSET TYPE .TYPE LENGTH SIZE

These operators return the attribute values of variables and labels.

- RECORD specific operators:

  shift count WIDTH MASK

These operators isolate fields within a RECORD.

Details

Attribute operators used as operands perform one of three functions:

- Override an operand's attributes.
- Return the values of operand attributes.
- Isolate record fields (record specific operators).

The following list shows all the attribute operators by type:

Override operators
- PTR
- colon (:) (segment override)
- SHORT
- THIS
- HIGH
- LOW

Value returning operators
- SEG
- OFFSET
- TYPE
- .TYPE
- LENGTH
- SIZE

RECORD specific operators
- Shift count (field name)
- WIDTH
- MASK

Override Operators

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

Pointer (PTR)

Brief

Format: <**attribute**> **PTR** <**expression**>

Pointer (PTR) overrides the type or distance of an operand. The operand preceding PTR replaces the type or distance of the operand following it. PTR is most often used to make explicit the type of a variable defined in a forward reference. For example:

```
ADD BYTE PTR FOO,9
```

PTR is also used to access data as a type other than the type specified when the data was defined. For example:

```
MOV AL,BYTE PTR WHOLEWORD
```

Details

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

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

The expression is the operand whose attribute is to be overridden.

The most important and frequent use for PTR is to assure that MACRO-86 understands what attribute the expression is supposed to have. This is especially true for the type attribute. Whenever you place forward references in your program, PTR will clear the distance or type of expression. This way you can avoid phase errors.

The second use of PTR is to access data by type other than the type in the variable definition. Most often this occurs in structures. If the structure is defined as WORD, but, you want to access an item as a byte, PTR is the operator for this.

However, a much easier method is to enter a second statement that defines the structure in bytes, too. This eliminates the need to use PTR for every reference to the structure. Refer to the LABEL directive, "Memory Directives", on Page 10.111.

Application

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

ADD BYTE PTR FOO,9
```

Segment Override (:) (colon)

Brief

Format: <**segment-register**>:<**address-expression**>
<**segment-name**>:<**address-expression**>
<**group-name**>:<**address-expression**>

The colon (:) overrides the assumed segment of an operand containing a memory reference. The correct segment precedes the colon; and the address to be referenced follows it. Describe the segment with either a segment register, a segment name, or a group name. For example:

```
JMP ES:ERROR_ROUTINE

MOV AX,DATASEG:VARIABLE

MOV AX,OFFSET DGROUP:VARIABLE
```

Details

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

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

MACRO-86 assumes that labels are addressable through the current CS register. MACRO-86 assumes that variables are addressable through the current DS register, or possibly the ES register, by default. If the operand is in another segment and you have not alerted MACRO-86 through the ASSUME directive, you will need to use a segment override operator. Also, if you want to use a nondefault relative base, not the default segment register, you will need to use the segment override operator for forward references. If MACRO-86 can reach an operand through a nondefault segment register, it will use it, but the reference cannot be forward in this case.

<segment-register> is one of the four segment register names: CS, DS, SS, ES.

<segment-name> is a name defined by the SEGMENT directive.

<group name> is a name defined by the GROUP directive.

Application

```
MOV AX,ES:[BX+SI]

MOV CSEG:FAR_LABEL,AX

MOV AX,OFFSET DGROUP:VARIABLE
```

Short

Brief

Format: **SHORT <label>**

SHORT overrides a NEAR attribute of a label that follows a jump instruction. Its use shortens the jump instruction by one byte. It is legal only when the target label is within 127 bytes of the jump instruction. For example:

```
JMP SHORT NEXTLABEL
```

Details

SHORT overrides NEAR distance attribute of labels used as targets for the JMP instruction. SHORT tells MACRO-86 that the distance between the JMP statement and the <label> specified as its operand is not more than 127 bytes in either direction.

The major advantage of using the SHORT operator is to save a byte. Normally, the <label> carries a two-byte pointer to its offset in its segment. Because a range of 256 bytes can be handled in a single byte, the SHORT operator eliminates the need for the extra byte (which would carry 00 or FF anyway). However, you must be sure that the target is within + or − 127 bytes of the JMP instruction before using SHORT.

Application

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

This

Brief

Format: **THIS** <**distance**>
THIS <**type**>

THIS defines the current address within the current segment as a named operand. Legal attributes are NEAR, FAR, BYTE, WORD, or DWORD. For example:

```
STACKLIMIT = THIS FAR

FIRSTWORD = THIS WORD
```

Details

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

The argument to THIS may be:

1. A distance (**NEAR** or **FAR**)

2. A type (**BYTE**, **WORD**, or **DWORD**)

THIS <**distance**> creates an operand with the distance attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

THIS <type>creates an operand with the type attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

Application

```
TAG EQU THIS BYTE same as TAG LABEL BYTE

SPOT_CHECK = THIS NEAR same as SPOT_CHECK LABEL NEAR
```

High, Low

Brief

Format: **HIGH <expression>**
LOW <expression>

HIGH and LOW isolate the upper or lower byte of a 16-bit value. For example:

```
MOV AH,HIGH WHOLEWORD

MOV BH,LOW 0F003H
```

Details

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

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

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

Application

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

MOV AL,LOW 0FFFFH
```

VALUE RETURNING OPERATORS

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

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

Because variables in MACRO-86 have three attributes, you need to use value returning operators to isolate single attributes, as follows:

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

SEG

Brief

Format: **SEG** <**label**>
SEG <**variable**>

SEG followed by a variable or label returns the base address of the segment where it resides. For example:

```
MOV AX,SEG TABLE1
```

Details

SEG returns the segment value (segment base address) of the segment enclosing the label or variable.

Application

```
MOV AX,SEG VARIABLE_NAME
MOV AX,<segmentvariable>:<variable>
```

Offset

Brief

Format: **OFFSET** <**label**>
OFFSET <**variable**>

OFFSET followed by a variable or label returns its distance from the base address of the segment where it resides. For example:

```
MOV BX, OFFSET FOO
```

Details

OFFSET returns the offset value of the variable or label within its segment (the number of bytes between the segment base address and the address where the label or variable is defined).

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

NOTE: OFFSET does *not* make the value a constant.

Only LINK can resolve the final value.

NOTE: OFFSET is not required with uses of the DW or DD directives. The assembler applies an implicit OFFSET to variables in address expressions following DW and DD.

Example:

```
MOV BX, OFFSET FOO
```

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

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

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

Type

Brief

Format: **TYPE** <**label**>
TYPE <**variable**>

TYPE followed by a variable returns the number of bytes reserved for that variable. TYPE followed by a label returns its distance attribute (where 0FFFFH = NEAR and 0FFFEH = FAR). For example:

```
MOV AL,TYPE FOO

MOV AX,TYPE ERROR_ROUTINE
```

Details

If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the variable type, as follows:

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

If the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Application

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

.Type

Brief

Format: **.TYPE <variable>**

The .TYPE operator followed by an expression returns a byte which describes the mode and definition status of the evaluated expression. It is used most often to set up a test for conditional assembly. For example:

```
Z = .TYPE HORSES

IF Z...
```

Details

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

The byte that is returned is configured as follows:

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

0	the mode is Absolute
1	the mode is Program Related
2	the mode is Data Related

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

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

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

Application

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

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

Length

Format: **LENGTH <variable>**

LENGTH followed by a variable returns the number of type units allocated to the variable in a preceding DUP expression. LENGTH returns one if the variable is not defined in a DUP expression. For example:

```
FOO DW 100 DUP(1)
MOV CX,LENGTH FOO
```

Details

LENGTH accepts only one variable as its argument.

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for that variable.

If the variable is defined by a DUP expression, LENGTH returns the number of type units duplicated; that is, the number that precedes the first DUP in the expression.

If the variable is not defined by a DUP expression, LENGTH returns one.

Application

```
FOO DW 100 DUP(1)

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

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

LENGTH BAZ is still 100. regardless of the expression following DUP.

```
GOO DD (?)

LENGTH GOO
```

returns one because only one unit is involved.

Size

Format: **SIZE <variable>**

SIZE followed by a variable returns the number of bytes allocated to the variable in a preceding DUP expression. For example:

```
FOO DW 100 DUP(1)
MOV BX,SIZE FOO
```

Details

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

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

Application

```
FOO DW 100 DUP(1)

MOV BX,SIZE FOO                    ;get total bytes in array
```

SIZE = LENGTH × TYPE
SIZE = 100 × WORD
SIZE = 100 × 2
SIZE = 200

RECORD SPECIFIC OPERATORS

Record specific operators are used to isolate fields in a record.

Records are defined by the RECORD directive (see "Memory Directives", Page 10.122). A record may be up to 16 bits long. The record is defined by fields, which may be from one to 16 bits long. To isolate one of the three characteristics of a record field, you use one of the record specific operators, as follows:

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

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

FOO is a record defined by the RECORD directive

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

BAZ is a variable used to allocate FOO

```
BAZ FOO < >
```

FIELD1, FIELD2, and FIELD3 are the fields of the record FOO.

Shift-count — (Record fieldname)

Brief

Format: <**record-fieldname**>

Shift count is a function derived from the fieldname of the field to be isolated from a RECORD. Use it as follows:

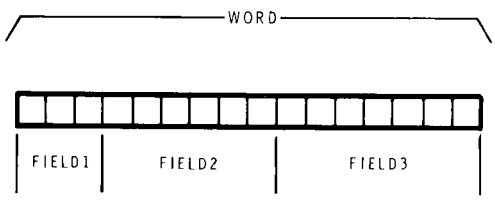
```
MOV DX, WHOLERECORD
MOV CL, FIELD2                    ; CL = SHIFT COUNT
SHR DX, CL                        ; FIELD2 @LOW END OF DX
```

Details

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

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

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



FIELD1 has a shift count of 3.
FIELD2 has a shift count of 7.
FIELD3 has a shift count of 0.

Field Distribution Within a Record

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

Application

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

FIELD2 is now right shifted, ready for access.

Mask

Brief

Format: **MASK <record-fieldname>**

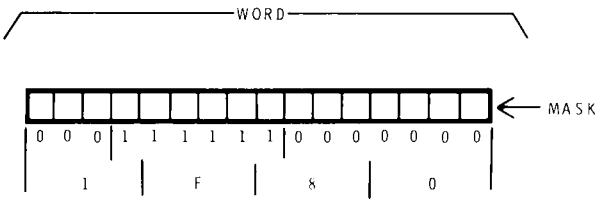
MASK followed by a fieldname returns a bit mask to isolate the selected field. For example:

```
MOV DX,WHOLERECORD
AND DX,MASK FIELD2 ;BITS OUTSIDE FIELD2 = 0
```

Details

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

With the same diagram as was used for Shift-count, MASK would appear as:



The MASK of FIELD2 equals 1F80H.

Masking a Record to Return Field Value

Application

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

FIELD2 is now isolated.

Width

Brief

Format: **WIDTH <record-fieldname>**
WIDTH <record>

WIDTH followed by a recordname or fieldname returns its width in bits. For example:

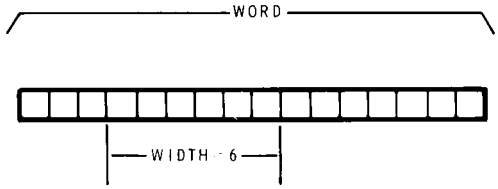
```
MOVCL,WIDTHFIELD2
```

Details

When a record-fieldname is given as the argument, WIDTH returns the width of a record field as the number of bits in the record field.

Using the diagram under Shift-count again, WIDTH record-fieldname, can be diagrammed as:

The WIDTH of FIELD1 equals 3.
The WIDTH of FIELD2 equals 6.
The WIDTH of FIELD3 equals 7.



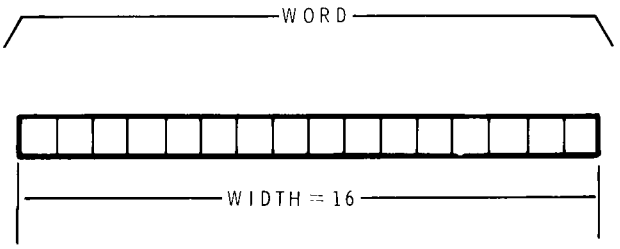
Record Fieldname WIDTH

Application

```
MOVCL,WIDTHFIELD2
```

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

When a record is given as the argument, WIDTH returns the width of a record as the number of bits in the record.



The WIDTH of this <record> equals 16.

Record WIDTH

NOTE: Though the diagram examples that were shown on these last few pages for the Record Specific operators (Shift-count, MASK and WIDTH) all used a 16-bit record, the record length is not limited to 16 bits.

Arithmetic Operators

Brief

These arithmetic operators are used to combine elements of an expression:

\+ − * / − (unary)

MOD (returns remainder of division)

SHR, SHL (shift right or left)

Details

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

The arithmetic operators are used to combine operands to form an expression that results in a data item or an address.

Except for + and − (binary), operands must be constants.

For plus (+), one operand must be a constant.

For minus (−), the first (left) operand may be a nonconstant, or both operands may be nonconstants. But, the right may not be a nonconstant if the left is constant.

*	Multiply
/	Divide
MOD	Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolute.

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 will be 11B (03).

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

Example:

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

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

– (Unary Minus) Indicates that the following value is negative, as in a negative integer.

+ Add. One operand must be a constant; one may be a nonconstant.

– Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be nonconstants. But the right may be a nonconstant only if the left is also a nonconstant and in the same segment.

Relational Operators

Brief

These relational operators set up conditional directives:

EQ(=)
NE(<>)
LT(<)
LE(<=)
GT(>)
GE(>=)

Details

Relational operators compare two constant operands.

If the relationship between the two operands matches the operator, FFFFH is returned.

If the relationship between the two operands does not match the operator, a zero is returned.

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

EQ Equal. Returns true if the operands equal each other.

NE Not Equal. Returns true if the operands are not equal to each other.

LT Less Than. Returns true if the left operand is less than the right operand.

LE Less than or Equal. Returns true if the left operand is less than or equal to the right operand.

GT Greater Than. Returns true if the left operand is greater than the right operand.

GE Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.

Logical Operators

Brief

These logical operators may be used in conditional directives or bit-for-bit evaluations:

NOT AND OR XOR

Details

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate for the logical relationship defined by the logical operator.

Logical operators can be used two ways.

1. To combine operands in a logical relationship. In this case, all bits in the operands will have the same value (either 0000 or FFFFH). In fact, it is best to use these values for true (FFFFH) and false (0000) for the symbols you will use as operands because in conditionals anything nonzero is true.

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

NOT Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.

AND Logical AND. Returns true if both operands are true. Returns false if either operand is false or if both are false. Both operands must be absolute values.

OR Logical OR. Returns true if either operand is true or if both are true. Returns false if both operands are false. Both operands must be absolute values.

XOR Exclusive OR. Returns true if either operand is true and the other is false. Returns false if both operands are true or if both operands are false. Both operands must be absolute values.

Expression Evaluation: Precedence Of Operators

Brief

Expressions are evaluated left to right for each of eleven levels of operator precedence. Operators of equal precedence are then performed left to right. Precedence hierarchy is:

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: ( ), < >, [ ]
   Structure variable operand: <var>.<field>
2. colon
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MOD, SHL, SHR
6. +, − (unary and binary)
7. EQ, NE, LT, LE, GT, GE
8. NOT
9. AND
10. OR, XOR
11. SHORT, .TYPE

Details

Expressions are evaluated with higher precedence operators first, then left to right for equal precedence operators.

For example:

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

MOV AX, 101B SHL (2*2) =MOV AX, 01010000B
```

SHL and * are equal precedence. Therefore, their functions are performed in the order the operators are encountered (left to right).

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

1. LENGTH, SIZE, WIDTH, MASK

   Entries inside: parentheses ( )
   angle brackets < >
   square brackets [ ]

   structure variable operand: <variable>.<field>

2. segment override operator: colon (:)

3. PTR, OFFSET, SEG, TYPE, THIS

4. HIGH, LOW

5. *,/, MOD, SHL, SHR

6. +, − (both unary and binary)

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

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT,.TYPE

Application

The wide range of addressing modes and data types supported by MACRO-86 contribute to the efficiency of the programs developed with it. To make optimum use of these capabilities, you will need to understand the full set of operand choices described in this section. In most cases, it is the operand structure, or the expression structure of combined operands, which determines the functional addressing mode or the data type being accessed or created. The attribute operators in particular exist to maximize the power of the 16-bit architecture in your system. By familiarizing yourself with them, you will be able to structure and access your program data in many powerful ways not available in an eight-bit system.

Action: Instructions and Directives

INTRODUCTION

The action field contains either an 8086 (8088) instruction mnemonic or a MACRO-86 assembler directive.

Following a name field entry (if any), action field entries may begin in any column. Specific spacing is not required. The only benefit of consistent spacing is improved readability. If a statement does not have a name field entry, the action field is the first entry.

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

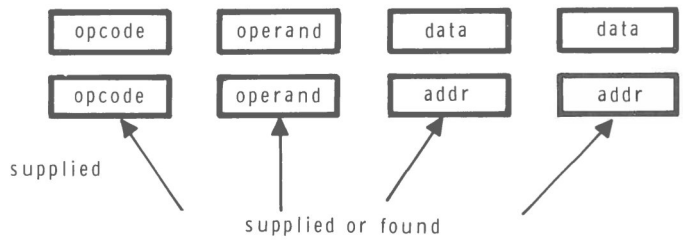
INSTRUCTIONS

Brief

The action field may contain either a directive to the assembler or an assembly language mnemonic. The action field may be entered in either of the first two columns. If a label is present, it will be the second field; otherwise it will be the first. Some mnemonics imply the operand. Others require the specification of one or two operands. Assembly language mnemonics are not detailed in this manual. They are listed in Appendices L and M.

Details

Instructions command the processor's actions. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements.

(opcode equates to the binary code for the action of an instruction)

This manual does not contain detailed descriptions of the 8086/8088 instruction mnemonics and their characteristics. For this, you need to consult other texts:

1. Mores, Stephen P, *The 8086 Primer*. Rochell Park, NJ: Hayden Publishing Co., 1980.

2. Rector, Russel and George Alexv, *The 8086 Book*. Berkeley, CA: Osbourne/McGraw-Hill. 1980.

3. *The 8086 Family User's Manual*. Santa Clara, CA: Intel Corporation. 1980.

Appendices K and L contain an alphabetical listing and a grouped listing of the instruction mnemonics, respectively. The alphabetical listing shows the full name of the instruction. Following the alphabetical list is a list that groups the instruction mnemonics by the number and type of arguments they take. Within each group, the instruction mnemonics are arranged alphabetically.

DIRECTIVES

Brief

Assembler directives occur in four functional categories: memory, conditional, macro, and listing. Appendix K contains a categorical list.

Details

Directives give the assembler directions for input and output, memory organization, conditional assembly, listing and cross reference control, and definitions.

The directives have been divided into groups by the function they perform. Within each group, the directives are described alphabetically.

The groups are:

Memory Directives — Directives in this group are used to organize memory. Because there is no "miscellaneous" group, the memory directives group contains some directives that do not, strictly speaking, organize memory, such as COMMENT.

Conditional Directives — Directives in this group are used to test conditions of assembly before proceeding with assembly of a block of statements. This group contains all of the IF (and related) directives.

Macro Directives — Directives in this group are used to create blocks of code called macros. This group also includes some special operators and directives that are used only inside macro blocks. The repeat directives are considered macro directives for descriptive purposes.

Listing Directives — Directives in this group are used to control the format and, to some extent, the content of listings that the assembler produces.

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

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

Memory Directives

ASSUME

Brief

Format: **ASSUME<seg-reg>:<seg-name>[...]**

or

ASSUME NOTHING

ASSUME takes two arguments separated by a colon. The first selects a segment register: CS, DS, SS, or ES. The second provides the segment or group name to be accessed through that register. The argument pair may be repeated up to four times. If NOTHING is entered for the second argument, a segment register must prefix every location reference. For example:

```
ASSUME CS:CODE_AREA DS:STORAGE_AREA

ASSUME CS:NOTHING
```

Details

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register. When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from one to four arguments to ASSUME.

The valid seg-reg entries are:

CS, DS, ES, and SS.

The possible entries for seg-name are:

1. the name of a segment declared with the SEGMENT directive
2. the name of a group declared with the GROUP directive
3. an expression: either SEG variable-name or SEG label-name.
4. the key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement.

If ASSUME is not used or if NOTHING is entered for seg-name, each reference to variables, symbols, labels, and so forth in a particular segment must be prefixed by a segment register. For example, DS:FOO is used instead of simply FOO.

Application

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

COMMENT

Brief

Format: **COMMENT<delimiter><text><delimiter>**

COMMENT allows you to enter a comment string of any length. Any delimiter other than a space may be used. The second occurrence of the delimiter terminates the string.

Details

The first symbolic character (i.e., not a space, tab, non-printing character or letter) encountered after COMMENT is the delimiter. The following text comprises a comment block which continues until the next occurrence of delimiter.

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

If you use COMMENT inside a macro block, the comment block does not appear on your listing unless you also place the .CALL directive in your source file.

Application

Using an asterisk as the delimiter, the format of the comment block would be:

```
COMMENT *
any amount of text entered
here as the comment block
.
.
.    *    ;return to normal mode
```

DEFINE (BYTE, WORD, DOUBLE WORD, QUAD WORD, TEN BYTES)

Brief

Format:

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

The DEFINE directives are entered in abbreviated form:

DB = BYTE
DW = WORD
DD = DOUBLE WORD
DQ = QUAD WORD
DT = TEN BYTES

A label or variable name may be assigned to the location. The number of type units allocated equals the number of operands. The initial value of the location is set to the value of the operand. This value must respect the size of the defined storage. For example:

```
VAR_NAME  DB 255
VAR_NAME  DW START
          DD 'AB','CD'
```

Details

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

If the optional varname is entered, the DEFINE directives define the name as a variable. If varname has a colon, it becomes a NEAR label instead of a variable. (Also see "Labels", Page 10.29 and "Variables" on Page 10.32.)

The DEFINE directives allocate memory in units specified by the second letter of the directive (each define directive may allocate one or more of its units at a time):

DB allocates one byte (8 bits)
DW allocates one word (2 bytes)
DD allocates two words (4 bytes)
DQ allocates four words (8 bytes)
DT allocates ten bytes

<exp> may be one or more of the following:

1. A constant expression.

2. The character ? for indeterminate initialization. Usually the ? is used to reserve space without placing any particular value into it. (It is the equivalent of the DS pseudo-op in MACRO-80).

3. An address expression (for DW and DD only).

4. An ASCII string (longer than 2 characters for DB only).

5. <exp>DUP(?)

   When this type of expression is the only argument to a define directive, the define directive produces an uninitialized data block. This expression with the ? instead of a value results in a smaller object file because only the segment offset is changed to reserve space.

6. <exp> DUP<exp>[...]

   This expression, like item five, produces a data block, but it is initialized with the value of the second <exp>. The first <exp> must be a constant greater than zero and must not be a forward reference.

Application

Example — Define Byte (DB):

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

Example — Define Word (DW):

```
ITEMS           DW      TABLE,TABLE+10,TABLE+20
SEGVAL          DW      0FFF0H
BSIZE           DW      4 * 128
LOCATION        DW      TOTAL +1
AREA            DW      100 DUP(?)
CLEARED         DW      50 DUP(0)
SERIES          DW      2 DUP(2,3 DUP(BSIZE))
                ;two words with the byte values
                ;2,BSIZE,BSIZE,BSIZE,2,BSIZE,BSIZE,BSIZE
DISTANCE        DW      START_TAB - END_TAB
                ;difference of two labels is a constant
```

Example — Define Double word (DD):

```
DBPTR           DD      TABLE           ;16-bit OFFSET, then 16-bit
                                        ;SEG base value
SEC_PER         DD      60*60*24        ;arithmetic is performed
                                        ;by the assembler
LIST            DD      'XY',2 DUP(?)
HIGH            DD      4294967295          ;maximum
FLOAT           DD      6.735E2             ;floating point
```

Example — Define Quad word (DQ)

```
LONG_REAL       DQ      3.141597            ;decimal makes it real
STRING          DQ      'AB'              ;no more than 2 characters
HIGH            DQ      18446744073709551615      ;maximum
LOW             DQ      -18446744073709551615     ;minimum
SPACER          DQ      2 DUP(?)            ;uninitialized data
FILLER          DQ      1 DUP(?,?)          ;initialized with
                                                ;indeterminate value
```

Example — Define Ten bytes (DT):

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

END

Brief

Format: END [<**exp**>]

END must be the last line of a source file. It may be followed by an expression which evaluates to the entry address of the program. If separate modules are linked, only the main module's END statement contains an expression.

Details

The END statement specifies the end of the program.

If <exp> is present, it is the start address of the program. If several modules are to be linked, only the main module may specify the start of the program with the END <exp> statement.

If <exp> is not present, then no start address is passed to LINK for that program or module.

Application

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

EQU

Format: <**name**> **EQU** <**exp**>

EQU permanently assigns the value of the expression following it to the name which precedes it. EQU is also used to create synonyms for opcodes. For example:

```
FOO EQU 256
CBD EQU AAD ;REDEFINED OPCODE
```

Details

EQU assigns the value of the expression to the name. If the expression is an external symbol, an error is generated. If name already has a value, an error is generated. If you want to be able to redefine a name in your program, use the equal sign (=) directive instead.

In many cases, EQU is used as a primitive text substitution, like a macro.

<exp> may be any one of the following:

1. A symbol, name becomes an alias for the symbol in the expression. Shown as an alias in the symbol table.

2. An instruction name. Shown as an opcode in the symbol table.

3. A valid expression. Shown as a number or L (label) in the symbol table.

4. Any other entry, including text, index references, segment prefix and operands. Shown as text in the symbol table.

Application

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

EQUAL SIGN (=)

Brief

Format: <**name**> = <**exp**>

The equal sign is similar to the EQU statement, except it permits the name to be redefined any number of times.

Details

<exp> must be a valid expression. It is shown as a Number or L (label) in the symbol table (same as <exp> type 3) under the EQU directive above.

The equal sign (=) allows you to set and to redefine symbols. The equal sign is like the EQU directive, except that you can redefine the symbol without generating an error. Redefinition may take place more than once, and redefinition may refer to a previous definition.

Application

```
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

Brief

Format: **EVEN**

EVEN adds one NOP, if necessary, to advance the program counter to an even value. For example:

```
        ;PC = ANY ODD VALUE
EVEN    ;PC = NEXT EVEN VALUE
EVEN    ;PC UNCHANGED
```

Details

The EVEN directive causes the program counter to go to an even boundary; that is, to an address that begins a word. If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP instruction so that the counter reaches an even boundary.

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

Application

Before: The PC points to 0019 hex (25 decimal)

```
EVEN
```

After: The PC points to 1A hex (26 decimal), 0019 hex now contains an NOP instruction.

EXTRN

Brief

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

EXTRN cues the assembler that a symbol will be used which is defined in a separate module. It takes two arguments separated by a colon. The first is the symbol name; the second is its type. The assembler assumes the symbol occurs within the same segment as the EXTRN directive unless an alternate segment is specified.

Details

name is a symbol that is defined in another module. Name must have been declared PUBLIC in the module where name is defined.

Type may be any one of the following, but must be a valid type for name:

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

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

ASSUME <seg-req>:SEG <name>

or an explicit segment prefix.

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

Application

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
CSEGB   ENDS
```

GROUP

Brief

Format: <**name**> **GROUP** <**seg-name**>**[...]**

GROUP is preceded by a group name and followed by one or more segment names. It causes the assembler to reference each of the segments to the same base address, which has the group name as its label. The entire group may be accessed through a single segment register. Group size is limited to 64k.

Details

The GROUP directive collects the segments named after GROUP (<seg-name>) under one name. The GROUP is used by LINK so it knows which segments should be loaded together. The order in which the segments are named here does not influence the order in which the segments are loaded; that is, handled by the CLASS designation of the SEGMENT directive, or by the order you name object modules in response to the LINK Object module prompt.

All segments in a GROUP must fit into 64K bytes of memory. The assembler does not check this at all, but leaves the checking to LINK.

<seg-name> may be one of the following:

1. A segment name, assigned by a SEGMENT directive. The name may be a forward reference.

2. An expression: either SEG <var> or SEG <label>

Both of these entries resolve themselves to a segment name.

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

1. As an immediate value:

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

   DGROUP is the paragraph address of the base of DGROUP.

2. In ASSUME statements:

```
ASSUME DS:DGROUP
```

   The DS register cannot be used to reach any symbol in any segment of the group.

3. As an operand prefix (for segment override):

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

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

Application

Example (using GROUP to combine segments):

In Module A:

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

In Module B:

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

INCLUDE

Brief

Format: **INCLUDE <filename>**

INCLUDE followed by a filename causes the assembler to read in and assemble the entire contents of a secondary source file at the current location. The letter C flags each line of the secondary file in the listing. Assembly aborts if the file is nonexistent or defective. Nesting of INCLUDE statements is permitted but discouraged.

Detail

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

The filename is any valid file specification for the operating system. If the device designation is other than the default, the source filename specification must include it. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the INCLUDE directive statement. When end-of-file is reached, assembly resumes with the next statement following the INCLUDE directive.

Nested includes are allowed (the file inserted with an INCLUDE statement may contain an INCLUDE directive). However, this is not a recommended practice with small systems because of the amount of memory required.

The file specified must exist. If the file is not found, an error is returned,and the assembly aborts.

On a MACRO-86 listing, the letter "C" is printed between the assembled code and the source line on each line assembled from an included file. See "Formats of Listings and Symbol Tables", Page 10.73, for a description of listing file formats.

Application

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

LABEL

Brief:

Format: <**name**> **LABEL** <**type**>

LABEL is functionally identical to the THIS directive. It is normally used to define a label or data area with more than one type. This permits access as an alternate type without the use of the PTR directive.

Details:

By using LABEL to define a <name>, you cause the assembler to associate the current segment offset with <name>.

The item is assigned a length of one.

<type> varies depending on the use of name.
<name> may be used for code or for data.

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

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

   If you want, you can define a NEAR label using the name form (the LABEL directive is not used in this case). If you are defining a BYTE or WORD NEAR label, you can place the name in front of a Define directive.

When using a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

Example — for code:

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

2. For data:

   Type may be BYTE, WORD, DWORD, <structure-name>, or <record-name>. When STRUC or RECORD name is used, the name is assigned the size of the structure or record.

Example — For Data:

```
BARRAY    LABEL    BYTE
ARRAY     DW       100 DUP(0)
          .
          .
          .
          ADD AL,BARRAY[99] ;ADD 100th byte to AL
          ADD AX,ARRAY[98] ;ADD 50th word to AX
```

By defining the array two ways, you can access entries either by byte or by word. Also, you can use this method for STRUC. It allows you to place your data in memory as a table, and to access it without the offset of the STRUC.

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

NAME

Brief

Format: **NAME** <**module-name**>

NAME followed by a module name gives the module a name which is used as a reference by LINK. Only the first six characters are significant. If the NAME directive is absent, a name is formed from the TITLE statement.

Details

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

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

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

1. A valid NAME directive statement

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

Application

```
NAME CURSOR
```

ORG

Brief

Format: **ORG <exp>**

ORG followed by an expression sets the location counter to the value of the expression within the current segment. An expression is invalid if it cannot be evaluated on pass one. For example:

```
ORG   0                    ; = START OF SEGMENT
ORG   $+20                 ;SKIP 20 BYTES
ORG   1                    ;FOLLOWED BY...
ORG   $+0FFFFH             ;WRAPS TO ORG 0
```

Details

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

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

Application

```
ORG   120H                 ;2-byte absolute value
                           ;maximum 0FFFFH
ORG   $+2                  ;skip two bytes
```

Example: -ORG to a boundary (conditional):

```
CSEG        SEGMENT      PAGE
BEGIN       =             $
            .
            .
            .
IF ($-BEGIN) MOD 256        ;if not already on
                            ;256 byte boundary
          ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
ENDIF
CSEG      ENDS
```

ENDIF

See "Conditional Directives", Page 10.135, for an explanation of conditional assembly.

PROC

Brief

Format: <**procname**> **PROC** **[NEAR]**
or **FAR**
.
.
.
RET
<**procname**> **ENDP**

PROC takes a preceding procedure name and an optional following NEAR or FAR attribute. NEAR is the default. Its use enhances structured coding by naming modules and groups of modules. A module may be entered in sequence or from a CALL or JUMP. Sequential procedures may be nested. The NEAR or FAR attribute simplifies coding by specifying the addressing mode required to locate a CALL or JUMP entered procedure. When declared in a PUBLIC directive, a procedure may be referenced externally.

Details

The default, if no operand is specified, is NEAR. Use FAR if:

The procedure name is an operating system entry point.

The procedure is called from code which has another ASSUME CS value.

Each PROC block should contain a RET statement.

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

The PROC directive, through the NEAR/FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL and RETs to generate a NEAR or a FAR RET. PROC is used for coding simplification so that you do not have to worry about NEAR or FAR for CALLs and RETs.

A NEAR CALL or RETURN changes the IP but not the CS register. A FAR CALL or RETURN changes both the IP and the CS registers.

Procedures are executed either in-line, from a JMP, or from a CALL.

PROCs may be nested, which means that they are put in line.

Combining the PUBLIC directive with a PROC statement (both NEAR and FAR), permits you to make external CALLs to the procedure or to make other external references to the procedure.

Application

```
                PUBLIC    FAR_NAME
FAR_NAME        PROC      FAR
                CALL      NEAR_NAME
                RET
FAR_NAME        ENDP
                PUBLIC    NEAR_NAME
NEAR_NAME       PROC      NEAR
                .
                .
                .
                RET
NEAR_NAME       ENDP
```

The second subroutine above can be called directly from a NEAR segment (that is, a segment addressable through the same CS and within 64K):

```
CALL NEAR_NAME
```

A FAR segment (that is, any other segment that is not a NEAR segment) must call to the first subroutine, which then calls the second; an indirect call:

```
CALL FAR_NAME
```

PUBLIC

Brief

Format: PUBLIC <**symbol**>**[...]**

PUBLIC followed by one or more names declares those names to be available to external modules. If the name is a symbol, it must evaluate to an integer no larger than two bytes. PUBLIC must be used within the module where its operands are defined.

Details

Place a PUBLIC directive statement in any module that contains symbols you want to use in other modules without defining the symbol again. PUBLIC makes the listed symbol(s), which are defined in the module where the PUBLIC statement appears, available for use by other modules to be linked with the module that defines the symbol(s). This information is passed to LINK.

The symbol may be a number, a variable, or a label (including PROC labels).

The symbol may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes (non integers or values that are greater than 0FFFFH).

Application

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

Example: — illegal PUBLIC:

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

.RADIX

Brief

Format: **.RADIX** <**exp**>

The .RADIX directive, followed by an expression which evaluates to two, eight, ten, or sixteen, sets the input radix to that base. The default radix is ten. Values following a DEFINE statement are not affected by the .RADIX directive. Unless they are decimal, their radix must be individually specified.

Details

The default input base (or radix) for all constants is decimal. The .RADIX directive permits you to change the input radix to any base in the range two to 16.

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

Example:

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

The two MOVs in this example are identical.

The .RADIX directive does not affect the generated code values placed in the .OBJ .LST, or .CRF output files.

The .RADIX directive does not affect the DD, DQ, or DT directives. Numeric values entered in the expression of these directives are always evaluated as decimal unless a data type suffix is appended to the value.

Application

```
.RADIX16
NUM_HAND   DT          773  ;773 = decimal
HOT_HAND   DQ          773Q ;773 = octal here only
COOL_HAND  DD          773H ;now 773 = hexadecimal
```

RECORD

Brief

Format:

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

RECORD defines a data type in which one or two bytes contain up to sixteen named fields. The WIDTH, MASK, and shift count functions isolate specific fields. RECORD takes a preceding recordname and three succeeding operands: the fieldname, which names a field, the width, which defines its size in bits. An optional expression initializes the value. The operands are repeated for each field. Forward references are not allowed. The first field defined occupies the highest order position in the RECORD. During assembly the entire RECORD is right-shifted, if necessary, to make the lowest order bit significant. For example:

```
ZOODATA RECORD COW :  3  =  21  ,  LION  :  2
```

COW is the higher order field. It is initialized to 21. Bits five to seven are unused. Allocate memory for a RECORD with the expression:

```
ANIMALS ZOODATA <,3> ;LION = 3
```

The expression in the angle brackets is optional but the brackets must be entered. Consecutive commas skip over fields to be ignored. Access a specific field this way:

```
MOV DL,ANIMALS       ;WHOLE RECORD IN DL
AND DL,MASK COWS     ;BITS OUTSIDE COWS = 0
MOV CL,COWS          ;SHIFT COUNT IN CL
SHR DL,CL            ;COWS IN LO ORDER BITS
MOV CL,WIDTH COWS    ;ACTUAL # OF BITS
```

Details

The fieldname is the name of the field. Width specifies the number of bits in the field defined by the fieldname. The expression contains the initial (or default) value for the field. Forward references are not allowed in a RECORD statement.

The fieldname becomes a value that can be used in expressions. When you use a fieldname in an expression, its value is the shift count to move the field to the far right. Using the MASK operator with the fieldname returns a bit mask for that field.

The width is a constant in the range one to 16 that specifies the number of bits contained in the field defined by the fieldname. The WIDTH operator returns this value. If the total width of all declared fields is larger than eight-bits, then the assembler uses two-bytes. Otherwise, only one-byte is used.

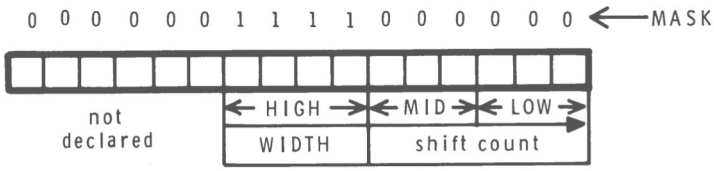
The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8-bits or exactly 16 bits, the entire record is right shifted so that the last bit of the last field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

Application

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

Initially, the bit map would be:

Total bits >8 means use a word; but total bits <16 means right shift, place undeclared bits at high end of word. Thus:



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

For example:

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

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

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

or

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

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

The recordname is the name used as a label for the RECORD directive.

The exp (both forms) contains the values you want placed into the fields of the record. In the latter case, the parentheses and angle brackets are required only around the second exp (following DUP). If [exp] is left blank, either the default value applies (the value given in the original record definition), or the value is indeterminate (when not initialized in the original record definition). For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields.

For example:

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

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

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

```
recordname<[value[....]]>
```

The value entry is optional. The angle brackets must be coded as shown, even if the optional values are not given. A value entry is the value to be placed into a field of the record. For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields, as shown above.

```
FOO  RECORD     HIGH:5,MID:3,LOW:3
     .
     .
     .
BAX  FOO        <> ;leave indeterminate here
JANE FOO        10DUP(<16.7>) ;HIGH=16,MID=7,
                ;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

Brief

Format:

```
<segname>  SEGMENT  [<align>]  [<combine>]  [< 'class'>]
                    .
                    .
                    .
<segname>  ENDS
```

SEGMENT and ENDS define the beginning and end of a program segment. Each is preceded by the same segment name. SEGMENT takes up to three optional operands which select memory map parameters. Code segments may be nested. They may not partially overlap. The assembler treats the nested segment as if it followed the macro segment.

Details

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

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

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

Combine may be PUBLIC, COMMON, AT <exp>, STACK, MEMORY, or no entry (which defaults to not combinable, called Private in the LINK chapter).

The class name is used to group segments at link time.

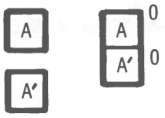
All three operands are passed to LINK.

The *alignment* tells the linker on what kind of boundary you want the segment to begin. The first address of the segment for each alignment type is:

PAGE — address is xxx00H (low byte is 0)
PARA — address is xxxx0H (low nibble is 0) bit map — X X X X 0 0 0 0
WORD — address is xxxxeH (e=even number;low bit is 0)
BYTE — address is xxxxxH (place anywhere)

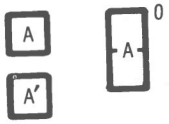
The *combine* type tells LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or Private)



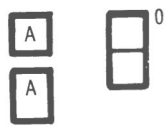
Private segments are loaded separately and remain separate. They may be physically continuous but not logically, even if the segments have the same name. Each private segment has its own base address.

Public and Stack



Public segments of the same name and class name are loaded continuously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine type stack is treated the same as public. However, the stack pointer is set to the first address of the first stack segment. LINK requires at least one stack segment.)

Common



Common segments of the same name and class name are loaded overlapping 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.

Ostensibly, the memory combine type causes the segment(s) to be placed as the highest segments in memory. The first memory combinable segment encounter is placed as the highest segment in memory. Subsequent segments are treated the same as common segments.

NOTE: This feature is not supported by LINK. LINK treats Memory segments the same as public segments.

AT <exp>

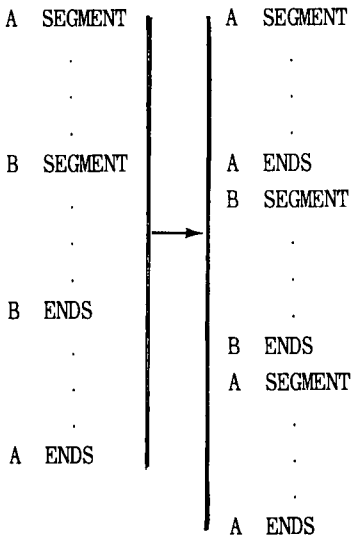
The segment is placed at the PARAGRAPH address specified in the expression. The AT type may not be used to force loading at fixed addresses. Rather, the AT combine type permits labels and variables to be defined at fixed offsets within fixed areas of storage, such as ROM or the vector space in low memory.

NOTE: This restriction is imposed by LINK and Z-DOS.

Class names must be enclosed in quotation marks. Class names may be any legal name.

Segment definitions may be nested. When segments are nested, the assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, the assembler takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. Overlapping segments are not permitted.

Application



The following arrangement is not allowed:

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

Example:

In Module A:

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

In Module B:

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

STRUC

Brief

Format:

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

STRUC and ENDS define the beginning and end of a structure data type. Each directive is preceded by the same structurename. Within the structure block, any number of DEFINE statements describe the separate fields. These DEFINE statements plus their associated names, operands and comments are the only legal entries. An optional preceding name serves as a fieldname within the structure. The operand(s) initialize its value. A field with more than one operand cannot be overridden in the allocation statement. Create a structure like this:

```
FLOWERS     STRUC
            DW ROSE,TULIP,DAISY
INDEX       DB 0,0,255
INDEX2      DB 16
FLOWERS     ENDS
```

Allocate memory to a structure like this:

```
GARDEN  FLOWERS<,,32>;OPTIONAL OVERRIDE
```

Only INDEX2 is overrideable. Access a structure like this:

```
MOV AL,GARDEN.INDEX2
```

Details

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

Inside the STRUC/ENDS block, the DEFINE directives (DB, DW, DD, DQ, DT) may be used to allocate space. The DEFINE directives and comments set off by semicolons (;) may be used to allocate space. The DEFINE directives and comments set off by semicolons (;) are the only statement entries allowed inside a STRUC block.

Any label on a DEFINE directive inside a STRUC/ENDS block becomes a fieldname of the structures. (This is how structure fieldnames are defined.) Initial values given to fieldnames in the STRUC/ENDS block are default values for the various fields. These values of the fields are one of two types: overrideable or not overrideable. A simple field, a field with only one entry (but not a DUP expression), is overrideable. A multiple field, a field with more than one entry is not overrideable.

For example:

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

If the expression following the DEFINE directive contains a string, it may be overridden by another string. However, if the overriding string is shorter than the initial string, the assembler pads it with spaces. If the overriding string is longer, the assembler truncates the extra characters.

Usually, structure fields are used as operands in some expression. The format for a reference to a structure field is:

<variable>.<field>

variable represents an anonymous variable, usually set up when the structure is allocated. To allocate a structure, use the structure name as a directive with a label (the anonymous variable of a structure reference) and any override values in angle brackets:

```
FOO STRUC
    .
    .
    .
FOO ENDS

GOO FOO <,7,,'JOE'>
```

The .<field> represents a label given to a DEFINE directive inside a STRUC/ENDS block (the period must be coded as shown). The value of the field is the offset within the addressed structure.

Application

To define a structure:

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

The DEFINE directives in this example define the fields of the structure and the order corresponds to the order values are given in the initialization list when the structure is allocated. Every DEFINE directive statement line inside a STRUC block defines a field, whether or not the field is named.

To allocate the structure:

```
DBAREA    S    <,,7,'ANDY'>                ;overrides 3rd and 5th
                                           ;fields only
```

To refer to a structure:

```
MOV         AL,[BX].FIELD3
MOV         AL,DBAREA.FIELD3
```

Conditional Directives

Brief

Format:

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

Conditional directives permit a block of code to be assembled only if a test set up by the programmer returns true. The optional ELSE clause specifies an alternate block to be assembled if the test returns false. Conditionals may be nested to 255 levels. All arguments must be known on pass one. All expressions must contain only predefined values. An expression, which must evaluate to an absolute, is true if it evaluates to non-zero. The structural format is:

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

Details

Conditional directives allow you to design blocks of code which test for specific conditions then proceed accordingly.

All conditionals follow the format:

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

Each IFxxxx must have a matching ENDIF to terminate the conditional. Otherwise, an "unterminated conditional" message is generated at the end of each pass. An ENDIF without a matching IF causes error code 8, "Not in conditional block".

Each conditional block may include the optional ELSE directive, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional causes error code 7, "Already had ELSE clause".

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass one to avoid Phase errors and incorrect evaluation. For IF and IFE the expression must involve values which were previously defined, and the expression must be Absolute. If the name is defined after an IFDEF or IFNDEF, pass one considers the name to be undefined, but considers it defined on pass two.

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

IF <**exp**> If the expression evaluates to nonzero, the statements within the conditional block are assembled.

IFE <**exp**> If the expression evaluates to 0, the statements in the conditional block are assembled.

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

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

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

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

IFB <**arg**> The angle brackets around the argument are required.

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

IFB (and IFNB) are normally used inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy is replaced by a parameter passed by the macro call. If the macro call does not specify a parameter to replace the dummy following IFB, the expression is blank, and the block is assembled. (IFNB is the opposite case.)

IFNB <**arg**> The angle brackets around the argument are required.

If the argument is not blank, the statements in the conditional block are assembled.

IFNB (and IFB) are normally used inside macro blocks. The expression following the IFNB directive is typically a dummy symbol. When the macro is called, the dummy is replaced by a parameter passed by the macro call. If the macro call specifies a parameter to replace the dummy following IFNB, the expression is not blank, and the block is assembled. (IFB is the opposite case.)

IFIDN <**arg1**>,<**arg2**> The angle brackets around the arguments are required.

If the string in argument one is identical to the string the argument two, the statements in the conditional block are assembled.

IFIDN (and IFDIF) are normally used inside macro blocks. The expressions following the IFIDN directive are typically two dummy symbols. When the macro is called, the dummies are replaced by parameters passed by the macro call. If the macro call specifies two identical parameters to replace the dummies, the block is assembled. (IFDIF is the opposite case.)

IFDIF <**arg1**>,<**arg2**> The angle brackets around the two arguments are required.

If the string in argument one is different from the string in argument two, the statements in the conditional block are assembled.

IFDIF (and IFIDN) are normally used inside macro blocks. The expressions following the IFDIF directive are typically two dummy symbols. When the macro is called, the dummies are replaced by parameters passed by the macro call. If the macro call specifies two different parameters to replace the dummies, the block is assembled. (IFIDN is the opposite case.)

ELSE The ELSE directive allows you to generate alternate code when the opposite condition exists. May be used with any of the conditional directives. Only one ELSE is allowed for each IFxxxx conditional directive. ELSE takes no expression.

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

Macro Directives

Brief

Source code blocks used repeatedly within a program can be entered once as a "macro definition." A one line "macro call" causes the assembler to insert the code at any desired point in the program. Nesting of macro calls is limited only by memory size. These are the macro directives of MACRO-86:

MACRO ENDM EXITM LOCAL PURGE REPT IRP IRPC

These are the special macro operators:

& ;; ! %

Details

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

The macro directives of the MACRO-86 macro assembler include:

macro definition:
    MACRO

termination:
    ENDM
    EXITM

unique symbols within macro blocks:
    LOCAL

undefine a macro:

PURGE

repetitions:

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

The macro directives also include some special macro operators:

& ;; ! %

Macro Definitions

Brief

Format: <**name**> **MACRO** [<**dummy**>...]

.
.
.

ENDM

MACRO is the first line of a macro definition. It is preceded by a macro name and followed by an optional chain of dummy arguments enclosed in angle brackets and separated by commas. When the macro is called, the calling statement will replace all occurrences of dummy arguments inside the macro with actual values. ENDM is the last line of a macro definition. In parsing a source line, the assembler checks the macro definition table first. Any reserved word can be redefined as a macro. This necessitates caution in choosing macro names.

Details

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

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

A dummy is formed as any other name is formed. A dummy is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. You should include all dummies used inside the macro block on this line. The number of dummies 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 as any list of symbol names.

NOTE: A dummy is always recognized exclusively as a dummy. Even if a register name (such as AX or BH) is used as a dummy, it is replaced by a parameter during expansion.

One alternative is to list no dummies:

```
<name> MACRO
```

This type of macro block allows you to call the block repeatedly, even if you do not want or need to pass parameters to the block. In this case, the block will not contain any dummies.

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

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable, or symbol. When MACRO-86 evaluates a statement, it first looks at the macro table it builds during pass one. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: MACRO-86 evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, MACRO-86 assembles the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., -MACRO, MACROS, and so on.

Calling a Macro

Brief

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

Call a macro by entering its name. Pass arguments by enclosing them in angle brackets and separating them with commas. If you pass more arguments than specified in the definition, the extras are ignored. If you pass too few, the unused dummies become nulls. Multiple values separated by commas within angle brackets are passed as a single argument. A plus sign flags each line of a listing which was generated by a macro call.

Details

To use a macro, enter a macro call statement:

Name is the name of the MACRO block. A parameter replaces a dummy on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, the assembler passes all the items inside the angle brackets as a single parameter. For example:

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

passes five parameters to the macro, but:

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

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummies in the MACRO definition. If there are more parameters than dummies, the extras are ignored. If there are fewer, the extra dummies are made null. The assembled code includes the macro block after each macro call statement.

Application

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

If you then enter a macro call statement:

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

assembly generates the statements:

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

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

End Macro

Brief

Format: **ENDM**

ENDM is the last line of a macro definition. It also terminates REPT, IRP, and IRPC blocks.

Details

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

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM directive. Otherwise, the error message is generated. An unmatched ENDM also causes an error.

If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Exit Macro

Brief

Format: **EXITM**

EXITM terminates a macro expansion from the point where it is encountered. It is normally used after a conditional directive. If ENDM is executed within a nested macro, assembly continues at the next higher level.

Details

The EXITM directive is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional directive.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

Application

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

LOCAL

Brief

Format: **LOCAL <dummy>[,<dummy>...]**

LOCAL creates unique names within each expansion of a macro. Without it, names used within a macro definition (except =) generate a "symbol is muti-defined" error. LOCAL must be the second line of a macro definition. It is followed by dummy arguments enclosed in angle brackets and separated by commas. A unique name of the form ??nnnn replaces each dummy name used in the definition whenever the macro is called. Avoid using names of the form ??nnnn, since this function might attempt to "steal" them, causing an error. For example:

```
    FOO   MACRO                ;DEFINITION
          LOCAL <A>
      A:  MOV AL,AH
          ENDM

          FOO                  ;EXPANSION
+??0000:  MOV AL,AH
```

Details

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

When LOCAL is executed, the assembler creates a unique symbol for each occurrence of the dummy in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ??0000 to ??FFFF. You should avoid the form ??nnnn for use as one of your own symbols.

Application

```
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

Brief

FORMAT: **PURGE <macro-name>[...]**

PURGE deletes a macro definition. It is followed by the name of one or more macros enclosed in angle brackets and separated by commas. It is not necessary to PURGE a macro before defining a new one with the same name. The deletion in this case is automatic . A PURGE statement may occur anywhere in a program.

Details

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

PURGE provides three benefits:

1. It frees text space of the macro body.

2. It returns any instruction mnemonics or directives that were redefined by macros to their original function.

3. It allows you to "edit" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method allows you to use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you do not plan to use in this program.

   It is not necessary to PURGE a macro before redefining it. Simply place another MACRO statement in your program, reusing the macro name.

Application

```
INCLUDE   MACRO.LIB
PURGE     MAC1                          ;frees space used by MAC1
```

REPEAT DIRECTIVES

Brief

REPEAT permits a block of code to be assembled a specific number of times. Unlike a MACRO, the parameters are built into the block and remain the same on each pass. REPEAT directives may be used inside or outside a MACRO. ENDM ends the block.

Details

The directives in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the REPEAT directives and MACRO directives are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

2. MACRO allows parameters to be passed to the MACRO block when a MACRO is called; hence, parameters can be changed.

REPEAT directive parameters must be assigned as a part of the code block. If the parameters are known in advance and are not going to change, and if the repetition is to be performed for every program execution, then REPEAT directives are convenient. With the MACRO directive, you must call in the MACRO each time it is needed.

Note that each REPEAT directive must be matched with the ENDM directive to terminate the repeat block.

REPT

Brief

Format: <**exp**>
.
.
.
ENDM

REPT is followed by an expression which evaluates to the number of repetitions. The code block follows REPT and ends with ENDM. External symbols are illegal in the expression. For example:

```
X = 0                ;DEFINITION
REPT 10
X=X+1
DB X
ENDM

0000 01 + DB X       ;ASSEMBLED CODE
0001 02 + DB X
0002 03 + DB X       ;ETC HRU 0A
```

Details

Repeat block of statements between REPT and ENDM <exp> times. The expression is evaluated as a 16-bit unsigned number. If the expression contains an external symbol or undefined operands, an error is generated.

Application

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

Assembles as:

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

IRP INDEFINITE REPEAT

Brief

Format:

IRP <dummy>,<parameters (inside angle brackets)>

.
.
.

ENDM

IRP takes two arguments. The second argument is enclosed in angle brackets and separated from the first argument by a comma. The first is a dummy variable. The second is a series of values delimited by commas. A block of code under the IRP statement is assembled once for each value in the second argument. Either argument may be passed from a macro call, since the angle brackets are stripped before it is passed. ENDM ends the block. This definition produces the same code shown for REPT:

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

Details

Parameters must be enclosed in brackets. Parameters may be any legal symbol, string, numeric, or other character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of dummy in the block. If a parameter is null (i.e.,<>), the block is processed once with a null parameter.

Application

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

This example generates the same bytes (DB 1 — DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

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

When the macro call statement

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

is assembled, the macro expansion becomes:

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

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

IRPC INDEFINITE REPEAT CHARACTER

Brief

Format: **IRPC <dummy>,<string>**
.
.
.
ENDM

IRPC functions similar to IRP, except the second argument is a string enclosed in angle brackets. The block is assembled once for each character in the string. The corresponding character replaces the dummy variable on each pass. For example:

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

Details

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of the dummy in the block.

Application

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

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

SPECIAL MACRO OPERATORS

Brief

The following special macro operators provide additional capabilities within macro definitions:

& < > ;; ! %

Details

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

& Ampersand concatenates text or symbols. (The & may not be used in a macro call statement.) A dummy parameter in a quoted string is not substituted in expansion unless preceded immediately by &. To form a symbol from text and a dummy, put & between them.

For example:

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

The call ERRGEN A then generates:

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

In MACRO-86, unlike MACRO-80, the ampersand does not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, extra ampersands may be needed. You need to supply as many ampersands as there are levels of nesting.

For example:

Correct form

```
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
        ENDM
        ENDM
```

1. MACRO build, find dummies and change to di

When called, for example, by FOO BAZ, the expansion would be: (shown correctly in the left column, incorrectly in the right)

```
        IRP     Z,<1,2,3>          IRP     Z,<1,2,3>
di&Z    DB      Z           diZ    DB      Z
        ENDM                       ENDM
```

2. MACRO expansion, substitute parameter text for di

```
        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 di

```
BAZ&di    DB      di          BAZZ DB     di
```

4. IRP expansion, substitute parameter text for di

```
BAZ1    DB    1       BAZZ    DB    1
BAZ2    DB    2       BAZZ    DB    2 ;here it's an error,
BAZ3    DB    3       BAZZ    DB    3 ;multi-defined symbol
```

<text> Angle brackets cause MACRO-86 to treat the text between the angle brackets as a single literal. Placing either the parameters to a macro call or the list of parameters following the IRP directive inside angle brackets causes two results:

1. All text within the angle brackets is seen as a single parameter, even if commas are used.

2. Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows. One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you need to supply as many sets of angle brackets around parameters as there are levels of nesting.

;; In a macro or repeat block, a comment preceded by two semicolons is not saved as a part of the expansion.

The default listing condition for macros is .XALL (see "Listing Directives", Page 10.153). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments inside macro and repeat blocks are saved and listed. This can be the cause of an out of memory error. To avoid this error, place double semicolons before comments inside macro and repeat blocks, unless you specifically want a comment to be retained.

! An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must evaluate to an absolute (non-relocatable) constant.

Application

```
PRINTE  MACRO   MSG,N
        %OUT    *MSG,N*
        ENDM
SYM1    EQU     100
SYM2    EQU     200
        PRINTE  <SYM1 + SYM2 =>.,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the string (SYM1 + SYM2) to be substituted for the dummy N. The result would be:

```
%OUT    *SYM1 + SYM2 = (SYM1 + SYM2)    *
```

When the % is placed in front of the parameter, the assembler generates:

```
%OUT    * SYM1 + SYM2 = 300 *
```

Assembling a Source File

Listing Directives

Brief

These directives are used to adjust the parameters of the assembler listing:

PAGE	**TITLE**	**SUBTTL**	**%OUT**
.LIST	**.XLIST**	**.SFCOND**	**.LFCOND**
.TFCOND	**.XALL**	**.LALL**	**.SALL**
.CREF	**.XCREF**		

Details

Listing directives perform two general functions: format control and listing control. Format control directives allow the programmer to insert page breaks and direct page headings. Listing control directives turn on and off the listing of all or part of the assembled file.

PAGE

Brief

Format: **PAGE [<length>][,<width>]**
PAGE [+]

Details

PAGE with no arguments or with the optional [+] argument causes the assembler to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

The PAGE directive with either the length or width arguments does not start a new listing page.

The value of the length, if included, becomes the new page length (measured in lines per page) and must be in the range 10 to 255. The default page length is 50 lines per page.

The value of the width, if included, becomes the new page width (measure in characters) and must be in the range of 60 to 132. The default page width is 80 characters.

The plus sign (+) increments the major page number and resets the minor page number to one. Page numbers are in the form major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Application

```
        .
        .
        .
PAGE +          ;increment Major, set minor to 1
        .
        .
        .
PAGE 58,60              ;page length=58 lines,
                        ;width=60 characters
```

TITLE

Brief

Format: **TITLE** <**text**>

Details

TITLE specifies a title to be listed on the first line of each page. The <text> may be up to 60 characters long. If more than one TITLE is given, an error results. The first six characters of the title, if legal, are used as the module name, unless a NAME directive is used.

Application

```
TITLE PROG1 - 1st Program
  .
  .
  .
```

If the NAME directive is not used, the module name is now `PROG1-1st Program`. This title text appears at the top of every page of the listing.

SUBTTL

Brief

Format: **SUBTTL** <**text**>

Details

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The text is truncated after 60 characters.

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the text from the previous SUBTTL with the text from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for text.

Application

```
SUBTTL SPECIAL I/O ROUTINE
   .
   .
   .
SUBTTL
   .
   .
   .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

%OUT

Brief

Format: **%OUT** <**text**>

Details

The text is listed on the terminal during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

%OUT outputs on both passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See "Conditional Directives," Page 10.135, for descriptions of the IF1 and IF2 directives.

Application

```
%OUT *Assembly half done*
```

The assembler sends the delimited message to the terminal screen when the line is processed.

```
IF1
%OUT *Pass 1 started*
ENDIF

IF2
%OUT *Pass 2 started*
ENDIF
```

.LIST
.XLIST

Brief

Format: **.LIST**
.XLIST

Details

.LIST lists all lines with their code (the default condition).

.XLIST suppresses all listing.

If you specify a listing file following the Listing prompt, a listing file with all the source statements included is listed.

When .XLIST is encountered in the source file, source and object code are not listed. .XLIST remains in effect until a .LIST is encountered.

Application

```
        .
        .
        .
    .XLIST                  ;listing suspended here, ".XLIST" is not printed
        .
        .
        .
    .LIST                   ;listing resumes here, ".LIST" is printed
```

.SFCOND suppresses portions of the listing containing conditional expressions that evaluate as false.

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

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch when running the assembler. When /X is used, .TFCOND causes false conditionals to list. When /X is not used, .TFCOND suppresses false conditionals.

.XALL is the default.

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

.LALL 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

Brief

Format: **.CREF**
.XCREF [<variable list>]

Details

.CREF is the default condition. .CREF remains in effect until MACRO-86 encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until MACRO-86 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Use .CREF to restart the creation of a cross reference file after using the .XCREF directive.

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

Neither .CREF nor .XCREF without arguments takes effect unless you specify a cross reference file when you run the assembler. .XCREF variable list suppresses the variables from the symbol table listing regardless of the creation of a cross reference file.

Example:

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

Application

Use assembler directives to specify the actions or assumptions you want MACRO-86 to make in processing your source code. Their use can greatly simplify the task of assembly language programming. Macro and conditional source modules eliminate repetitious code entry. Listing directives produce a clear and readable printout. Data definition directives allow you to name and structure storage areas consistent with the way they will be accessed. All of these features are tools designed to enhance development of your assembly language programs.

MACRO-86

Assembling a Source File

INTRODUCTION

Brief:

Assembling with MACRO-86 requires two types of commands: a command to invoke MACRO-86 and answers to command prompts. In addition, three switches control alternate MACRO-86 features. Usually, you enter all the commands to MACRO-86 on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in a batch file. Some command characters are provided to assist you while you enter assembler commands.

Details

Invoke MACRO-86 by entering MASM on the terminal. The assembler responds with a series of four queries requesting names for the .ASM, .OBJ, .LST, and .CREF files. You may override the queries by entering responses to all four, separated by commas, following MASM. There is no default for the first query. A source filename must be supplied. Unless an extension is entered, the assembler assumes it to be .ASM. The source filename becomes the default option for the .OBJ file. The default for the last two queries is not to produce a file. A RETURN selects the default for a query. In the group method, adjacent commas perform the same function. After the first query, a semicolon followed by a return may be entered at any time to terminate the query process. Defaults are then assigned to all remaining filenames. Control-C aborts the assembler at any time. The Z-DOS prompt indicates the end of assembly.

Invoking MACRO-86

MACRO-86 may be invoked two ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke MACRO-86.

Summary of Methods to invoke MACRO-86:

Method 1 **MASM**

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

METHOD ONE: MASM

Enter:

MASM

MACRO-86 loads into memory. Then, MACRO-86 returns a series of four text prompts that appear one at a time. You answer the prompts as commands to MACRO-86 to perform specific tasks.

At the end of each line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, the MACRO-86 default is to not perform the function described for the switches in the chart below.

The command prompts are summarized here. Following the summary of prompts is a summary of switches.

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

SWITCH	ACTION
/D	Produce a listing on both assembler passes.
/O	Show generated object code and offsets in octal radix on listing.
/X	Suppress the listing of false conditionals. Also used with the .TFCOND directive.

Command Switches Summary

Command Characters

MACRO-86 provides two command characters.

; Use a single semicolon (;), followed immediately by a RETURN, at any time after responding to the first prompt (from Source filename on) to select default responses to the remaining prompts. This feature saves time and overrides the need to keep entering RETURNS.

NOTE: Once the semicolon has been entered, you can no longer respond to any of the prompts for that assembly. Therefore, do not use the semicolon to skip over some prompts. For this, use RETURN.

Example:

```
Source filename [.ASM]: FUN RETURN
Object filename [FUN.OBJ]: ; RETURN
```

The remaining prompts are skipped, and MACRO-86 uses the default values (including no listing file and no cross reference file).

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

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

This response produces the same files as the previous example.

CTRL-C Use CTRL-C at any time to abort the assembly. If you enter an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press **CTRL-C** to exit MACRO-86 then reinvoke MACRO-86 and start over. If the error has been typed and not entered, you may delete the erroneous characters, but for that line only.

METHOD TWO: MASM<FILENAMES></x>

Enter:

MASM <source>,<object>,<listing>,<cross-ref></x...>

MACRO-86 is loaded into memory. Then MACRO-86 immediately begins assembly. The entries following MASM are responses to the command prompts. The entry fields for the different prompts must be separated by commas where <source> is the source filename; where <object> is the name of the file to receive the relocatable output; where <listing> is the name of the file to receive the listing; where <crossref> is the name of the file to receive the cross reference output; and where </x...> are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <cross-ref>, as shown.

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

Example:

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

This example causes MACRO-86 to be loaded, then causes the source file FUN.ASM to be assembled. MACRO-86 then outputs the relocatable object code to a file named FUN.OBJ (default caused by two commas in a row), creates a listing file named FUN.LST and a cross reference filenamed FUN.CRF. If names were not listed for listing and cross reference, these files would not be created. If listing file switches are given but no filename, the switches are ignored.

MACRO-86 Command Prompts

MACRO-86 is commanded by entering responses to four text prompts. When you have entered a response to the current prompt, the next appears. When the last prompt has been answered, MACRO-86 begins assembly automatically without further command. When assembly is finished, MACRO-86 exits to the operating system.

MACRO-86 prompts you for the names of source, object, listing, and cross reference files.

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

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

You may not enter only a filename extension.

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

Enter the filename of your source program. MACRO-86 assumes by default that the filename extension is .ASM, as shown in square brackets in the prompt text. If your source program has any other filename extension, you must enter it along with the filename. Otherwise, the extension may be omitted.

```
Object filename [source.OBJ]:
```

Enter the filename you want to receive the generated object code. If you simply press the RETURN key when this prompt appears, the object file is given the same name as the source file, but with the filename extension .OBJ. If you want to change only the filename but keep the .OBJ extension, enter the filename only. To change the extension only, you must enter both the filename and the extension. If you specify a drive other than the default drive for the source filename prompt, that specified drive name is not carried over into the default response for this prompt. If you want the object file to be on the same non-default drive as the source file, you must specify the drive name here.

Source listing [NUL.LST]:

Enter the name of the file, if any, that you want to receive the source listing. If you press the RETURN key, MACRO-86 does not produce this listing file. If you enter a filename only, the listing is created and placed in a file with the name you enter plus the filename extension .LST. You may also enter your own extension.

The source listing file contains a list of 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, if any, you want to receive the cross reference file. If you press only the RETURN key, MACRO-86 does not produce a cross reference file. If you enter a filename only, the cross reference file is created and placed in a file with the name you enter plus the filename extension .CRF. You may also enter your own extension.

The cross reference file is used as the source file for the CREF Cross Reference Facility. CREF converts this cross reference file into a cross reference listing, which you can use to aid you during program debugging.

The cross reference file contains a series of control symbols that identify records in the file. CREF uses these control symbols to create a listing that shows all occurrences of every symbol in your program. The occurrence that defines the symbol is also identified.

MACRO-86 Command Switches

Brief

/D, /O, and /X switches may be entered at the end of any query line containing a filename. /D produces a listing on both passes. It is useful for locating phase errors. On pass 1, all forward references generate error messages. /O causes octal listings of object code and relative offsets. It has no effect on the .OBJ file. /X suppresses the listing of false conditionals. .LFCOND and .SFCOND directives have a higher priority. The .TFCOND directive alters the effect of /X.

Details

The three switches control alternate assembler functions. Switches must be entered at the end of a prompt response, regardless of which method is used to invoke MACRO-86. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is entered at the end of one response, each switch must be preceded by the slash mark (/). You may not enter only a switch as a response to a command prompt.

Switch	Function
/D	Produce a source listing on both assembler passes. The listing, when compared, shows where in the program phase errors occur and, possibly, gives you a clue as to why the errors occur. The /D switch does not take effect unless you command MACRO-86 to create a source listing (enter a filename in response to the source listing command prompt).
/O	Output the listing file in octal radix. The generated code and the offsets shown on the listing are all given in octal. The actual code in the object file is the same as if the /O switch were not given. The /O switch affects only the listing file.

/X Suppress the listing of false conditionals. If your program contains conditional blocks, the listing file shows the source statement but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not generate code, use the /X switch to suppress the blocks that evaluate false from your listing.

The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives.

If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, the second restores listing of false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When MACRO-86 encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing.

Of course, the /X switch has no effect if no listing is created.

The following chart illustrates the various effects of the conditional listing directives in combination with the /X switch. ON = false condition listed; OFF = false not listed.

PSEUDO-OP	NO/X	/X
(none)	OFF	ON
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF

NOTE: True conditions are always listed and assembled.

Formats of Listings and Symbol Tables

Brief

The assembler listing is divided into two parts. In part one, each line contains: 1) a line number, unless .CREF is suppressed; 2) the offset address, if the line generates code; 3) the object code; 4) a + or a C for a macro or INCLUDE block; 5) the source code. Part two contains an error message count plus the name and description of all macros, structures, records, segments, groups, and symbols.

Details

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

The first part of the listing shows:

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

The second part of the listing shows:

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

PROGRAM LISTING

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

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

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

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

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

R	= linker resolves entry to left of R
E	= External
----	= segment name, group name, or segment variable used in MOV AX,<---->, DD <---->, JMP <---->, and so on.
=	= statement has an EQU or = directive
nn:	= statement contains a segment override
nn/	= REPxx or LOCK prefix instruction. Example:

003C F3/ A5 REP MOVSW ;move DS:SI to ES:DI until CX=0

[xx]	= DUP expression;xx is the value in parentheses following the DUP; for example: DUP(?) places ?? where xx is shown here
+	= line comes from a macro expansion
C	= line comes from file name in INCLUDE directive

Summary of Listing Symbols

```
The Microsoft MACRO-86 MACRO Assembler    mm-dd-yy    PAGEP-P

ENTX    PASCAL entry for initializing programs

                                                    ;
0000                                STACK    SEGMENT  WORD STACK  'STACK'
=0000                               HEAPbeg  EQU      THIS BYTE
                                                    ;Base of heap before init

done
0000     14 [                       DB       20 DUP(?)
            ??
              ]
= 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

000C  26:  8B 1E 0002                             MOV      BX,ES:2  ;Highest Paragraph
```

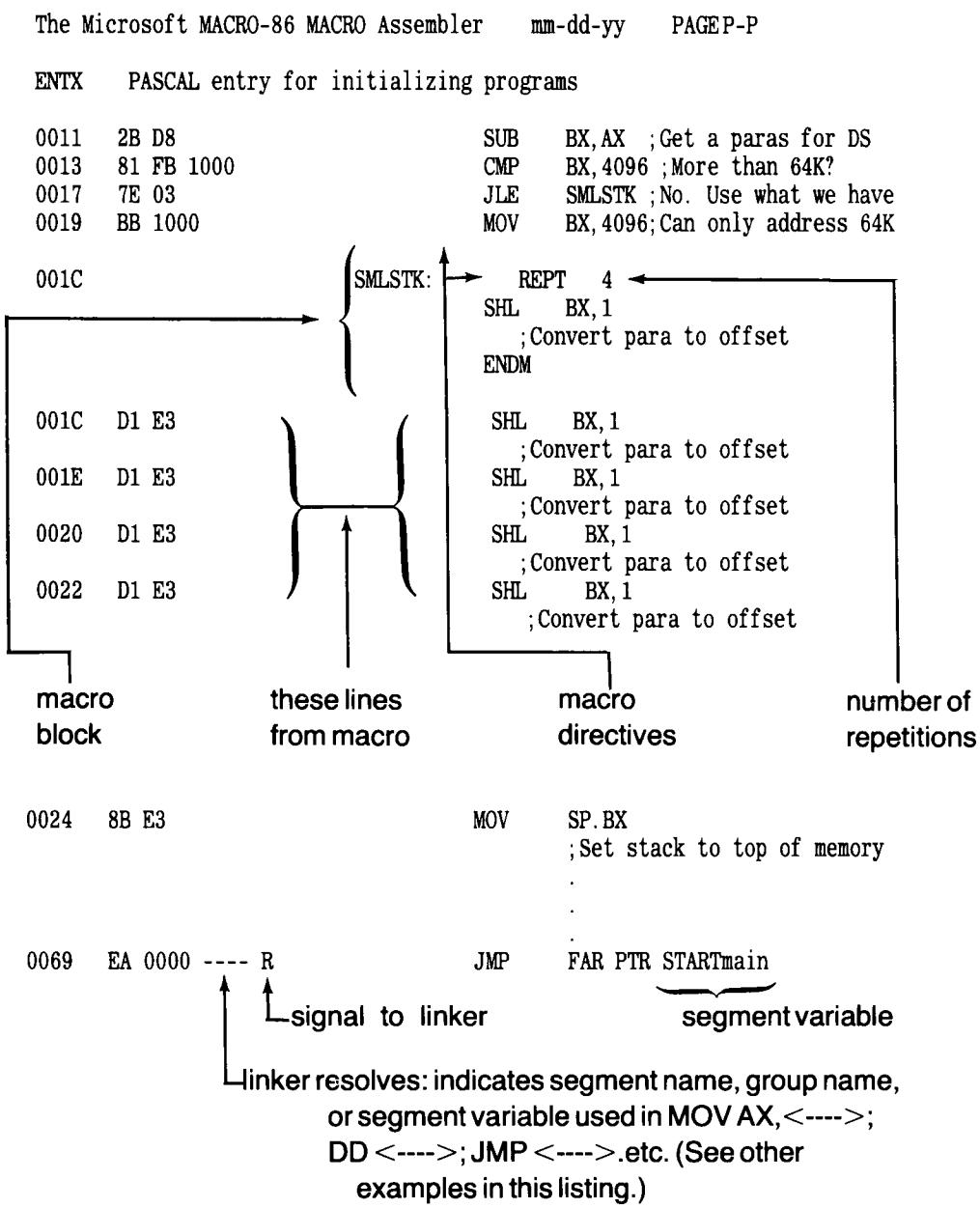
Indicates EQU or = directive

shows value in parentheses

Indicates DUP expression

offset | generated code | name | action | expression | comment

segment override


The Microsoft MACRO-86 MACRO Assembler mm-dd-yy PAGEP-P
ENTX PASCAL entry for initializing programs
0011 2B D8 SUB BX,AX ;Get a paras for DS
0013 81 FB 1000 CMP BX,4096 ;More than 64K?
0017 7E 03 JLE SMLSTK ;No. Use what we have
0019 BB 1000 MOV BX,4096;Can only address 64K
001C SMLSTK:
REPT 4
SHL BX,1
;Convert para to offset
ENDM
001C D1 E3 SHL BX,1
;Convert para to offset
001E D1 E3 SHL BX,1
;Convert para to offset
0020 D1 E3 SHL BX,1
;Convert para to offset
0022 D1 E3 SHL BX,1
;Convert para to offset
macro block
these lines from macro
macro directives
number of repetitions
0024 8B E3 MOV SP.BX
;Set stack to top of memory
0069 EA 0000 ---- R JMP FAR PTR STARTmain
signal to linker
segment variable
linker resolves: indicates segment name, group name, or segment variable used in MOV AX,<---->; DD <---->; JMP <---->.etc. (See other examples in this listing.)

```
0006E                   BEGXQQ      ENDP
                                    .
                                    .
                                    .
007E                    MAINSTARTUP         ENDS
0000                    ENTXCM      SEGMENT WORD 'CODE'
                                    ASSUME CS:ENTXCM
                                    PUBLIC ENDXQQ,DOSXQQ
```

```
The Microsoft MACRO-86 MACRO Assembler  mm-dd-yy     PAGEP-P

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 openfiles
000F  9A 0000 ---- E                     CALL   ENDUQQ
                                           ;file system termination
0014  C7 06 0020 R 0000                  MOV    DOSOF.0
```

offset

External symbol

linker signal; goes with number to left; shows DOSOFF is in segment

```
00 2E 0020 R                  STARTmain  ENDP
                                         .
                                         .
                                         .
0037                          ENTXCM     ENDS
                                         END    BEGXQQ
```

Differences Between Pass One Listing and Pass Two Listing

If you give the /D switch when you run MACRO-86 to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that assembled without reporting any errors. When the source file was reassembled using the /D switch, an error was produced on pass one, but not on pass two (which is when errors are usually reported).

Application

During Pass one 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 two this same instruction is fixed up and does not return an error.

```
0017  7E 03              JLE       SMLSTK   ;No, use what we have
0019  BB 1000            MOV       BX,4096  ;Can only address 64K
001C              SMLSTK: REPT      4
```

Notice that the JLE instruction's code now contain 03 instead of 00; a jump of three-bytes.

The same amount of code was produced during both passes, so there was no phase error. The only difference in this case is content, not of size.

Symbol Table Format

The symbol table portion of a listing separates all "symbols" into their respective categories, showing appropriate descriptive data. This data gives you an idea how your program is using various symbolic values. Use this information to help you debug.

Also, you can use a cross reference listing, produced by CREF, to help you locate uses of the various "symbols" in your program.

On the next page is a complete symbol table listing. Following the complete listing, sections from different symbol tables are shown with explanatory notes.

For all sections of symbol tables, this rule applies: if there are no symbolic values in your program for a particular category, the heading for the category is omitted from the symbol table listing. For example, if you did not use macros in your program, there is no macro section in the symbol table.

MACR

The Microsoft MACRO-86 MACRO Assembler Date PAGE Symbols-1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXMP1M.ASM)

Macros:

Name	Length
BIOSCALL	0002
DISPLAY	0005
DOSCALL	0002
KEYBOARD	0003
LOCATE	0003
SCROLL	0004

Structures and records:

Name	width	# fields	
	Shift	Width	Mask Initial
PARMLIST	001C	0004	
BUFSIZE	0000		
NAMESIZE	0001		
NAMETEXT	0002		
TERMINATOR	001B		

Segments and groups:

Name	Size	align	combine	class'
CSEG	0044	PARA	PUBLIC	'CODE'
STACK	0200	PARA	STACK	'STACK'
WORKAREA	0031	PARA	PUBLIC	'DATA'

Symbols:

Name	Type	Value	Attr	
CLS	N PROC	0036	CSEG	Length =000E
MAXCHAR	Number	0019		
MESSG	L BYTE	001C	WORKAREA	
PARMS	L 001C	0000	WORKAREA	
RECEIVR	L FAR	0000		External
START	F PROC	0000	CSEG	Length =0036

Warning Errors	Severe Errors
0	0

```
Macros:
                Name                    Length
BIOSCALL. . . . . .                     0002
DISPLAY . . . . . .                     0005
DOSCALL . . . . . .                     0002
KEYBOARD. . . . . .                     0003
LOCATE. . . . . .                       0003
SCROLL. . . . . .                       0004
```

number of 32 byte blocks macro occupies in memory

names of macros

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

Structures and records:

Example for Structures

This line applies to structure names (begin in column 1)

```
                Name                    Width   # fields
                                        Shift   Width   Mask    Initial
PARMLIST . . . . . .                    001C    0004
  BUFSIZE . . . . . .                   0000
  NAMESIZE . . . . . .                  0001
  NAMETEXT . . . . . .                  0002
  TERMINATOR . . . . .                  001B
```

This line for fields of records (indented).

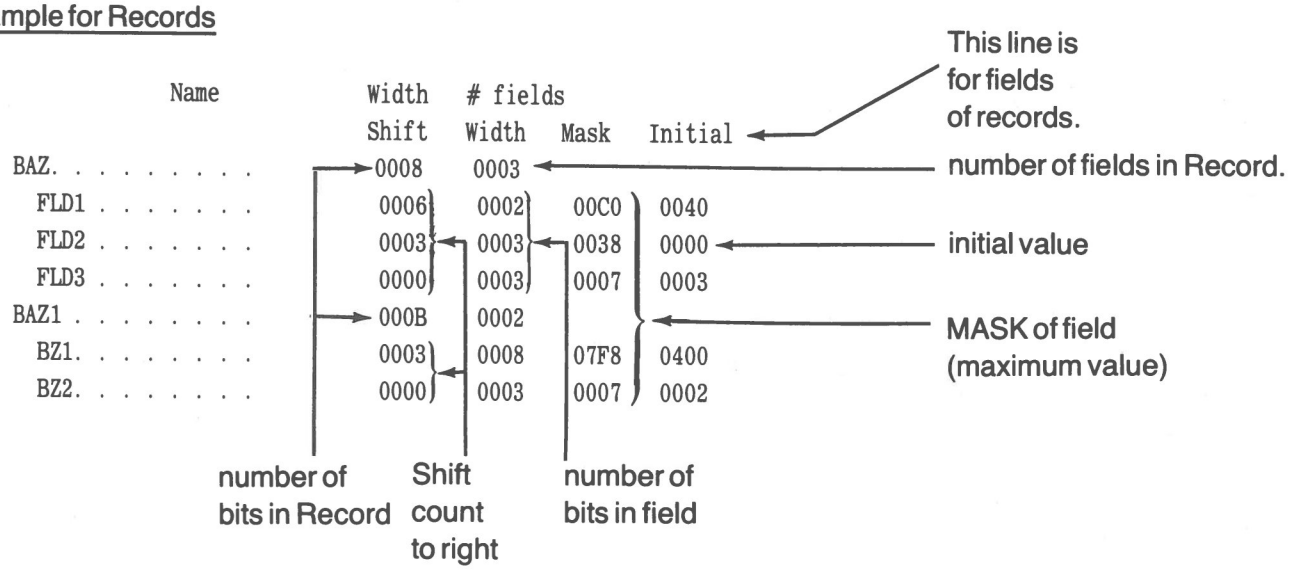
Number of fields in structure

field names of PARMLIST structure

Offset of field into structure

The number of bytes wide of the structure

Example for Records



This section lists your structures and/or records and their fields. The upper line of column headings applies to structure names, record names, and to field names of structures. The lower line of column headings applies to field names of records.

For structures:

Width (upper line) shows the number of bytes your structure occupies in memory.

 # fields shows how many fields comprise your structure.

For records:

Width (upper line) shows the number of bits the record occupies.

 # fields shows how many fields comprise your record.

For fields of structures:

Shift shows the number of bytes the fields is offset into the structure. The other columns are not used for fields of structures.

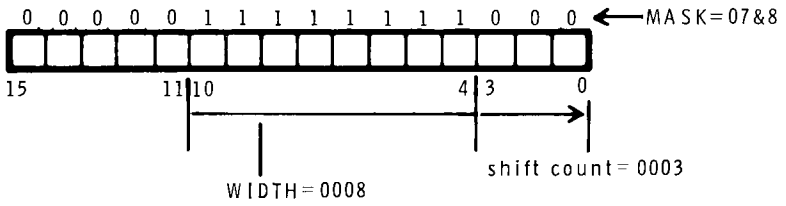
For fields of records:

Shift is the shift count to the right.

10.185

Mask shows the maximum value of record, expressed in hexadecimal, if one field is masked and ANDed (field is set to all 1's and all other fields are set to all 0's).

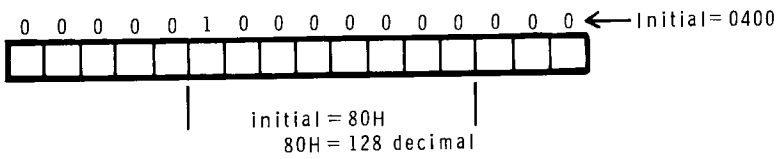
Using field BZ1 of the record BAZ1 above to illustrate:



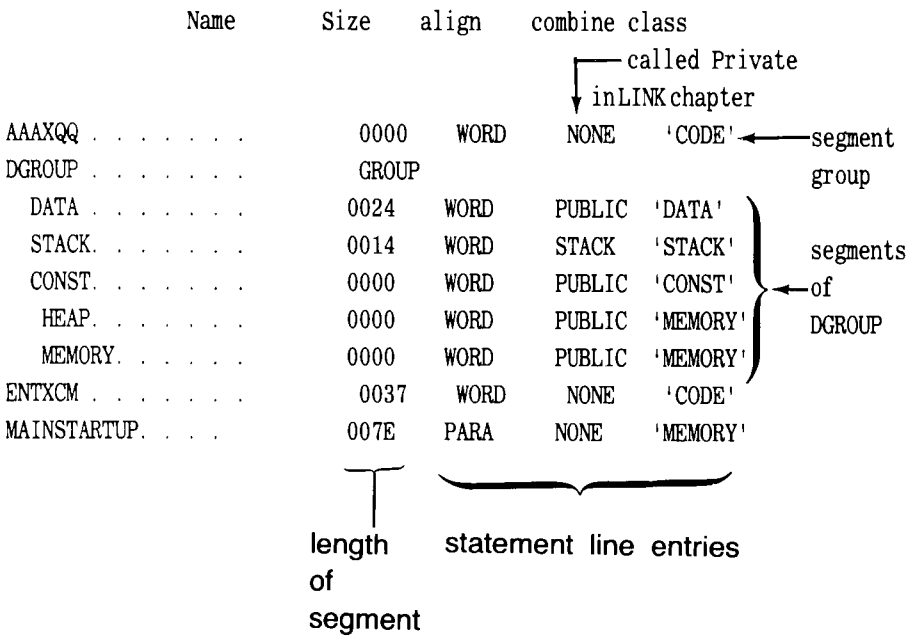
Initial shows the value specified as the initial value for the field, if any.

When naming the field, you specified:
fieldname:# = value

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



Segments and groups:



For groups:

the name of the group appears under the Name column, beginning in column 1 with the applicable segment names indented 2 spaces. The word Group appears under the Size column.

For segments:

the segment names may appear in column 1 (as here) if you do not declare them part of a group. If you declare a group, the segment names appear indented under their group name.

For all segments, whether a part of a group or not:

Size is the number of bytes the segment occupies.

Align is the type of boundary where the segment begins:

PAGE = page — address is xxx000H (low byte = 0); begins on a 256 byte boundary
PARA = paragraph — address is xxxx0H (low nibble = 0); default
WORD = word — address is xxxxeH (low bit of low byte = 0) bit map - |x|x|x|x|x|x|x|0|
BYTE = byte — address is xxxxxH (anywhere)

Combine describes how LINK utility combines the various segments. (See LINK Utility Chapter for a full description.)

Class is the class name under which LINK combines segments in memory. (See LINK Utility Chapter for a full description.)

10.187

Symbols:

```
                Name      Type      Value     Attr
F00. . . . . .            Number    0005
F001 . . . . . .          Text      1.234
F002 . . . . . .          Number    0008                 { all formed by
F003 . . . . . .          Alias     F00                  { EQU or =
F004 . . . . . .          Text      5[BP][DI]            { directive
F005 . . . . . .          Opcode
```

Symbols:

```
                Name      Type     Value     Attr
BEGHQQ . . . . . .        L WORD   0012      DATA    Global
BEGOQQ . . . . . .        L FAR    0000        External
BEGXQQ . . . . . .        F PROC   0000      MAIN_STARTUP  Global Length =006E
CESXQQ . . . . . .        L WORD   0022      DATA    Global
CLNEQQ . . . . . .        L WORD   0002      DATA    Global
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      EXTXCM  Length =001E
STKBQQ . . . . . .        L WORD   0018      DATA    Global
STKHQQ . . . . . .        L WORD   001A      DATA    Global
```

Global Length =006E — 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.

This section lists all other symbolic values in your program that do not fit under the other categories.

Type shows the symbol's type:

L = Label
F = Far
N = Near
PROC = Procedure
Number
Alias
Text
Opcode
—all defined by EQU or = directive

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

For all procedures, the length of the procedure is given after its attribute (segment).

You may also see an entry under "Type" like:

```
L 0031
```

This entry results from code such as the following:

```
BAZ LABEL FOO
```

where FOO is a STRUC that is 31 bytes long.

BAZ is shown in the symbol table with the L 0031 entry. Basically, Number (and 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 shows some text — when the symbol was defined by the EQU or = directives.)

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 under type, the symbol was created by an EQU directive or an = directive. All information that follows one of these entries is considered its "value," even if the "value" is simple text.

Each of the four types shows a value as follows:

Number shows a constant numeric value

Opcode shows a blank. The symbol is an alias for an instruction mnemonic.

Sample directive statement: `FOO EQU ADD`

Alias shows a symbol name which the named symbol equals.

Sample directive statement: `FOO EQU BAX`

Text shows the "text" the symbol represents. "Text" is any other operand to an EQU directive that does not fit one of the other three categories above.

Sample directive statements:

```
GOO EQU 'WOW'
BAZ EQU DS:8[BX]
ZOO EQU 1.234
```

SUMMARY

The ability to rapidly and effectively debug programs is essential to the productive use of assembly language. The symbol and cross reference tables, as well as the object listing itself, all exist to speed the debugging task. You can greatly increase your programming effectiveness by learning to understand and rely on the information they contain.

Introduction to LINK

FEATURES AND BENEFITS OF LINK

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

For all the necessary and optional commands, LINK gives prompts. Your answers to the prompts are the commands for LINK.

The output file from LINK (run file) is not bound to specific memory addresses and, therefore, can be loaded and executed at any convenient address by your specification.

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

LINK is capable of linking files totaling 384K bytes.

LINK

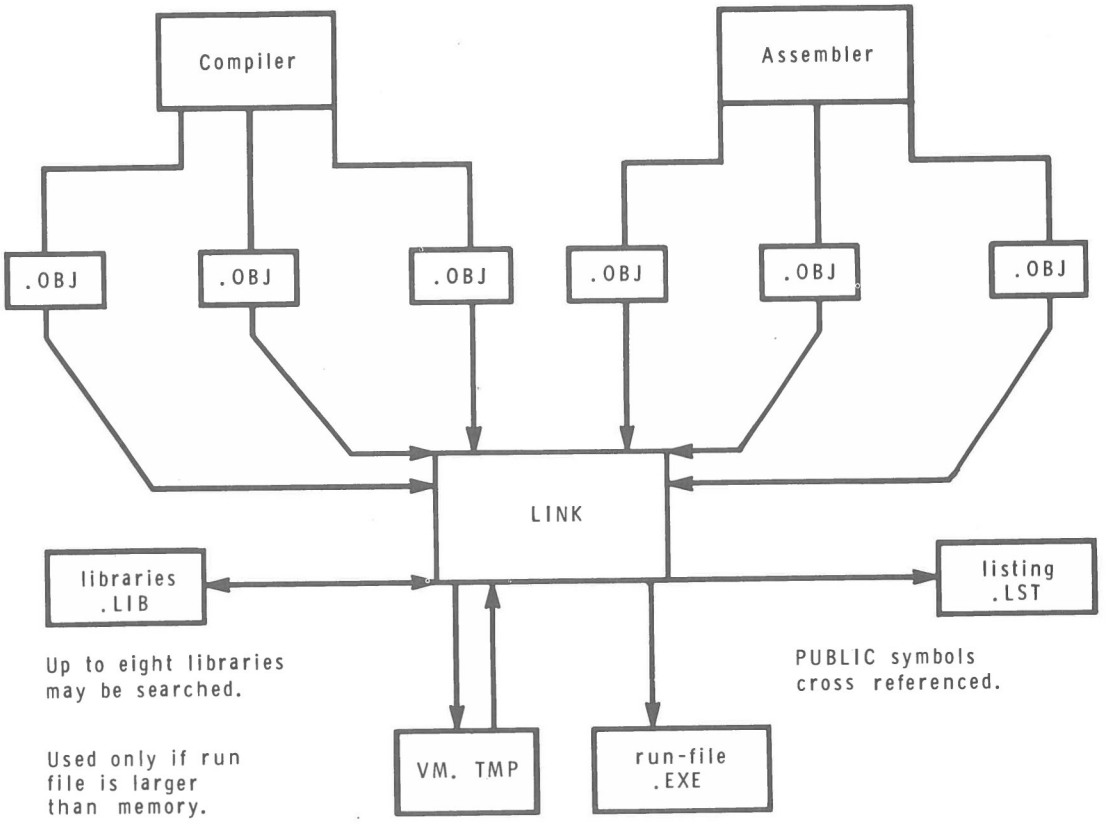
Overview of LINK Operation

LINK combines several object modules into one relocatable load module called a run file.

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

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

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



LINK Operations

Definitions

Three terms appear frequently in the LINK error messages. These terms describe the underlying functioning of LINK. An understanding of the concepts that define these terms provides a basic understanding of the way LINK works.

SEGMENT

A Segment is a continuous area of memory up to 64K bytes in length. A Segment may be located anywhere in 8086 memory on a "paragraph" (16 byte) boundary. The contents of a Segment are addressed by a Segment-register/offset pair.

GROUP

A Group is a collection of Segments which fit within 64K bytes of memory. The Segments are named to the Group by the assembler, by the compiler, or by you. The Group name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler.

The Group is used for addressing Segments in memory. Each Group is addressed by a single Segment register. The Segments within the Group are addressed by the Segment register plus an offset. LINK checks to see that the object modules of a Group meet the 64K byte constraint.

CLASS

A Class is a collection of Segments. The naming of Segments to a Class controls the order and relative placement of Segments in memory. The Class name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler.

Overview of LINK Operation

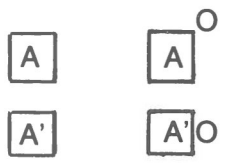
The Segments are named to a Class at compile time or assembly time. The Segments of a Class are loaded into memory continuously. The Segments are ordered within a Class in the order LINK encounters the Segments in the object files. One Class precedes another in memory only if a Segment for the first Class precedes all Segments for the second Class in the input to LINK, Classes may be loaded across 64K byte boundaries. The Classes are divided into Groups for addressing.

How LINK Combines and Arranges Segments

LINK arranges the object module according to the combine types (private, public, stack, ad common) declared in the segment directives. (The memory combine type available in Microsoft's MACRO-86 is treated the same as public. LINK does not automatically place memory combine type as the highest segments.)

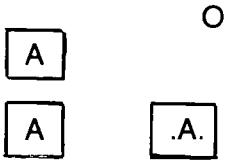
LINK combines segments for these combine types as follows:

PRIVATE



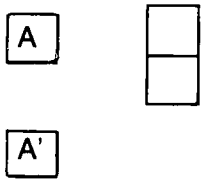
Private segments are loaded separately and remain separate. They may be physically, but not logically, continuous, even if the segments have the same name. Each private segment has its own base address.

PUBLIC



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

COMMON

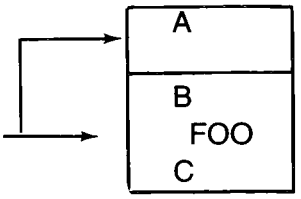


Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Place segments in a Group in the assembler provides offset addressing of items from a single base address for all segments in that Group.

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

Any number of other segments may intervene between segments of a group. Thus, the offset of FOO may be greater than the size of segments in group combined, but no larger than 64K.



An operand of DGROUP:FOO returns the offset of FOO from the beginning of the first segment of DGROUP (segment A here)

Segments are grouped by declared class names. LINK loads all the segments belonging to the first class name it encounters, then loads all the segments of the next class name it encounters, and so on until all classes have been loaded.

If your program contains:	**They load 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 exercise control over the ordering of classes in memory by writing a dummy module and listing it first after the LINK Object Modules prompt. The dummy module declares segments into classes in the order you want the classes loaded.

NOTE: Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs. Allow the compiler and the linker to perform their own class ordering without dummy modules.

For example:

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

You should be careful to declare all classes to be used in your program in this module. If you do not, you lose absolute control over the ordering of classes.

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

Files That LINK Uses

LINK works with one or more input files, produces two output files, may create a virtual memory file, and may be directed to search one to eight library files. For each type of file, you may give a three-part file specification. The format for LINK file specifications is:

d:filename.ext

where d: is the drive designation. Permissible drive designations for LINK are A through D. The colon is always required as part of the drive designation; where filename is any legal filename of one to eight characters; and where .ext is a one- to three-character extension to the filename. The period is always required as part of the extension.

INPUT FILES

If no extensions are given in the input (object) file specifications, LINK recognizes by default:

File	Default Extension
Object	.OBJ
Library	.LIB

OUTPUT FILES

LINK appends to the output (run and list) files the following default extensions:

File	Default Extension
Run	.EXE (may not be overridden)
List	.MAP (may be overridden)

VM.TMP FILE

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary file and names it VM.TMP. If LINK needs to create VM.TMP, it displays the message:

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

Once this message is displayed, you must not remove the disk from the default drive until the link session ends. If the disk is removed, the operation of LINK is unpredictable, and LINK might return the error message:

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

LINK uses VM.TMP as a virtual memory. The contents of VM.TMP are subsequently written to the file named following the run file: prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

NOTE: Do not use VM.TMP as a file name for any file. If you have a file named VM.TMP on the default drive and LINK requires the VM.TMP file, LINK deletes the old VM.TMP. Thus, the contents of the previous VM.TMP file are lost.

Running LINK

LINK requires two types of commands: a command to invoke LINK and answers to command prompts. In addition, six switches control alternate LINK features. Usually you enter all the commands to LINK on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in a response file. Some Command Characters are provided to assist you while you are entering linker commands.

Invoking LINK

There are three ways you may invoke LINK. With the first method, you enter the commands as answers to individual prompts. With the second method, you enter all commands on the line used to invoke LINK. With the third method, you create a response file that contains all the necessary commands.

Summary of Methods to invoke LINK

Method 1 **LINK**

Method 2 **LINK <filenames>[</x>]**

Method 3 **LINK @<filespec>**

METHOD 1: LINK

Enter:

```
A: LINK
```

LINK loads into memory. Then it returns a series of four text prompts that appear one at a time. Answer the prompts as commands to LINK to perform specific tasks.

At the end of each line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, LINK defaults to not performing the function described for the switches in the chart below.

The command prompts are summarized here and described in more detail under "Command Prompts", Page 11.16. Following the summary of prompts is a summary of switches, which are described in more detail under "Switches", Page 11.18.

Summary of Prompts

PROMPT	RESPONSES
`Object Modules [.OBJ]:`	The .OBJ files to be linked, separated by blank spaces or plus signs (+). If a plus sign is the last character you enter, a prompt will reappear. (No default: response required.)
`Run File [Object-file.EXE]:`	The filename for the executable object code. (Default: first-object-filename.EXE.)
`List File [NUL.MAP]:`	The filename for the listing. (Default: NUL filename.)
`Libraries [.LIB]:`	The filenames to be searched, separated by blank spaces or plus signs (+). If a plus sign is the last character you enter, a prompt reappears. (Default: no search.)

LINK

Running LINK

Summary of Switches

SWITCH	ACTION
/DSALLOCATE	Load data at the high end of the Data Segment. This is required for Pascal and FORTRAN programs.
/HIGH	Place the run file as high as possible in memory. Do not use this with Pascal or FORTRAN Programs.
/LINENUMBERS	Include line numbers in the list file.
/MAP	List all global symbols with definitions.
/PAUSE	Halt the linker session and wait for the RETURN key.
/STACK:<number>	Set a fixed stack size in the run file.

Command Characters

LINK provides three command characters.

+ Use the plus sign (+) to separate entries and to extend the current physical line following the Object Modules and Libraries prompts. (A blank space may be used to separate object modules.) To enter a large number of responses (each which may also be very long), enter a plus sign/ RETURN at the end of the physical line (to extend the logical line). If the plus sign/ RETURN is the last entry following these two prompts, LINK prompts you for more module names. When the Object Modules or Libraries prompt appears again, continue to enter responses. When all the modules to be linked have been listed, be sure the response line ends with a module name and a RETURN and not a plus sign/ RETURN.

Example:

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

; Use a single semicolon (;) followed immediately by a RETURN at any time after the first prompt (from run file on) to select default responses to the remaining prompts. This feature saves time and overrides the need to enter a series of RETURNS.

NOTE: Once the semicolon has been entered, you can no longer respond to any of the prompts for that link session. Therefore, do not use the semicolon to skip over some prompts. For this, use a RETURN.

Example:

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

The remaining prompts do not appear, and LINK uses the default values (including NUL.MAP for the list file).

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

LINK

Running LINK

METHOD 2: LINK <filenames>[</x>]

Enter:

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

The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

Where: <object-list> is a list of object modules, separated by plus signs; <runfile> is the name of the file to receive the executable output; <listfile> is the name of the file to receive the listing; <lib-list> is a list of library modules to be searched; and </x> are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <lib-list>, as shown).

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

Example:

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

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

METHOD 3: LINK @<filespec>

Enter:

LINK @<filespec>

Where: <filespec> is the name of a response file. A response file contains answers to the LINK prompts (shown under Method 1 for invoking), and may also contain any of the switches. Method 3 permits you to conduct the LINK session without interactive (direct) user responses to the LINK prompts.

NOTE: Before using Method 3 to invoke LINK, you must first create the response file.

A response file has text lines, one for each prompt. Responses must appear in the same order as the command prompts appear.

Switches and Command Characters in the response file are used the same way as they are used for responses entered on the terminal keyboard.

When the LINK session begins, each prompt displays, in turn with the responses from the response file. If the response file does not contain answers for all the prompts, either in the form of filenames or the semicolon Command Character or RETURNS, LINK will, after displaying the prompt which does not have a response, wait for you to enter a legal response. When a legal response has been entered, LINK continues the link session.

Example:

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

This response file causes LINK to load the four Object modules. LINK pauses before creating and producing a public symbol map to permit you to swap disks (see this discussion under /PAUSE in the section on "Switches", Page 11.19 before using this feature). When you press RETURN the output files are named FUN.EXE and FUNLST, and MAP. LINK searches the library file COBLIB. LIB. LINK uses the default settings for the flags.

Command Prompts

You can command LINK by entering responses to four text prompts. When you have entered a response to the current prompt, the next appears. When the last prompt has been answered, LINK begins linking automatically without further command. When the link session is finished, LINK exits to the operating system. When the operating system prompt is displayed, LINK has finished successfully. If the link session is unsuccessful, LINK returns the appropriate error message.

LINK prompts you for the names of object, run, list files and libraries. The prompts are listed in their order of appearance. For prompts which can default to preset responses, the default response is shown in square brackets ([ ]) following the prompt. The Object Modules: prompt is followed by only a filename extension default response because it has no preset filename response and requires a filename from you.

`Object Modules [.OBJ]:` Enter a list of the object modules to be linked. LINK assumes by default that the filename extension is .OBJ. If an object module has any other filename extension, the extension must be given here. Otherwise, the extension may be omitted.

Modules must be separated by plus signs (+).

Remember that LINK loads Segments into Classes in the order encountered (see "Definitions" on Page 11.3). Use this information for setting the order in which the object modules are entered.

Run File [First-Object-filename.EXE]: The filename entered is created to store the run (executable) file that results from the link session. All run files receive the filename extension .EXE, even if you specify an extension (your specified extension is ignored).

If no response is entered to the run file prompt, LINK uses the first filename entered in response to the Object Modules prompt as the RUN filename.

Example:

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

This response directs LINK to create the run file PAYROLL.EXE on drive B:. Also, LINK pauses, which allows you to insert a new disk to receive the run file.

List File [NUL.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 no list filename with the default filename extension .MAP.

Libraries [.LIB]: The valid responses are one to eight library filenames or simply a RETURN. (A RETURN only means no library search.) Library files must have been created by a library utility. LINK assumes by default that the filename extension is .LIB for library files.

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

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

If LINK cannot find a library file on the disks in the disk drives, it returns the message:

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

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

LINK does not search within each library file sequentially. LINK uses a method called dictionary indexed library search. This means that LINK finds definitions for external references by index access rather than by searching from the beginning of the file to the end, for each reference. This indexed search reduces the link time for any sessions involving substantial library searches.

Switches

The six switches control alternate linker functions. Switches must be entered at the end of a prompt response regardless of which method is used to invoke LINK. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is entered at the end of one response, each switch must be preceded by the slash mark (/).

All switches may be abbreviated, from a single letter through the whole switch name. The only restriction is that an abbreviation must be a sequential sub-string from the first letter through the last entered; no gaps or transpositions are allowed. For example:

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

/DSALLOCATE Use of the /DSALLOCATE switch directs LINK to load all data (DGroup) at the high end of the Data Segment. Otherwise, LINK loads all data at the low end of the Data Segment. At runtime, the DS pointer is set to the lowest possible address and allows the entire DS segment to be used. Use of the /DSALLOCATE switch in combination with the default load low (that is, the /HIGH switch is **not** used), permits your application to allocate dynamically any available memory below the area specifically allocated within DGroup, yet to remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

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

/HIGH Use of the /HIGH switch causes LINK to place the run image as high as possible in memory. Otherwise, LINK places the run file as low as possible.

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

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

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

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

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

/PAUSE The /PAUSE switch causes LINK to pause in the link session when the switch is encountered. Normally, LINK performs the linking session without stop from beginning to end. This allows you to swap the disks before the LINK outputs the run (.EXE) file.

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

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

LINK resumes processing when you press **ENTER** or **RETURN**.

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

/STACK:<number> Number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If the /STACK switch is not used for a link session, LINK calculates the necessary stack size automatically.

If you enter a value from 1 to 511, LINK uses 512.

All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, LINK returns a `WARNING: NO STACK STATEMENT` error message.

Introduction to LIB

FEATURES AND BENEFITS OF LIB

Brief

LIB creates an indexed master file of up to 500 .OBJ files produced by the MACRO-86 assembler. Directing LINK to search a library greatly speeds program development. Once a library is built, you may create a complex program by writing and assembling a single control module based upon calls to external routines. Specify the control module as the .OBJ file to be linked and the library name as the .LIB file to be searched. All required modules will be extracted and linked to your program.

Details

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

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

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

The command scanner in LIB is the same as the one used in Microsoft's LINK, MS-PASCAL, MS-FORTRAN, and other 16-bit Microsoft products. If you have used any of these products, using LIB is familiar to you. Command syntax is straightforward, and LIB prompts you for any of the commands it needs that you have not supplied.

OVERVIEW OF LIB OPERATION

Brief

LIB provides five basic functions:

1. Create a library (.LIB) file.
2. Add a module.
3. Delete a module.
4. Replace a module with a revised version.
5. Copy a module to a separate .OBJ file.

You may also select an optional cross reference of all PUBLIC symbols.

Details

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

1. Delete a module.

2. Extract a module and place it in a separate object file.

3. Append an object file as a module of a library.

4. Replace a module in the library file with a new module.

5. Create a library file.

Introduction to LIB

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

For example:

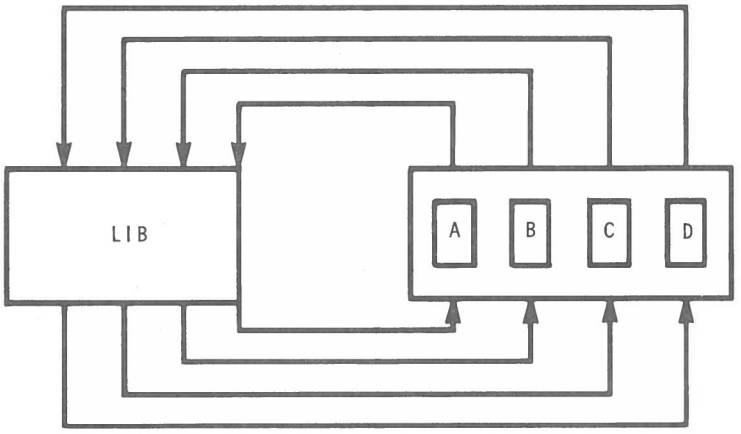
```
LIB PASCAL -HEAP-HEAP;
```

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

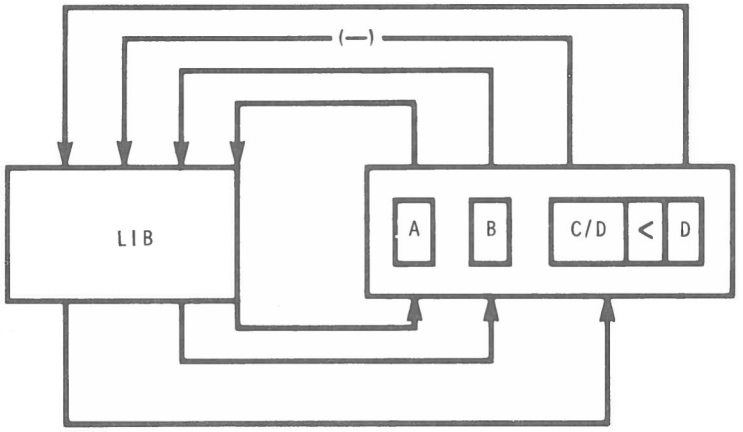
LIB

Introduction to LIB

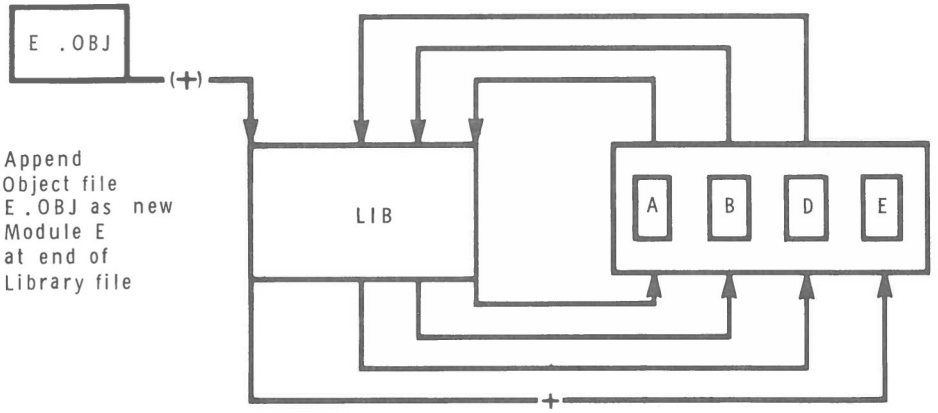
Consistency check only

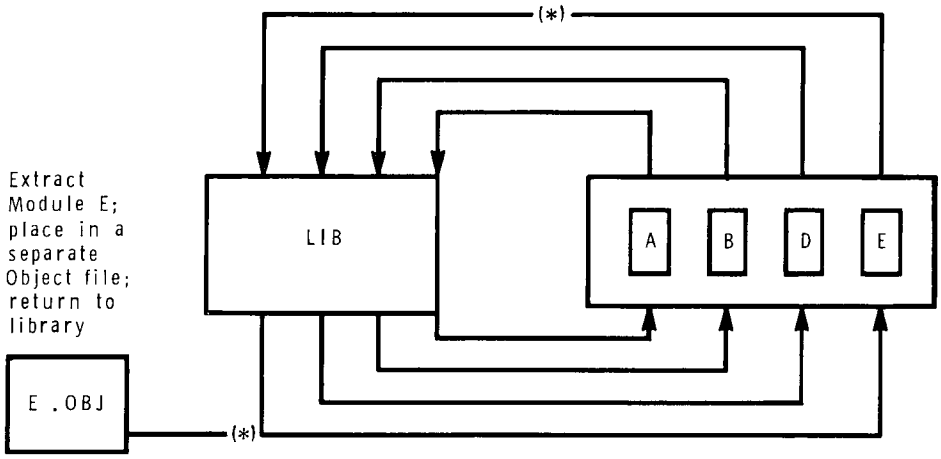


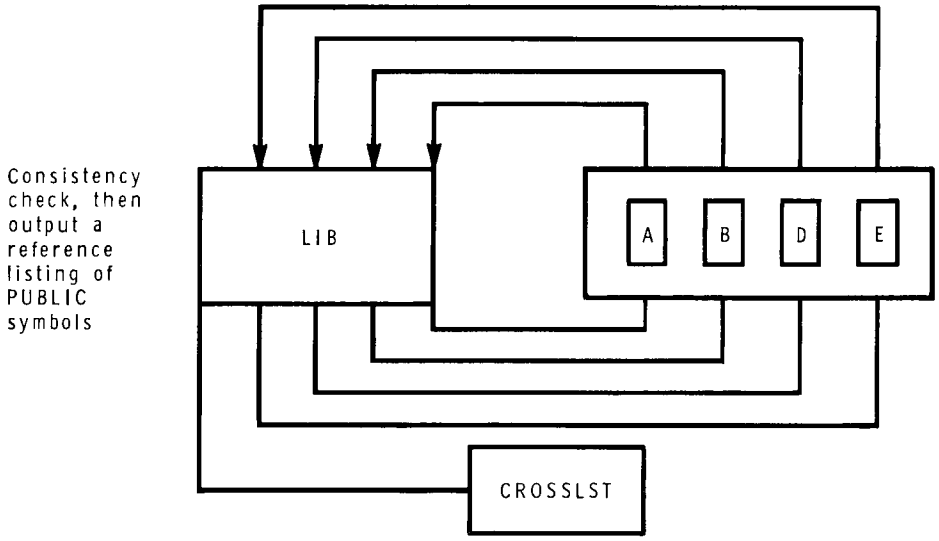
Delete Module C; Module D written to space of Module C



Append Object file E.OBJ as new Module E at end of Library file




(*)
Extract Module E; place in a separate Object file; return to library
LIB
A
B
D
E
E .OBJ
(*)


Consistency check, then output a reference listing of PUBLIC symbols
LIB
A
B
D
E
CROSSLST

LIB

Running LIB

Brief

Invoke LIB by entering LIB after the operating system prompt. LIB responds with a series of three queries asking the name of the library file to be manipulated, the operation(s) to be performed, and the name for an optional listing file. Syntactic details of the responses are covered in the next brief.

You must enter a library filename. LIB expects it to have an extension of .LIB. Any other extension must be entered. If the file does not exist, you are given the option to create it. The default for the operations query is to perform no operations. The default for the listing query is not to produce a listing file. Select the default for a query by entering a carriage return. You may enter a semicolon any time after the filename to select the default for all of the remaining queries.

If the desired operations do not fit on one line, end the line with an ampersand. An additional "operations" query will be provided.

You may override the query process by entering all the responses after the LIB entry. There is no delimiter between the library filename and the operations response. A comma separates the operations response from the listing filename. Alternately, a batch response file may be created to eliminate user interaction. Control-C aborts LIB at any time.

Details

Running LIB requires two types of commands: a command to invoke LIB and answer to command prompts. Usually you enter all the commands to LIB on the terminal keyboard. As an option, answers to the command prompts may be contained in a response file. Some Command Characters exist. Some are used as a required part of LIB commands. Others assist you while entering LIB commands.

INVOKING LIB

LIB may be invoked three ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke LIB. By the third method, you must first create a response file that contains all the necessary commands.

Method 1	**LIB**
Method 2	**LIB** <**library**><**operations**>,<**list**>
Method 3	**LIB** @<**filespec**>

Summary of Methods to Invoke LIB

Method 1: LIB

Enter:

LIB

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

The Command Prompts and Command Characters are summarized here. The Command Prompts and Command Characters are described fully on Pages 12.15 and 12.17 respectively.

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

Summary of Command Prompts

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

Summary of Command Characters

Method 2: LIB <library><operations>,<list>

Brief

There are three possible responses to the operations query:

Enter a plus (+) followed by the name of an accessible object file to enter it into the library. A default extension of .OBJ is expected. Any other extension must be specified. You may include a drive specifier. LIB strips both the driver specifier and the extension, leaving a module name equal to the basic filename.

Enter a minus (–) followed by the name of a library module to delete it.

Enter an asterisk (*) followed by the name of a library module to copy it to a new .OBJ file. You may not override the .OBJ extension or specify an alternate drive.

Operations responses may be chained without delimiters. A replacement operation is equal to a deletion plus an entry. Deletions are always performed before entries, regardless of their sequence in the command line. This simplifies replacement operations by allowing use of the same module name. LIB makes all additions to the end of the library and compacts the file to fill space left by deletions. Lastly, the module index is generated. The operating system prompt will appear when all operations are complete.

Details

Enter:

LIB <library><operations>,<list>

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

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

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

Enter **Y**es (or any response beginning with "**Y**") to create a new library file. Enter **N**o (or any other response not beginning with Y) to abort the library session.

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

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

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

If you enter a library filename followed immediately by a semicolon, 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 filename followed immediately by a comma and a List filename, LIB performs its consistency check of the library file, then produces the cross-reference listing file.

Example:

LIB PASCAL-HEAP+HEAP;

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

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

Example:

LIB PASCAL; RETURN

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

Example:

LIB PASCAL,PASCROSS.PUB RETURN

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

Method 3: LIB @<filespec>

Brief

Enter:

LIB @<filespec>

Filespec is the name of a response file. A response file contains answers to the LIB prompts (summarized under Method 1 for invoking, and described fully on Page 12.8. Method 3 permits you to conduct the LIB session without interactive (direct) user responses to the LIB prompts.

NOTE: Before using Method 3 to invoke LIB, you must first create the response file.

A response file has text lines, one for each prompt. Responses must appear in the same order as the command prompts appear.

Use Command Characters in the response file the same way as they are used for responses entered on the terminal keyboard.

When the library session begins, each prompt displays in turn with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses (no changes to the modules currently in the library file for Operation, and no cross-reference listing file created).

If you enter a library filename followed immediately by a semicolon, 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 filename followed by a carriage return, then a comma and a list filename, LIB performs its consistency check of the library file, then produces the cross-reference listing file.

Example:

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

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

Details

Command Prompts

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

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

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

If you enter a library filename and follow it immediately with a semicolon command character, LIB performs a consistency check only, then returns to the operating system. Any errors in the file are reported.

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

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

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

LIB

Running LIB

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

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

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

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

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

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

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

Command Characters

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

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

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

  ```
  B:CURSOR.OBJ
  ```

  a response to the Operation prompt of:

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

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

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

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

*

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

```
CURSOR
```

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

```
*CURSOR
```

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

```
A:CURSOR.OBJ
```

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

;

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

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

Example:

```
Library File:  FUN  RETURN
Operations:  +CURSOR;  RETURN
```

The remaining prompt does not appear, and LIB uses the default value (no cross-reference file).

&

Use the ampersand to extend the current physical line. This command character is needed only for the Operation prompt. LIB can perform many functions during a single library session. The number of modules you can append is limited only to 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. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. LIB displays the Operation prompt again; then you should enter more responses. You may use the ampersand character as many times as you need. For example:

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

LIB deletes the module HEAP, extracts the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then appends the object files CURSOR, HEAP, ASSUME, and RIDE. Note, however, that LIB allows you to enter your Operation responses in any order.

CTRL-C

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

Introduction to CREF

FEATURES AND BENEFITS

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

The CREF produced listing is meant to be used with the symbol table produced by your assembler.

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

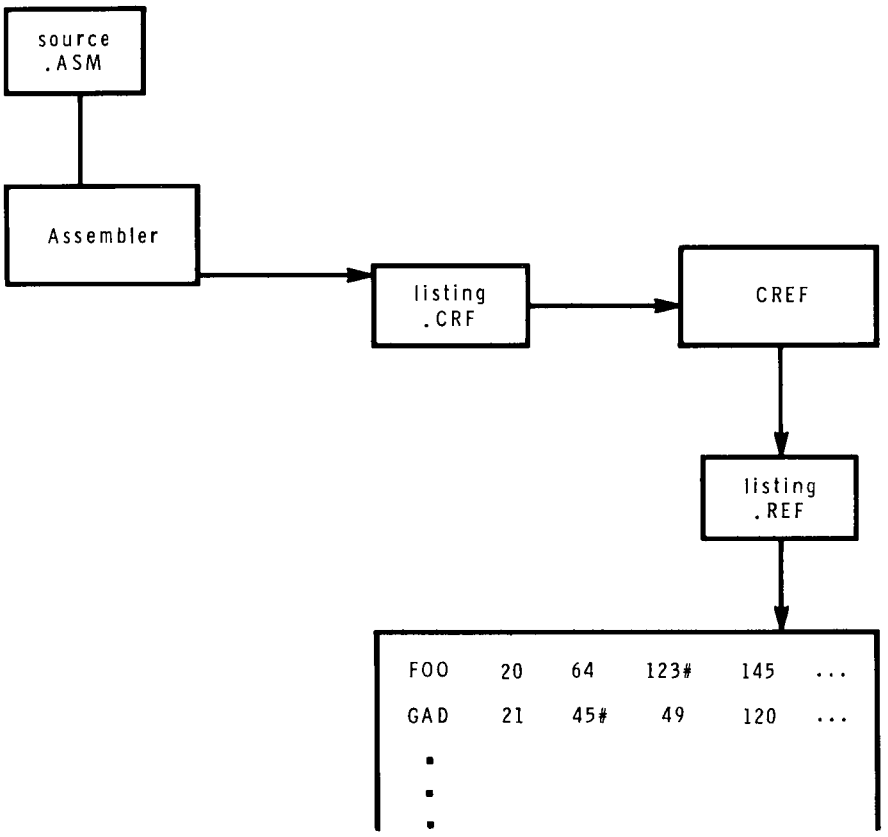
The cross-reference listing produced by CREF provides you with the locations, speeding your search and allowing for faster debugging.

OVERVIEW OF CREF OPERATION

CREF produces a file with cross-references for symbolic names in your program.

First, you must create a cross-reference file with the assembler. Then, CREF takes this cross-reference file, which has the filename extension .CRF, and turns it into an alphabetical listing of the symbols in the file. The cross-reference listing file is given the default filename extension .REF.

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


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

Running CREF

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

Before you can use CREF to create the cross-reference listing, you must first have created a cross-reference file using your assembler. This step is reviewed on this page.

CREATING A CROSS-REFERENCE FILE

A cross-reference file is created during an assembly session.

To create a cross-reference file, answer the fourth assembler command prompt with the name of the file you want to receive the cross-reference file.

The fourth assembler prompt is:

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

If you do not enter a filename in response to this prompt, or if you in any other way use the default response to this prompt, the assembler does not create a cross-reference file. Therefore, you must enter a filename. You may also specify which drive or device you want to receive the file and what filename extension you want the file to have, if different from .CRF. If you change the filename extension from .CRF to anything else, you must remember to specify the filename extension when naming the file in response to the first CREF prompt (see Page 13.4).

When you have given a filename in response to the fourth assembler prompt, the cross-reference file is generated during the assembly session.

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

INVOKING CREF

CREF may be invoked two ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke CREF.

Method 1	**CREF**
Method 2	**CREF <crffile>,<listing>**

Summary of Methods to Invoke CREF

Method 1: CREF

Enter:

CREF

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

Command Prompts

`CREF filename [.CRF]:` Enter the name of the cross-reference file you want CREF to convert into a cross-reference listing. The name of the file is the name you gave your assembler when you directed it to produce the cross-reference file.

CREF assumes that the filename extension is .CRF. If you do not specify a filename extension when you enter the cross-reference filename, CREF looks for a file with the name you specify and the filename extension .CRF. If your cross-reference file has a different extension, specify the extension when entering the filename.

See "Format of CREF Compatible Files", on Page 13.11 for a description of what CREF expects to see in the cross-reference file. You need this information only if your cross-reference file was not produced by a Microsoft assembler.

`List filename [crffile.REF]:` Enter the name you want the cross-reference listing file to have. CREF automatically gives the cross-reference listing the filename extension .REF.

If you want your cross-reference listing to have the same filename as the cross-reference file but with the filename extension .REF, simply press the RETURN key when the List filename prompt appears. If you want your cross-reference listing file to be named anything else and/or to have any other filename extension, you must enter a response following the List filename prompt.

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

Special Command Characters

;

Use a single semicolon (;) followed immediately by the RETURN key at any time after responding to the cross-reference prompt, to select the default response to the Listing prompt. This feature saves time and overrides the need to answer the Listing prompt.

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

Example:

```
Cref filename [.CRF]:    FUN;
```

CREF processes the cross-reference file named FUN.CRF and outputs a listing file named FUN.REF.

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

Method 2: CREF <crffile>,<listing>

Enter:

CREF <crffile>,<listing>

CREF loads into memory, then immediately proceeds to convert your cross-reference file into a cross-reference listing.

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

Crffile is the name of a cross-reference file produced by your assembler, CREF assumes that the filename extension is .CRF, which you may override by specifying a different extension. If the file named for the crffile does not exist, CREF displays the message:

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

Control then returns to your operating system.

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

To select the default filename and extension for the listing file, enter a semicolon after you enter the crffile name.

Example:

```
CREF FUN;  RETURN
```

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

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

```
CREF FUN,B:WORK.ARG
```

This example causes CREF to process the cross-reference file named FUN.CRF and to produce a listing file named WORK.ARG, which is placed on the disk in drive B:.

Format of Cross-Reference Listings

The cross-reference listing is an alphabetical list of all the symbols in your program.

Each page is headed with the title of the program or program module.

Then comes the list of symbols. Following each symbol name is a list of the line numbers where the symbol occurs in your program. The line number for the definition has a pound sign (#) appended to it.

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

```
ENTX    PASCAL entry for initializing programs    ←comes from TITLE directive

  Symbol Cross Reference    (# is definition)         Cref-1

AAAXQQ . . . . . .    37#   38

BEGHQQ . . . . . .    83    84#  154  176
BEGOQQ . . . . . .    33   162
BEGXQQ . . . . . .    11   126#  223

CESXQQ . . . . . .     9    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

INIQQ. . . . . .      31    161

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

PNUXQQ . . . . . .    69    70   150

ENTX    PASCAL entry for initializing programs       Cref -2

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

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

Format of CREF Compatible Files

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

GENERAL DESCRIPTION OF CREF FILE PROCESSING

In essence, 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.

CREF keeps track of line numbers in the source file by the number of end-of-line characters it encounters. Therefore, every line in the source file must contain at least one end-of-line character (see the chart on Page 13.12).

CREF attempts to place a heading at the top of every page of the listing. The name it uses as a title is the text passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol (see the chart on Page 13.12). If CREF encounters more than one title symbol in the source file, it uses the last title read for all page headings. If CREF does not encounter a title symbol in the file, the title line on the listing is left blank.

FORMAT OF SOURCE FILES

CREF uses the first three bytes of the source file as format specification data. The rest of the file is processed as a series of records that either begin or end with a byte that identifies the type of record.

First Three Bytes

(The PAGE directive in your assembler, which takes arguments for page length and line length, passes this information to the cross-reference file.)

First Byte — The number of lines to be printed per page (page length ranges from 1 to 255 lines).

Second Byte — The number of characters per line (line length ranges from 1 to 132 characters).

Third Byte — The Page Symbol (07) that tells CREF that the two preceding bytes define listing page size.

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

Control Symbols

The two charts on Page 13.12 show the types of records that CREF recognizes and the byte values and placement it uses to recognize record types.

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

Format of CREF Compatible Files

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

Records That Begin with a Control Symbol

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

Records That End with a Control Symbol

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

01 CTRL-A
02 CTRL-B
03 CTRL-C
04 CTRL-D
05 CTRL-E
06 CTRL-F
07 CTRL-G

The Control Symbols are defined as follows:

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

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

End-of-line — Record is an end-of-line symbol character only (04H or Control-D).

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

Title defined — ASCII characters of the title to be printed at the top of each listing page. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used for the title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the listing is left blank.

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

Summary of CREF File Record Contents

Byte contents	Length of record
01 symbol name	2-81 bytes
02 symbol name	2-81 bytes
04	1 byte
05 1A	2 bytes
title text 06	2-81 bytes
PL LL 07	3 bytes

Part 4

Appendices and Index

APPENDIX A

Operating System Error Messages

CHKDSK ERRORS

If an error is detected, CHKDSK returns one of the following error messages:

```
Allocation error for file <filename>
```

The named file had a data block allocated to it that did not exist (that is, a data block number larger than the largest possible block number). CHKDSK truncates the file short of the bad block.

```
Disk not initialized
```

No directory or file allocation table was found. If files exist on the disk, and the disk has been physically harmed, it may still be possible to transfer files from this disk to recover data.

```
Directory error-file: <filename>
```

No valid data blocks are allocated to the named file. CHKDSK deletes the file.

```
Files cross-linked: <filename> and <filename>
```

The same data block is allocated to both files. No corrective action is taken. To correct the problem, first use the COPY command to make copies of both files; then, delete the originals. Review each file for validity and edit as necessary.

```
File size error for file <filename>
```

The size of the file in a directory is different from its actual size. The size in the directory is automatically adjusted to indicate its actual size on the disk. (The amount of useful data may be less than the size shown because the last data block may not be used fully.)

```
XXXXXX bytes of disk space freed
```

Disk space shown as allocated was not actually allocated and has been freed.

APPENDIX A

Operating System Error Messages

COPY ERRORS

```
File cannot be copied onto itself
        0 File(s) copied
```

During a COPY command if the first filespec (source) references a file that is on the default drive and the second filespec (destination) is not given, the COPY will be aborted. (Copying a file to itself is not allowed.)

The Z-DOS prompt will reappear following the error message.

```
Content of destination lost before copy
```

It is easy to enter a concatenation COPY command where one of the source files is the same as the destination, yet this often cannot be detected. For example, the following command is an error if ALL.LST already exists:

```
A: COPY  *.LST  ALL.LST
```

This is not detected, however, until it is ALL.LST's turn to be appended. At this point it could already have been destroyed.

COPY handles this problem like this: as each input file is found, its name is compared with the destination. If they are the same, that one input file is skipped, and the message "Content of destination lost before copy" is printed. Further concatenation proceeds normally.

DATE ERROR

If the parameters or separators are not legal, Z-DOS returns the messages:

```
Invalid date, enter as mm-dd-yy
Enter new date:_
```

and waits for the user to enter a legal date.

APPENDIX A

Operating System Error Messages

DEBUG ERRORS

ERROR CODE	DEFINITION
BF	Bad Flag

The user attempted to alter a flag, but the characters entered were not one of the acceptable pairs of flag values. See the REGISTER command for the list of acceptable flag entries.

BP	Too many Breakpoints

The user specified more than ten breakpoints as parameters to the G command. Reenter the Go with ten or fewer breakpoints.

BR	Bad Register

The user entered the R command with an invalid register name. See the REGISTER command for the list of valid register names.

DF	Double Flag

The user entered two values for one flag. The user may specify a flag value only once per RF command.

If a syntax error occurs in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow and the word error. For example:

```
>dcs:100 cs:110
          ^error          (not a valid hex digit)
```

Any combination of upper and lower case may be used in DEBUG commands. Spaces or commas are legal delimiters for parameters. A delimiter is required only between two consecutive hexadecimal values.

DISK ERRORS

If a disk error occurs at any time during any command or program, Z-DOS retries the operation three times. If the operation cannot be completed successfully, Z-DOS returns an error message in the following format:

```
<type> error <I/O action> drive d
Abort, Retry, Ignore: _
```

In this message, `type` may be one of the following:

```
Write protect
Not ready
SEEK
DATA
SECTOR NOT FOUND
WRITE FAULT
DISK
```

The `I/O-action` may be either of the following:

```
reading
writing
```

The drive `d` indicates the drive in which the error has occurred.

Z-DOS waits entry of one of the following responses:

A Abort. Terminate the program requesting the disk read or write.

I Ignore. Ignore the bad sector and pretend the error did not occur.

R Retry. Repeat the operation. This response is particularly useful if the operator has corrected the error (such as with NOT READY or WRITE PROTECT).

Usually, you will want to attempt recovery by entering responses in the order:

R (to try again)

A (to terminate program and try a new disk)

One other error message might be related to faulty disk read or write:

```
FILE ALLOCATION TABLE BAD FOR DRIVE d
```

This message means that the copy in memory of one of the allocation tables has pointers to nonexistent blocks. Possibly the disk was not formatted before use.

APPENDIX B

MACRO-86 Assembler Error Messages

Most of the messages output by MACRO-86 are error messages. The non-error messages output by MACRO-86 are the banner MACRO-86 displays when first invoked, the command prompt messages, and the end of (successful) assembly message. These nonerror messages are classified here as operating messages, I/O handler messages, and runtime messages.

OPERATING MESSAGES

Banner Message and Command Prompts:

```
The Microsoft MACRO Assembler Version 1.05,
Copyright (c) Microsoft, Inc, 1981, 82

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

End of Assembly Message:

```
Warning   Severe
Errors    Errors
n         n        (n=number of errors)
```

followed by the Z-DOS system prompt (the currently logged drive).

If the assembler encounters errors, error messages are output, along with the number of warning and fatal errors, and control is returned to your disk operating system. The message is output either to your terminal screen or to the listing file if you command one to be created.

Error messages are divided into three categories: assembler errors, I/O handler errors, and runtime errors. In each category, messages are listed in alphabetical order with a short explanation where necessary. At the end of this appendix, the error messages are listed in a single numerical order list but without explanations.

ASSEMBLER ERRORS

`Already defined locally (Code 23)`

Tried to define a symbol as EXTERNAL that had already been defined locally.

`Already had ELSE clause (Code 7)`

Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

`Already have base register (Code 46)`

Trying to double base register.

`Already have index register (Code 47)`

Trying to double index address.

`Block nesting error (Code 0)`

Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is the close of an outer level of nesting with inner level(s) still open.

`Byte register is illegal (Code 58)`

Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

`Can't override ES segment (Code 67)`

Trying to override the ES segment in an instruction where this override is not legal. For example, store string.

`Can't reach with segment reg (Code 68)`

There is no assume that makes the variable reachable.

Can't use EVEN on BYTE segment (Code 70)

Segment was declared to be byte segment and attempt to use EVEN was made.

Circular chain of EQU aliases (Code 83)

An alias EQU eventually points to itself.

Constant was expected (Code 42)

Expected a constant but received something else.

CS register illegal usage (Code 59)

Attempt made to use the CS register illegally. For example, XCHG CS,AX.

Directive illegal in STRUC (Code 78)

All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

Division by 0 or overflow (Code 29)

An expression is given that results in a divide by 0.

Illegal use of register (Code 49)

Use of a register with an instruction where there is no 8086 instruction possible.

Illegal value for DUP count (Code 72)

DUP counts must be a constant that is not 0 or negative.

Improper operand type (Code 52)

Use of an operand such that the opcode cannot be generated.

```
Improper use of segment reg (Code 61)
```

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

```
Index displ. must be constant (Code 54)
```

```
Label can't have seg. override (Code 65)
```

Illegal use of segment override.

```
Left operand must have segment (Code 38)
```

Used something in right operand that required a segment in the left operand. (For example, ":.")

```
More values than defined with (Code 76)
```

Too many fields given in REC or STRUC allocation.

```
Must be associated with code (Code 45)
```

Use of data related item where code item was expected.

```
Must be associated with data (Code 44)
```

Use of code related item where data related item was expected. For example, MOV AX,<code-label>.

```
Must be AX or AL (Code 60)
```

Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

```
Must be index or base register (Code 48)
```

Instruction requires a base or index register and some other register was specified in square brackets, [].

```
Must be declared in pass 1 (Code 13)
```

Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

```
Must be in segment block (Code 69)
```

Attempt to generate code when no + in a segment.

```
Must be record field name (Code 33)
```

Expecting a record field name but got something else.

```
Must be record or field name (Code 34)
```

Expecting a record name or field name and received something else.

```
Must be register (Code 18)
```

Register expected as operand but user furnished symbol — was not a register.

```
Must be segment or group (Code 20)
```

Expecting segment or group and something else was specified.

```
Must be structure field name (Code 37)
```

Expecting a structure field name but received something else.

```
Must be symbol type (Code 22)
```

Must be WORD, DW, QW, BYTE, or TB but received something else.

```
Must be var, label or constant (Code 36)
```

Expecting a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)

Use of one of the prefix instructions without specifying any opcode after it.

Near JMP/CALL to different CS (Code 64)

Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)

Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

No or unreachable CS (Code 62)

Trying to jump to a label that is unreachable.

Normal type operand expected (Code 41)

Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Not proper align/combine type (Code 25)

SEGMENT parameters are incorrect.

One operand must be const (Code 39)

This is an illegal use of the addition operator.

Only initialize list legal (Code 77)

Attempt to use STRUC name without angle brackets, < >.

Operand combination illegal (Code 63)

Specification of a two-operand instruction where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)

Illegal use of subtraction operator.

Operand must have segment (Code 43)

Illegal use of SEG directive.

Operand must have size (Code 35)

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

Operand not in IP segment (Code 51)

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

Operand types must match (Code 31)

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

Operand was expected (Code 27)

Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)

Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)

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

APPENDIX B

MACRO-86 Assembler Error Messages

```
Override with DUP is illegal (Code 79)
```

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

```
Phase error between passes (Code 6)
```

The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass one and pass two of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass two causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes (see "Switches" on page 11.18).

```
Redefinition of symbol (Code 4)
```

This error occured on pass two and succeeding definitions of a symbol.

```
Reference to mult defined (Code 26)
```

The instruction references something that has been multi-defined.

```
Register already defined (Code 2)
```

This occurs only if the assembler has internal logic errors.

```
Register can't be forward ref (Code 82)
```

```
Relative jump out of range (Code 53)
```

Relative jumps must be within the range −128 to +127 of the current instruction, and the specific jump is beyond this range.

```
Segment parameters are changed (Code 24)
```

List of arguments to SEGMENT was not identical to the first time this segment was used.

Shift count is negative (Code 30)

A shift expression is generated that results in a negative shift count.

Should have been group name (Code 12)

Expecting a group name but something other than this was given.

Symbol already different kind (Code 15)

Attempt to define a symbol differently from a previous definition.

Symbol already external (Code 73)

Attempt to define a symbol as local that is already external.

Symbol has no segment (Code 21)

Trying to use a variable with SEG, and the variable has no known segment.

Symbol is multi-defined (Code 5)

This error occurs on a symbol that is later redefined.

Symbol is reserved word (Code 16)

Attempt to use an assembler reserved word illegally. (For example, to declare MOV as a variable.)

Symbol not defined (Code 9)

A symbol is used that has no definition.

Symbol type usage illegal (Code 14)

Illegal use of a PUBLIC symbol.

Syntax error (Code 10)

The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)

The type specified is of an unacceptable size.

Unknown symbol type (Code 3)

Symbol statement has something in the type field that is unrecognizable.

Usage of ? (indeterminate) bad (Code 75)

Improper use of the "?". For example, ?+5.

Value is out of range (Code 50)

Value is too large for expected use. For example, MOV AL,5000.

Wrong type of register (Code 19)

Directive or instruction expected one type of register, but another was specified. For example, INC CS.

I/O HANDLER ERRORS

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

MASM Error - error-message-text in: filename

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

The error-message-text is the name of the file being handled when the error occurred.

Date format (Code 114)

Device full (Code 108)

APPENDIX B

```
Device name (Code 102)

Device offline (Code 105)

File in use (Code 112)

File name (Code 107)

File not found (Code 110)

File not open (Code 113)

File system (Code 104)

Hard data (Code 101)

Line too long (Code 115)

Lost file (Code 106)

Operation (Code 103)

Unknown device (Code 109)

Runtime Errors
```

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

```
Internal Error
```

Usually caused by an arithmetic check. If it occurs, notify Zenith Software Consultation.

```
Out of Memory
```

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

APPENDIX B

NUMERICAL ORDER LIST OF ERROR MESSAGES

CODE	MESSAGE
0	Block nesting error
1	Extra characters on line
2	Register already defined
3	Unknown symbol type
4	Redefinition of symbol
5	Symbol is multi-defined
6	Phase error between passes
7	Already had ELSE clause
8	Not in conditional block
9	Symbol not defined
10	Syntax error
11	Type illegal in context
12	Should have been group name
13	Must be declared in pass 1
14	Symbol type usage illegal
15	Symbol already different kind
16	Symbol is reserved word
17	Forward reference is illegal
18	Must be register
19	Wrong type of register
20	Must be segment or group
21	Symbol has no segment
22	Must be symbol type
23	Already defined locally
24	Segment parameters are changed
25	Not proper align/combine type
26	Reference to mult defined
27	Operand was expected
28	Operator was expected
29	Division by 0 or overflow
30	Shift count is negative
31	Operand types must match
32	Illegal use of external
33	Must be record field name
34	Must be record or field name

APPENDIX B

Error Code Numbers and Messages

CODE	MESSAGE
5	Operand must have size
36	Must be var, label or constant
37	Must be structure field name
38	Left operand must have segment
39	One operand must be const
40	Operands must be same or 1 abs
41	Normal type operand expected
42	Constant was expected
43	Operand must have segment
44	Must be associated with data
45	Must be associated with code
46	Already have base register
47	Already have index register
48	Must be index or base register
49	Illegal use of register
50	Value is out of range
51	Operand not in IP segment
52	Improper operand type
53	Relative jump out of range
54	Index displ, must be constant
55	Illegal register value
56	No immediate mode
57	Illegal size for item
58	Byte register is illegal
59	CS register illegal usage
60	Must be AX or AL
61	Improper use of segment reg.
62	No or unreachable CS
63	Operand combination illegal
64	Near JMP/CALL to different CS
65	Label can't have seg. override
66	Must have opcode after prefix
67	Can't override ES segment
68	Can't reach with segment req
69	Must be in segment block
70	Can't use EVEN on BYTE seg.
71	Forward needs override

PPEND B

CODE	MESSAGE
	Illegal value for DUP count
73	Symbol already external
74	DUP is too large for linker
75	Usage of ? (indeterminate) bad
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

APPENDIX C

LINK Error Messages

All errors cause the link session to abort. Therefore, after the cause is found and corrected, LINK must be rerun.

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

Probably caused by a bad object file.

```
BAD NUMERIC PARAMETER
```

The numeric value is not in digits.

```
CANNOT OPEN TEMPORARY FILE
```

LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not change the disk that will receive the list.MAP file.

```
ERROR: DUP RECORD TOO COMPLEX
```

The DUP record in assembly language module is too complex. Simplify the DUP record in the 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 instruction. Edit the assembly language source and reassemble.

```
INPUT FILE READ ERROR
```

Probably caused by a bad object file.

APPENDIX C

```
INVALID OBJECT MODULE
```

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

```
SYMBOL DEFINED MORE THAN ONCE
```

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 may not exceed 348K bytes and the number of segments may not exceed 255.

```
REQUESTED STACK SIZE EXCEEDS 64K
```

Specify a size less than 64K bytes with the /STACK switch.

```
SEGMENT-SIZE-EXCEEDS 64K
```

64K bytes is the addressing system limit.

```
SYMBOL TABLE CAPACITY EXCEEDED
```

Very many and/or very 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 10 Groups.

APPENDIX C

```
TOO MANY LIBRARIES SPECIFIED
```

The limit is 8.

```
TOO MANY PUBLIC SYMBOLS
```

The limit is 1024.

```
TOO MANY SEGMENTS OR CLASSES
```

The limit is 256 (Segments and Classes taken together).

```
UNRESOLVED EXTERNALS: <list>
```

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

```
VM READ ERROR
```

This is created by a disk problem; not LINK caused.

```
WARNING: NO STACK SEGMENT
```

None of the object modules specified contains a statement allocating stack space, but you entered the /STACK switch.

```
WRITE ERROR IN TMP FILE
```

No more disk space remaining to expand VM.TMP file.

```
WRITE ERROR ON RUN FILE
```

Usually, not enough disk space for run file.

APPENDIX D

LIB Error Messages

```
<symbol> is a multiply defined PUBLIC.  Proceed?
```

Two modules define the same PUBLIC symbol. You are asked to confirm the removal of the definition of the old symbol. A **No** response leaves the library in an undetermined state.

Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

```
Allocate error on VM.TMP
```

Out of space on the disk.

```
Cannot create extract file
```

No room in directory for extract file.

```
Cannot create list file
```

No room in directory for list file.

```
Cannot nest response file
```

'@filespec' in response (or indirect) file.

```
Cannot write library file
```

Out of space on the disk.

```
Error: An internal error has occurred.
```

Contact Zenith Software Consultation.

```
Fatal Error: Cannot open input file <filename>
```

Mistyped object file name.

```
Fatal Error: Module is not in the library
```

Trying to delete a module that is not in the library.

PE DIX D

```
Input file read error
```

Bad object module or faulty disk.

```
 atal Error: 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 on the disk.

```
Fatal Error: No library file specified
```

No response to Library File prompt.

```
Read error on VM.TMP
```

Disk not ready for read.

```
Symbol table capacity exceeded
```

Too many symbols (about 30K characters are allowed for symbols).

```
Too many object modules
```

More than 500 object modules.

```
Too many public symbols
```

1024 public symbols maximum.

```
Fatal Error: Write error on library/extract file
```

Out of space.

```
Write error on VM.TMP
```

Out of space.

APPENDIX E

CREF Error Messages

All errors cause CREF to abort. Control is returned to Z-DOS.

All error messages are displayed in the format:

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

where filename is the name of the file where the error occurs and error number is one of the numbers in the following list of errors.

Number	Error
101	Hard data error

Unrecoverable disk I/O error

101	Device name error

Illegal device specification (for example, X:FOO.CRF)

103	Internal error

Report to Zenith Software Consultation

104	Internal error

Report to Zenith Software Consultation

105	Device offline

Disk drive door open, no printer attached, and so on.

DOS Error Messages

Number	Error
106	Internal error

Report to Zenith Software consultation

108	Disk full
110	File not found
111	Disk is write protected
112	Internal error

Report to Zenith Software Consultation

113	Internal error

Report to Zenith Software Consultation

114	Internal error

Report to Zenith Software Consultation

115	Internal error

Report to Zenith Software Consultation

Memory Test Utility

The MEMTST utility program has been included with your Z-DOS System to aid you in diagnosing any problems that may arise with the RAM in your Z-100. MEMTST runs under Z-DOS and may be invoked as a standard command, in the form:

A: **MEMTST RETURN**

This test utility is completely menu-driven; after initially invoking the command, you instruct it to carry out different phases of its testing by selecting options from menus.

MEMTST consists of a master menu, and two test menus. The master menu appears on your screen like:

```
                    MEMTST version 1.00
            Copyright (C) 1982, Zenith Data Systems

        Functions Available:

        S - Test System Memory

        V - Test Video Memory

        E - Exit Program


        Select desired function <E>:
```

Pressing **RETURN** defaults to E — Exit program.

PP F

To test the system memory (memory that is allocated for the operating system and user programs), press the **S** key. The System Memory Test menu appears and will look like:

```
Unit contains xxxK of memory

System Memory Test

Functions Available:

A - Test all memory, 0-192K

F - Test first bank, 0-64K

S - Test second bank, 64-128K

T - Test third bank, 128-192K

E - Exit system memory test.

Select desired function <A>:
```

Here, if you press **RETURN** without making a selection, the default is A — Test all available memory. You can also select the first, second, and third banks of memory.

If you receive an error message from the A option, and that occurs somewhere within the first bank of memory, you may want to be able to test the other banks of memory. Or, if you do not want to wait while MEMTST checks all of memory, you may want to just check a single bank.

F (first bank) checks the memory configured from 0 to 64K; S (second bank) checks the memory configured from 64 to 128K; and T (third bank) checks the memory that is configured from 128 to 192K.

APPENDIX F

Press **E** to exit the System Memory Test, and the master menu will reappear.

If you select the V — Test Video Memory option from the master menu, the Video Memory Test menu appears on your screen. This third menu appears like:

```
Video Memory Test

Functions available:

<A> - Test all video RAM

<B> - Test blue video ram

<G> - Test green video ram

<R> - Test red video ram

<E> - Exit video memory test

Select desired function <A>:
```

The default here, is A — Test all video RAM. If you press **RETURN** without entering a selection, MEMTST checks all available video memory. This is the option you would most likely select if you have a black and white monitor connected to (or built into) your Z-100.

The other three test options are:

<G> to check the green plane of memory, which starts at address E00000;

<R> to check the red plane of video memory, which starts at address D00000H; and

<B> to check the blue plane of video memory, which starts at address C00000H.

Press **E** to exit the Video Memory Test, and the master menu will reappear.

Once a selection (other than E) has been made from either the System or Video Memory Test menus, the Test displays a message on your monitor screen. The test messages are shown below:

System Memory:

```
Testing n bank system memory . . . <Passed>/<Failed>
```

where n is a number showing which bank of memory is currently being tested; and where either `Passed` (in regular video) or `Failed` (in reverse video) will appear on your screen at the outcome of the tests on the current bank of memory.

Video Memory:

```
Testing video memory X plane . . . <passed>/<failed>
```

where X is a letter showing which video plane of memory is currently being tested; and where either `passed` (in regular video) or `failed` (in reverse video) will appear on your screen at the outcome of the tests on the current video plane of memory.

If the test fails on one bank or plane, you will receive an error message telling you the offset address (from logical 0000H of that bank or plane) in hex of the failure. MEMTST also states what it expected to find at that address and what it actually found. The error message appears like:

```
Location of failure XXXXH
Expected data XXH     Actual data XXH
Hit return to continue test, any other key to abort
```

APPENDIX G

Instructions for Single Disk Drive Users

For single disk drive users the commands are exactly the same syntax as for two drive users. The difference lies in your perception of the "arrangement" of the drives.

You must think of this system as having two disk drives: drive A and drive B. However, instead of A and B designating physical disk drive mechanisms, the A and B designate disks. Therefore, when you specify drive B while operating on drive A (the prompt is A:), Z-DOS prompts you to "switch drives" by swapping disks.

The prompts are:

```
Place disk A in drive B:.

Hit any key when ready.

Place disk B in drive A:.

Hit any key when ready.
```

These procedures apply to any Z-DOS COMMAND commands (both system and file) that can request or direct a different drive as a part of its syntax. These commands include:

CHKDSK [d:]
COPY [</x>] <filespec> [d:][<filespec>]
DEL <filespec>
DIR [<filespec>] [</x>]
DSKCOMP [d:] [d:]
DSKCOPY [</x>] [d:] [d:]
ERASE <filespec>
FORMAT [d:][</x>...]
REN [<filespec>][<filespec>]
RENAME [<filespec>][<filespec>]
TYPE [<filespec>]

Also, if any of these commands are used in a batch file and call for a different drive, the single disk drive procedures apply. Execution is halted and the appropriate prompt is displayed.

Example:

The following example may serve as an illustration for all of the commands listed above:

```
A: COPY COMMAND.COM B:      RETURN
Place disk B in drive A:.
Hit any Key when ready.
     1 File(s) copied
```

APPENDIX H

Disk Directory Structures and FCB Definition

Disk size	5.25	5.25	5.25	8	8
Tracks per inch	48	48	96	77	77
Sides	1	2	2	1	2
Bytes per sector	512	512	512	128	1024
Reserved sectors	1	1	1	1	1
FATS	2	2	2	2	2
Directory entries	64	112	144	104	192
Sectors per unit	1	2	4	4	1
Physical sectors	320	640	1280	2002	1232
Sectors per track	8	8	8	26	8
Tracks per side	40	40	80	77	77
FAT ID	0FFH	0FEH	0FDH	0FEH	0FDH
FAT 1 start sector	1 (01H)	1 (01H)	1 (01H)	4 (04H)	1 (01H)
FAT 2 start sector	2 (02H)	2 (02H)	2 (02H)	10 (0AH)	3 (03H)
Dir start sector	3 (03H)	3 (03H)	3 (03H)	16 (10H)	5 (05H)
Data start sector	7 (07H)	10 (0AH)	12 (0CH)	42 (2AH)	11 (0BH)

Reserved sectors (loader)	FAT 1	FAT 2	Directory	Data ...

Z-DOS Disk Structures

# Directory Records	64/4 = 16 (10H)	112/4 = 28d = 1Ch	144/4 = 36d = 24h	104/4 = 26d = 1Ah	192/4 = 48d = 30h
# FAT Entries					

APPENDIX H

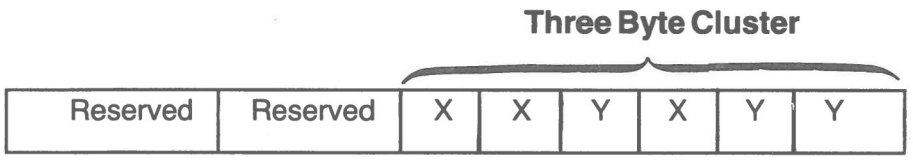
Disk Directory Structures and FCB Definition

FAT Structures

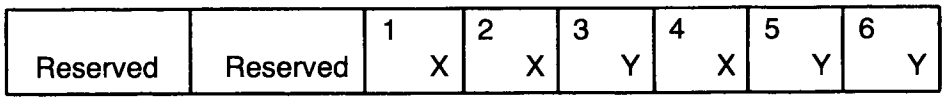
The FAT (File Allocation Table) uses a 12-bit entry for each allocation unit on the disk. These entries are packed, two for every three bytes. The contents of entry number N is found by:

1. multliplying N by 1.5;
2. adding the result to the base address of the allocation table;
3. fetching the 16 bit word at this address;
4. if N was odd, shift the word right four bits; and
5. mask to 12 bits.

Entry number zero is used as an end-of-file trap in the DOS and as a flag for disk structure. Entry 1 is reserved for future use. The first available allocation unit is assigned entry number two. Entries greater than 0FF8H are end-of-file marks; entries of zero are unallocated. Otherwise, the contents of a FAT entry is the number of the next allocation unit in the file.

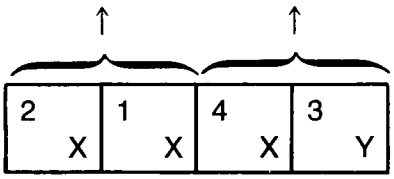


Loading flips the bytes in the cluster from:

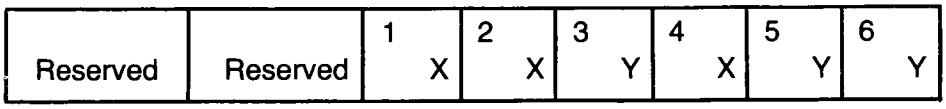


To:

"Y" is then masked -->

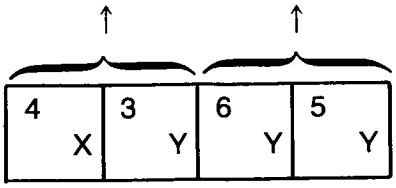


Or from:



To:

"X" is then masked-->



The following code shows how to compute the contents of the FAT entry in AX.

```
; RFATE--Read FAT entry
;
;     Call with:
;         DS:SI -> FAT
;         BX = entry number
;
;     Returns:
;         DI = contents of FAT entry
;
;     Uses: No registers are modified

RFATE   PROC    NEAR
        LEA     DI,[SI+BX]      ; Compute addr of FAT+entry number
        SHR     BX,1            ; Divide entry number by 2
        MOV     DI,[DI+BX]      ; Get word containing entry
        JNC     EVENE           ; Jmp if entry number even

ODDE:
        RCL     BX,1            ; Restore entry number
        SHR     DI,1            ; Shift word right by 4
        SHR     DI,1
        SHR     DI,1
        SHR     DI,1
        RET                     ; and return

EVENE:
        SHL     BX,1            ; Restore entry number
        AND     DI,0FFFH        ; Isolate lower 12 bits
        RET                     ;  and return
RFATE   ENDP
```

APPENDIX H

Disk Directory Structures and FCB Definition

Normal FCB

dr	f1	///	f8	e1	e2	e3	ex1	ex2	rs1	rs2	fs1	fs2	fs3	fs4
00	01	...	08	09	10	11	12	13	14	15	16	17	18	19

d1	d2	t1	t2	s1	///	s8	cr	r1	r2	r3	r4
20	21	22	23	24	...	31	32	33	34	35	36

The Z-DOS File Control Block (FCB) is defined as follows:

byte 0 (dr) — Drive Code. Zero specifies the default drive, 1=drive A, 2=drive B, etc.

bytes 1-8 (f1-f8) — Filename. If the file is less than eight characters, the name must be left justified with trailing blanks.

bytes 9-11 (e1-e3) — Extension to filename. If less than three characters, must be left justified with trailing blanks. May also be all blanks.

bytes 12-13 (ex1-ex2) — Current block (extent). This word (low byte first) specifies the current block of 128 records, relative to the start of the file, in which sequential disk reads and writes occur. If zero, then the first block of the file is being accessed; if one, then the second, etc. Combined with the current record field (byte 32) a particular logical record is identified.

bytes 14-15 (rs1-rs2) — Size of the record the user wishes to work with. This word may be filled immediately after an OPEN of the file if the default logical record size (128 bytes) is not desired. The Open and Create functions set this field to 128; it is also changed to 128 if a read or write is attempted with the field set to zero.

bytes 16-19 (fs1-fs4) — File size. This is the current size, in bytes, of the file. It may be read by user programs but must not be written by them.

APPENDIX H

Disk Directory Structures and FCB Definition

bytes 20-21 (d1-d2)

Date. This is normally the date of the last write to the file. It is set by all disk write operations and creates the "today's date". It is set by open to the date recorded in the disk directory for the file. User programs may modify this field after writing to a file but before closing it to change the date recorded in the disk directory.

The format of this 16-bit field is as follows: bits 0 – 4, day of month; bits 5 – 8, month of year; bits 9 – 15, year minus 1980. All zero means no date.

bytes 22-23 (t1-t2)

Time. Similar to Date, above. The format is bits 0–4, seconds/2; bits 5–10, minutes; bits 11–15, hours.

bytes 24-31 (s1-s8)

Reserved for Z-DOS.

byte 32 (cr)

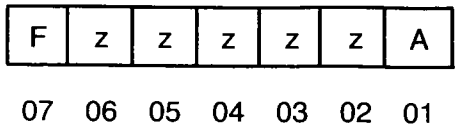
Current record. Identifies the record within the current block of 128 records that will be accessed with a sequential read or write function. See: Current Block, bytes 12-13.

bytes 33-36 (r1-r4)

Random record. This field must only be set if the file is to be accessed with a random read or write function. If the record size is greater than or equal to 64 bytes, only the first 3 bytes are used, as a 24-bit number representing the position in the file of a record. If the record size is less than 64 bytes, all 4 bytes are used as a 32-bit number of the same purpose. This field is thus large enough to address any byte in a file of the maximum size, 230 bytes.

The Extended FCB

Extended FCB:



The extended FCB is a special format used to search for files in the disk directory with special attributes. It consists of 7 bytes in front of a normal FCB, formatted as follows:

FCB-7 (F) — Flag. FF hex is placed here to signal an extended FCB.

FCB-6 to FCB-2 Zero field (z)

FCB-1 (A) — Attribute byte. If bit 1=1, hidden files will be included in directory searches. If bit 2 = 1, system files will be included in directory searches.

Any reference in the description of Z-DOS function calls to an FCB, whether opened or unopened, may use either a normal FCB or an extended FCB. A normal FCB has the same effect as an extended FCB with the attribute byte set to zero.

The values returned are:

```
0  write protect
2  disk not ready
4  data error
6  Seek error
8  Sector not found
A  Write fault
C  General disk failure
```

The registers will be set up for a BIOS disk call and the returned code will be in the lower half of the DI register with the upper half undefined. The user stack will look as follows from top to bottom:

IP
CS
FLAGS

Registers such that if an IRET is executed the DOS will respond according to (AL) as follows:

```
(AL)=0 ignore the error
    =1 retry the operation
       (IF THIS OPTION USED STACK DS, BX,
        CX AND DX MUST NOT BE MODIFIED!)
    =2 abort the program
```

AX
BX
CX
DX
SI
DI
BP
DS
ES

USER REGISTERS AT TIME OF REQUEST

IP
CS
FLAGS

The interrupt from the user to the DOS

Currently, the only error possible when AH bit 7=1 is a bad memory image of the file allocation table.

APPENDIX I

Interrupts, Function Calls and Entry Points

25 *Absolute disk read*. This transfers control directly to the DOS BIOS. Upon return, the original flags are still on the stack (put there by the INT instruction). This is necessary because return information is passed back in the flags. Be sure to pop the stack to prevent uncontrolled growth. For this entry point "records" and "sectors" are the same size. The request is as follows:

(AL)	Drive number (0=A, 1=B, etc.)
(CX)	Number of sectors to read
(DX)	Beginning logical record number
(DS:BX)	Transfer address

The number of records specified are transferred between the given drive and the transfer address. "Logical record numbers" are obtained by numbering each sector sequentially starting from zero and continuing across track boundries. For example, logical record number 0 is track 0 sector 1, whereas logical record number 12 hex is track 2 sector 3.

All registers but the segment registers are destroyed by this call. If the transfer was successful the carry flag (CF) will be zero. If the transfer was not successful CF=1 and (AL) will indicate the error as follows:

Return	Description
0	Attempt to write on write protected disk
2	Disk not ready
4	Data error
6	Seek error
8	Sector not found
C	General disk failure
A	Write fault

26 *Absolute disk write*. This vector is the counterpart to interrupt 25 above. Except for the fact that this is a write, the description above applies.

APPENDIX I

Interrupts, Function Calls and Entry Points

INTERRUPTS

Z-DOS reserves interrupt types 20 to 3F hex for its use. This means absolute locations 80 to FF hex are the transfer address storage locations reserved by the DOS. The defined interrupts are as follows with all values in hex:

20 *Program termination (Normal Exit)*. This is the normal way to exit a program. This vector transfers to the logic in the Z-DOS for restoration of CTRL-C exit addresses to the values they had on entry to the program. All file buffers are flushed to disk. All files that have changed in length should have been closed (see function call 10 hex) prior to issuing this interrupt. If the changed file was not closed its length will not be recorded correctly in the directory. When this interrupt is executed, CS MUST point to the 100H parameter area.

21 *Function request*. See "Function Requests" on Page I.5.

22 *Terminate address*. The address represented by this interrupt (88-8B hex) is the address to which control will transfer when the program terminates. This address is copied into low memory of the segment the program is loaded into at the time this segment is created. If a program wishes to execute a second program, it must set the terminate address prior to creation of the segment the program will be loaded into. Otherwise, once the second program executes, its termination would cause transfer to its host's termination address.

23 *CTRL-C exit address*. If the user types **CTRL-C** during keyboard input or video output, "C" will be printed on the console and an interrupt type 23 hex will be executed.

APPENDIX I

Interrupts, Function Calls and Entry Points

If the CTRL-C routine preserves all registers, it may end with a return-from-interrupt instruction (IRET) to continue program execution. If functions 9 or 10 (buffered output and input), were being executed, then I/O will continue from the start of the line. When the interrupt occurs, all registers are set to the value they had when the original call to Z-DOS was made. There are no restrictions on what the CTRL-C handler is allowed to do, including Z-DOS function calls, so long as the registers are unchanged if IRET is used.

If the program creates a new segment and loads in a second program which itself changes the CTRL-C address, the termination of the second program and return to the first will cause the CTRL-C address to be restored to the value it had before execution of the second program.

24 *Fatal error abort vector*. When a fatal error occurs within Z-DOS, control will be transferred with an INT 24H. On entry to the error handler, AH will have its bit 7 = 0 if the error was a hard disk error (probably the most common occurrence), bit 7=1 if not. If it is a hard disk error, bits 0 – 2 include the following:

bit	0	0	if read, 1 if write
bit	2	1	AFFECTED DISK AREA
	0	0	Reserved area
	0	1	File allocation table
	1	0	Directory
	1	1	Data area

AL, CX, DX, and DS:BX will be setup to perform a retry of the transfer with INT 25H or INT 26H (on next page). DI will have a 16-bit error code returned by the hardware.

APPENDIX I

Interrupts, Function Calls and Entry Points

27 *Terminate but stay resident.* This vector is used by programs which are to remain resident when COMMAND regains control. Such a program is loaded as an executing COM file by COMMAND. After it has initialized itself, it must set DX to its last address plus one in the segment it is executing in, then execute an interrupt 27H. COMMAND will then treat the program as an extension of Z-DOS, and the program will not be overlaid when other programs are executed.

Function Requests

The user requests a function by placing a function number in the AH register, supplying additional information in other registers as necessary for the specific function, then executing an interrupt type 21 hex. When Z-DOS takes control, it switches to an internal stack. User registers except AX are preserved unless information is passed back to the requester as indicated in the specific requests. The user stack needs to be sufficient to accommodate the interrupt system. It is recommended that it be 80 hex in addition to the user needs.

There is an additional mechanism provided for programs that conform to CP/M calling conventions. The function number is placed in the CL register, other registers are set as normal according to the function specification, and an intrasegment call is made to location five in the current code segment.

This method is only available to functions which do not pass a parameter in AL and whose numbers are equal to or less than 36. Register AX is always destroyed if this mechanism is used, otherwise it is the same as normal function requests. The functions are as follows with all values in hex:

0 *Program terminate.* The terminate and CTRL-C exit addresses are restored to the values they had on entry to the terminating program. All file buffers are flushed, but files which have been changed in length but not closed will not be recorded properly in the disk directory. Control transfers to the terminate address.

APPENDIX I

Interrupts, Function Calls and Entry Points

1 *Keyboard input*. Waits for a character to be typed at the keyboard, then echos the character to the video device and returns it in AL. The character is checked for a CTRL-C. If this key is detected an interrupt 23 hex will be executed.

2 *Video output*. The character in DL is output to the video device. If a CTRL-C is detected after the output, an interrupt 23 hex will be executed.

3 *Auxiliary input*. Waits for a character from the auxiliary input device, then returns that character in AL.

4 *Auxiliary output*. The character in DL is output to the auxiliary device.

5 *Printer output*. The character in DL is output to the printer.

6 *Direct console I/O*. If DL is FF hex, the AL returns with keyboard input character if one is ready, otherwise 00. If DL is not FF hex, then DL is assumed to have a valid character which is output to the video device.

7 *Direct console input*. Waits for a character to be typed at the keyboard, then returns the character in AL. As with function 6, no checks are made on the character.

8 *Console input without echo*. This function is identical to function 1, except the key is not echoed.

9 *Print string*. On entry, DS:DX must point to a character string in memory terminated by a "$" (24 hex). Each character in the string will be output to the video device in the same form as function 2.

APPENDIX I

Interrupts, Function Calls and Entry Points

A *Buffered keyboard input*. On entry, DS:DX points to an input buffer. The first byte must not be zero and specifies the number of characters the buffer can hold. Characters are read from the keyboard and placed in the buffer beginning at the third byte. Reading the keyboard and filling the buffer continues until RETURN is typed. If the buffer fills to one less than the maximum, then additional keyboard input is ignored until a RETURN is typed. The second byte of the buffer is set to the number of characters received excluding the carriage return (0D hex), which is always the last character. Editing of this buffer is described in the main Z-DOS document under "template editing".

B *Check keyboard status*. If a character is available from the keyboard, AL will be FF hex, otherwise AL will be 00.

C *Character input with buffer flush*. First the keyboard type-ahead buffer is emptied. Then if AL is 1, 6, 7, 8, or 0A hex, the corresponding Z-DOS input function is executed. If AL is not one of these values, no further operation is done and AL returns 00.

D *Disk reset*. Flushes all file buffers. Unclosed files that have been changed in size will not be properly recorded in the disk directory until they are closed. This function need not be called before a disk change if all files which have been written have been closed.

E *Select disk*. The drive specified in DL (0=A, 1=B, etc.) is selected as the default disk. The number of drives is returned in AL.

F *Open file*. On entry, DS:DX points to an unopened file control block (FCB). The disk directory is searched for the named file and AL returns FF hex if it is not found. If it is found, AL will return a 00 and the FCB is filled as follows:

If the drive code was 0 (default disk), it is changed to actual disk used (A=1, B=2, etc.). This allows changing the default disk without interfering with subsequent operations on this file. The high byte of the current block field is set to zero. The size of the record to be worked with (FCB bytes E-F hex) is set to the system default of 80 hex. The size of the file, and the time and date are set in the FCB from information obtained from the directory.

It is the user's responsibility to set the record size (FCB bytes E-F) to the preferred size, if the default 80 hex is not appropriate. It is also the user's responsibility to set the random record field and/or current block and record fields.

10 *Close file*. This function must be called after file writes to ensure that all directory information is updated. On entry, DS:DX points to an opened FCB. The disk directory is searched and if the file is found, its position is compared with that kept in the FCB. If the file is not found in the directory, it is assumed that the disk has been changed and AL returns FF hex. Otherwise, the directory is updated to reflect the status in the FCB and AL returns 00.

11 *Search for the first entry*. On entry, DS:DX points to an unopened FCB. The disk directory is searched for the first matching name (name could have "?"s indicating any letter matched) and if none are found AL returns FF hex. Otherwise, locations at the disk transfer address are set as follows:

1. If the FCB provided for searching was an extended FCB, then the first byte is set to FF hex, then 5 bytes of zeros, then the attribute byte from the search FCB, then the drive number used (A=1, B=2, etc.), then the 32 bytes of the directory entry. Thus the disk transfer address contains a valid unopened extended FCB with the same search attributes as the search FCB.

2. If the FCB provided for searching was a normal FCB, then the first byte is set to the drive number used (A=1, B=2, etc.) and the next 32 bytes contain the matching directory entry. Thus the disk transfer address contains a valid unopened normal FCB.

Directory entries are formatted as follows:

Location	Bytes	Description
0	11	File name and extension
11	1	Attributes. Bits 1 or 2 make file hidden
12	10	Zero field (for expansion)
22	2	Time. Bits 0-4 = secs*2 5-10 = min 11-15 = hrs
24	2	Date. Bits 0-4 = day 5-8 = month 9-15 = year
26	2	First allocation unit
28	4	File size, in bytes. (30 bits max.)

12 *Search for the next entry.* After function 11 has been called and found a match, function 12 may be called to find the next match to an ambiguous request ("?"'s in the search filename). Both inputs and outputs are the same as function 11. The reserved area of the FCB keeps information necessary for continuing the search, so it must not be modified.

APPENDIX I

Interrupts, Function Calls and Entry Points

13 *Delete file*. On entry, DS:DX points to an unopened FCB. All matching directory entries are deleted. If no directory entries match, AL returns FF, otherwise AL returns 00.

14 *Sequential read*. On entry, DS:DX points to an opened FCB. The record addressed by the current block (FCB bytes C-D) and the current record (FCB byte 1F) is loaded at the disk transfer address, then the record address is incremented. If end-of-file is encountered AL returns either 01 or 03. A return of 01 indicates no data in the record, 03 indicates a partial record is read and filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

15 *Sequential write*. On entry, DS:DX points to an opened FCB. The record addressed by the current block and current record fields is written from the disk transfer address (or, records less than sector size are buffered for write when a sector's worth of data is accumulated). The record address is then incremented. If the disk is full AL returns with a 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

16 *Create file*. On entry DS:DX points to an unopened FCB. The disk directory is searched for an empty entry, and AL returns FF if none is found. Otherwise, the entry is initialized to a zero-length file, the file is opened (see function F), and AL returns 00.

17 *Rename file*. On entry, DS:DX points to a modified FCB which has a drive code and file name in the usual position, and a second filename starting 6 bytes after the first (DS:DX+11 hex) in what is normally a reserved area. Every matching occurrence of the first is changed to the second (with the restriction that two files cannot have the exact same name and extension). If "?"'s appear in the second name, then the corresponding positions in the original name will be unchanged. AL returns FF hex if no match was found, otherwise 00.

APPENDIX I

Interrupts, Function Calls and Entry Points

19 *Current disk*. AL returns with the code of the current default drive (0=A, 1=B, etc.).

1A *Set disk transfer address*. The disk transfer address is set to DS:DX. Z-DOS will not allow disk transfers to wrap around within the segment, nor to overflow into the next segment.

1B *Allocation table address*. On return, DS:BX points to the allocation table for the current drive, DX has the number of allocation units, and AL has the number of records per allocation unit, and CX has the physical size of the sector. At DS:[BX – 1], the byte before the allocation table, is the dirty byte for the table. If set to 01, it means the table has been modified and must be written back to disk. If 00, the table is not modified. Any programs which get the address and directly modify the table must set this byte to 01 in order for the changes to be recorded. This byte should NEVER be set to 00—instead, a DISK RESET function (#0D hex) should be performed to write the table and reset the bit.

21 *Random read*. On entry, DS:DX points to an opened FCB. The current block and current record are set to agree with the random record field, then the record addressed by these fields is loaded at the current disk transfer address. If end-of-file is encountered, AL returns either 01 or 03. If 01 is returned no more data is available. If 03 is returned, a partial record is available, filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

22 *Random write*. On entry, DS: DX points to an opened FCB. The current block and current record are set to agree with the random record field, then the record addressed by these fields is written (or in the case of records not the same as sector sizes—buffered) from the disk transfer address. If the disk is full AL returns 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer was aborted. AL returns 00 if the transfer was completed successfully.

APPENDIX I

Interrupts, Function Calls and Entry Points

23 *File size*. On entry, DS:DX points to an unopened FCB. The disk directory is searched for the first matching entry and if none is found, AL returns FF. Otherwise the random record field is set with the size of the file (in terms of the record size field rounded up) and AL returns 00.

24 *Set random record field*. On entry, DS:DX points to an opened FCB. This function sets the random record field to the same file address as the current block and record fields.

25 *Set vector*. The interrupt type specified in AL is set to the 4-byte address DS:DX.

26 *Create a new program segment*. On entry, DX has a segment number at which to set up a new program segment. The entire 100 hex area at location zero in the current program segment is copied into location zero in the new program segment. The memory size information at location 6 is updated and the current termination and CTRL-C exit addresses are saved in the new program segment starting at 0A hex.

27 *Random block read*. On entry, DS:DX points to an opened FCB, and CX contains a record count that must not be zero. The specified number of records (in terms of the record size field) are read from the file address specified by the random record field into the disk transfer address. If end-of-file is reached before all records have been read, AL returns either 01 or 03. A return of 01 indicates end-of-file and the last record is complete, a 03 indicates the last record is a partial record. If wrap-around above address FFFF hex in the disk transfer segment would occur, as many records as possible are read and AL returns 02. If all records are read successfully, AL returns 00. In any case CX returns with the actual number of records read, and the random record field and the current block/record fields are set to address the next record.

28 *Random block write*. Essentially the same as function 27 above, except for writing and a write-protect indication. If there is insufficient space on the disk, AL returns 01 and no records are written. If CX is zero upon entry, no records are written, but the file is set to the length specified by the Random Record field, whether longer or shorter than the current file size (allocation units are released or allocated as appropriate).

29 *Parse file name*. On entry DS:SI points to a command line to parse, and ES:DI points to a portion of memory to be filled in with an unopened FCB. Leading TABs and spaces are ignored when scanning. If bit 0 of AL is equal to 1 on entry, then at most one leading filename seperator will be ignored, along with any trailing TABs and spaces. The four filename separators are:

```
; , = +
```

If bit 0 of AL is equal to 1, then all parsing stops if a separator is encountered. The command line is parsed for a file name of the form d:filename.ext, and if found, a corresponding unopened FCB is created at ES:DI. The entry value of AL bits 1, 2, and 3 determine what to do if the drive, filename, and extension, respectively, are missing. In each case, if the bit is a zero and the field is not present on the command line, then the FCB is filled with a fixed value (0, meaning the default drive for the drive field; all blanks for the filename and extension fields). If the bit is a 1, and the field is not present on the command line, then that field in the destination FCB at ES:DI is left unchanged. If an asterisk "*" appears in the filename or extension, then all remaining characters in the name or extension are set to "?".

The following characters are illegal within Z-DOS file specifications:

```
" / [] + = ; ,
```

APPENDIX I

Interrupts, Function Calls and Entry Points

Control characters and spaces also may not be given as elements of file specifications. If any of these characters are encountered while parsing, or the period (.) or colon (:) is found in an invalid position, then parsing stops at that point.

If either "?" or "*" appears in the file name or extension, then AL returns 01, otherwise 00. DS:SI will return pointing to the first character after the filename.

2A *Get date*. Returns date in CX:DX. CX has the year, DH has the month (1=Jan, 2=Feb, etc.), and DL has the day. If the time-of-day clock rolls over to the next day, the date will be adjusted accordingly, taking into account the number of days in each month and leap years.

2B *Set date*. On entry CX:DX must have a valid date in the same format as returned by function 2A above. If the date is indeed valid and the set operation is successful, then AL returns 00. If the date is not valid, then AL returns FF.

2C *Get time*. Returns with time-of-day in CX:DX. Time is actually represented as four 8-bit binary quantities, as follows: CH has the hours (0-23), CL has minutes (0-59), DH has seconds (0-59), DL has 1/100 seconds (0-99). This format is easily converted to a printable form yet can also be calculated upon (e.g., subtracting two times).

2D *Set time*. On entry, CX:DX has time in the same format as returned by function 2C above. If any component of the time is not valid, the set operation is aborted and AL returns FF. If the time is valid, AL returns 00.

2E *Set/Reset Verify Flag*. On entry, DL must be 0 and AL has the verify flag: 0=no-verify, 1=verify after write. This flag is simply passed to the I/O system on each write, so its exact meaning is interpreted there.

ENTRY POINTS

Interrupt Entry Points

File: DEFIPAGE.ASM

```
                ;
                ; Define the interrupt page offsets
                ;

0000            IPAGE_SEG SEGMENT AT 0

0000                    ORG     0
0000              INT_ZERO      LABEL DWORD

                ;  Hardware defined interrupts

0000                    ORG     4*0
0000              INT_DIV       LABEL DWORD           ; Divide error
0004                    ORG     4*1
0004              INT_STEP      LABEL DWORD           ; Single step
0008                    ORG     4*2
0008              INT_NMI       LABEL DWORD           ; Non-maskable interrupt
000C                    ORG     4*3
000C              INT_BRK       LABEL DWORD           ; Breakpoint
0010                    ORG     4*4
0010              INT_OVFL      LABEL DWORD           ; Overflow error

                ;  MS-DOS defined interrupts

0080                    ORG     4*DOSI_TERM
0080              INT_TERM      LABEL DWORD           ; Terminate program function
0084                    ORG     4*DOSI_FUNC
0084              INT_FUNC      LABEL DWORD           ; Perform function
0088                    ORG     4*DOSI_TADDR
0088              INT_TADDR     LABEL DWORD           ; Resume addr on program termination
008C                    ORG     4*DOSI_CADDR
008C              INT_CADDR     LABEL DWORD           ; ^C handler
```

```
0090            ORG     4*DOSI_FERADDR
0090    INT_FERADDR     LABEL DWORD             ; Fatal error handler
0094            ORG     4*DOSI_ADREAD
0094    INT_ADREAD      LABEL DWORD             ; Absolute disk read
0098            ORG     4*DOSI_ADWRITE
0098    INT_ADWRITE     LABEL DWORD             ; Absolute disk write
009C            ORG     4*DOSI_TERMR
009C    INT_TERMR       LABEL DWORD             ; Terminate program (but stay resident) function

        ; Master 8259A interrupt controller defined interrupts

0100            ORG     4*ZM8259AI
0100    INT_ZM8259A     LABEL DWORD             ; Base of Master 8259A interrupts

0100            ORG     4*(ZM8259AI+ZINTEI)
0100    INT_EI          LABEL DWORD             ; Parity or S-100 pin 98
0104            ORG     4*(ZM8259AI+ZINTPS)
0104    INT_PS          LABEL DWORD             ; Processor swap
0108            ORG     4*(ZM8259AI+ZINTTIM)
0108    INT_TIM         LABEL DWORD             ; Timer
010C            ORG     4*(ZM8259AI+ZINTSLV)
010C    INT_SLV         LABEL DWORD             ; Slave 8259A
0110            ORG     4*(ZM8259AI+ZINTSA)
0110    INT_SA          LABEL DWORD             ; Serial port A
0114            ORG     4*(ZM8259AI+ZINTSB)
0114    INT_SB          LABEL DWORD             ; Serial port B
0118            ORG     4*(ZM8259AI+ZINTKD)
0118    INT_KD          LABEL DWORD             ; Keyboard/Display/Light pen
011C            ORG     4*(ZM8259AI+ZINTPP)
011C    INT_PP          LABEL DWORD             ; Parallel port

        ; Slave 8259A interrupt controller defined interrupts

0120            ORG     4*ZS8259AI
0120    INT_ZS8259A     LABEL DWORD             ; Base of Slave 8259A interrupts

0120            ORG     4*(ZS8259AI+0)
0120    INT_SLV0        LABEL DWORD             ; Slave line 0
0124            ORG     4*(ZS8259AI+1)
0124    INT_SLV1        LABEL DWORD             ; Slave line 1
```

```
0128            ORG     4*(ZS8259AI+2)
0128      INT_SLV2      LABEL DWORD             ; Slave line 2
012C            ORG     4*(ZS8259AI+3)
012C      INT_SLV3      LABEL DWORD             ; Slave line 3
0130            ORG     4*(ZS8259AI+4)
0130      INT_SLV4      LABEL DWORD             ; Slave line 4
0134            ORG     4*(ZS8259AI+5)
0134      INT_SLV5      LABEL DWORD             ; Slave line 5
0138            ORG     4*(ZS8259AI+6)
0138      INT_SLV6      LABEL DWORD             ; Slave line 6
013C            ORG     4*(ZS8259AI+7)
013C      INT_SLV7      LABEL DWORD             ; Slave line 7
013C    IPAGE_SEG ENDS

-------------------------------------------------------------------
```

The Z-DOS BIOS Entry Points (Function Calls)

The Z-DOS BIOS version 2.19 and above written by Zenith Data Systems for their Z-100 computer contains all the functions (entry points) as defined by MicroSoft in MS-DOS version 1.25 plus a few additional ones for control of peripheral devices. The MicroSoft defined functions include:

- Console input/output/status.
- Printer output.
- Auxiliary device input/output.
- Disk input/output/mapping.
- Date-time setting/reading.

The ZDS defined functions provide complete control over:

- Disk.
- Console.
- Printer.
- Auxiliary devices.

In addition, a defined address is provided for the version of the BIOS and for configuration information. Assembly language include files have been written to help in program development.

The first include file (DEFMS.ASM) defines:

- All the entry points to the BIOS.
- The function codes for "interrupt 21" (perform system call).

- The MS-DOS interrupts.
- The program header.
- The "User" file control block (FCB).
- The directory entries.
- The drive parameter table used by Z-DOS at initialization time.
- Defines the disk error codes.

```
File:     DEFMS.ASM
-------------------------------------------------------
        ;
        ; Definitions for MS-DOS
        ;

            IFDEF BIOS
                FBIOS = BIOS
            ELSE
= 0000          FBIOS = 0
            ENDIF

            IF NOT FBIOS

= 0400  LORGADDR        =       400H    ; Loader org address

        ;
        ; BIOS entry points
        ;

0000    BIOS_SEG SEGMENT AT 40H         ; Segment where the BIOS is located

        ; MicroSoft(MS) defined entry points

0000            ORG 0*3
0000      BIOS_INIT     LABEL FAR       ; Initialization routine (only exists
                                        ;   at boot time)
```

```
0003            ORG 1*3
0003      BIOS_STATUS   LABEL FAR       ; Console input status
0006            ORG 2*3
0006      BIOS_CONIN    LABEL FAR       ; Console input
0009            ORG 3*3
0009      BIOS_CONOUT   LABEL FAR       ; Console output
000C            ORG 4*3
000C      BIOS_PRINT    LABEL FAR       ; Printer output
000F            ORG 5*3
000F      BIOS_AUXIN    LABEL FAR       ; Aux input
0012            ORG 6*3
0012      BIOS_AUXOUT   LABEL FAR       ; Aux output
0015            ORG 7*3
0015      BIOS_READ     LABEL FAR       ; Disk input
0018            ORG 8*3
0018      BIOS_WRITE    LABEL FAR       ; Disk output
001B            ORG 9*3
001B      BIOS_DSKCHG   LABEL FAR       ; Disk change status
001E            ORG 10*3
001E      BIOS_SETDATE  LABEL FAR       ; Set current date
0021            ORG 11*3
0021      BIOS_SETTIME  LABEL FAR       ; Set current time
0024            ORG 12*3
0024      BIOS_GETDATE  LABEL FAR       ; Get current date
0027            ORG 13*3
0027      BIOS_FLUSH    LABEL FAR       ; Flush keyboard buffer
002A            ORG 14*3
002A      BIOS_MAPDEV   LABEL FAR       ; Device mapping
002D            ORG 15*3
002D      BIOS_MRES9    LABEL FAR       ; Reserved for MicroSoft entry points
0030            ORG 16*3
0030      BIOS_MRES8    LABEL FAR
0033            ORG 17*3
0033      BIOS_MRES7    LABEL FAR
0036            ORG 18*3
0036      BIOS_MRES6    LABEL FAR
0039            ORG 19*3
0039      BIOS_MRES5    LABEL FAR
003C            ORG 20*4
003C      BIOS_MRES4    LABEL FAR
```

```
003F                    ORG 21*3
003F    BIOS_MRES3      LABEL FAR
0042                    ORG 22*4
0042    BIOS_MRES2      LABEL FAR
0045                    ORG 23*3
0045    BIOS_MRES1      LABEL FAR

                ; Zenith Data System(ZDS) defined entry points

0048                    ORG 24*3
0048    BIOS_DSKFUNC    LABEL FAR       ; Disk function
004B                    ORG 25*3
004B    BIOS_PRNFUNC    LABEL FAR       ; PRN: (Printer) function
004E                    ORG 26*3
004E    BIOS_AUXFUNC    LABEL FAR       ; AUX: (modem) function
0051                    ORG 27*3
0051    BIOS_CONFUNC    LABEL FAR       ; CON: (console) function
0054                    ORG 28*3
0054    BIOS_ZRES4      LABEL FAR       ; Reserved for Zenith entry points
0057                    ORG 29*3
0057    BIOS_ZRES3      LABEL FAR
005A                    ORG 30*3
005A    BIOS_ZRES2      LABEL FAR
005D                    ORG 31*3
005D    BIOS_ZRES1      LABEL FAR
0060                    ORG 32*3
0060    BIOS_REL        LABEL BYTE      ; Bios release number in hex
                                        ;  (ie 012H is release 1.2x)
0061                    ORG OFFSET BIOS_REL+1
0061    BIOS_CTADDR     LABEL WORD      ; Addr of configuration information
0063                    ORG OFFSET BIOS_CTADDR+2
0063    BIOS_SEG ENDS
                ;
                ; Configuration vector
                ;

= 0000  CONFG_DSK       EQU 0           ; Addr of disk vector
= 0002  CONFG_PRN       EQU CONFG_DSK+2 ; Addr of PRN: configuration table
= 0004  CONFG_AUX       EQU CONFG_PRN+2 ; Addr of AUX: configuration table
= 0006  CONFG_CON       EQU CONFG_AUX+2 ; Addr of CON: configuration table
```

APPENDIX I

Interrupts, Function Calls and Entry Points

```
= 0008  CONFG_FNT       EQU CONFG_CON+2 ; Addr of Font information
= 0000    FNT_RAM         EQU 0           ; Ptr to font table in RAM
= 0004    FNT_ROM         EQU FNT_RAM+4   ; Ptr to font table in ROM
= 0008    FNT_SIZE        EQU FNT_ROM+4   ; Size of font table in ROM
= 000A    FNT_MSIZE       EQU FNT_SIZE+2  ; Space allocated for font table in RAM
= 000A  CONFG_CLOCK     EQU CONFG_FNT+2 ; Addr of Date and time fields
= 0000    BIOS_DATE       EQU 0           ; Days since Jan 1, 1980
= 0002    BIOS_HRS        EQU BIOS_DATE+2 ; Hours since midnight
= 0003    BIOS_MIN        EQU BIOS_HRS+1  ; Minutes
= 0004    BIOS_SEC        EQU BIOS_MIN+1  ; Seconds
= 0005    BIOS_HSEC       EQU BIOS_SEC+1  ; Hundredths of seconds(a word)
= 000C  CONFG_DOSTB     EQU CONFG_CLOCK+2; Addr of DOS disk tables
= 000E  CONFG_MCL       EQU CONFG..DOSTB+ 2; Addr of value for map control latch
= 0010  CONFG_SIZE      EQU CONFG_MCL+2;  ; Length of configuration vector

          ENDIF

= 0010  BIOS_CREL       EQU 10H         ; Current release of BIOS
= 0100  BIOS_WORKSP     EQU 256         ; Number if bytes needed for
                                        ;   workspace in BIOS
= 03A9  BIOS_RELDATE    EQU 937         ; Release date 7/26/82 (this changes
                                        ;   for each release)
= 0001  MS_SIZEMEM      EQU 1           ; Flag for DOS to size memory at init

        ;
        ;  System functions for "interrupt 21"
        ;    (Note: functions followed by "*" are
        ;           not CP/M compatable
        ;

= 0000  DOSF_TERM       EQU  0          ; Program terminate
= 0001  DOSF_CONIN      EQU  1          ; Console input
= 0002  DOSF_CONOUT     EQU  2          ; Console output
= 0003  DOSF_AUXIN      EQU  3          ; Aux input
= 0004  DOSF_AUXOUT     EQU  4          ; Aux output
= 0005  DOSF_PRINTOUT   EQU  5          ; Printer output
= 0006  DOSF_DRCIO      EQU  6          ; Direct console I/O
= 0007  DOSF_DRCI       EQU  7          ; * Direct console input
= 0008  DOSF_DRCINE     EQU  8          ; * Console input(no echo)
= 0009  DOSF_OUTSTR     EQU  9          ; Output string
```

APPENDIX I

Interrupts, Function Calls and Entry Points

```
= 000A  DOSF_INSTR      EQU 10          ; Input string
= 000B  DOSF_STCON      EQU 11          ; Status of console
= 000C  DOSF_CONINF     EQU 12          ; * Flush keyboard buffer and input
= 000D  DOSF_RSDISK     EQU 13          ; Disk system reset
= 000E  DOSF_SELDISK    EQU 14          ; Select default disk
= 000F  DOSF_OPFILE     EQU 15          ; Open file
= 0010  DOSF_CLFILE     EQU 16          ; Close file
= 0011  DOSF_SRHFI      EQU 17          ; Search for first
= 0012  DOSF_SRHNX      EQU 18          ; Search for next
= 0013  DOSF_DEFILE     EQU 19          ; Delete file
= 0014  DOSF_SEQREAD    EQU 20          ; Sequential read
= 0015  DOSF_SEQWRITE   EQU 21          ; Sequential write
= 0016  DOSF_CRFILE     EQU 22          ; Create file
= 0017  DOSF_REFILE     EQU 23          ; Rename file
= 0018  DOSF_24         EQU 24          ; * not used
= 0019  DOSF_GETDISK    EQU 25          ; Get default disk
= 001A  DOSF_SDIOA      EQU 26          ; Set disk I/O address
= 001B  DOSF_GFATA      EQU 27          ; * Get file allocation table addr

        ;* The remaining functions are not CP/M compatable
= 001C  DOSF_GFATA128   EQU 28          ; Get file allocation table addr
= 001D  DOSF_29         EQU 29          ; not used
= 001E  DOSF_30         EQU 30          ; not used
= 001F  DOSF_31         EQU 31          ; not used
= 0020  DOSF_32         EQU 32          ; not used
= 0021  DOSF_RANREAD    EQU 33          ; Random read
= 0022  DOSF_RANWRITE   EQU 34          ; Random write
= 0023  DOSF_GFSIZE     EQU 35          ; Get file size
= 0024  DOSF_SFPOS      EQU 36          ; Set file position
= 0025  DOSF_SIVEC      EQU 37          ; Set interrupt vector
= 0026  DOSF_CESEG      EQU 38          ; Create segment
= 0027  DOSF_RBLREAD    EQU 39          ; Random block read
= 0028  DOSF_RBLWRITE   EQU 40          ; Random block write
= 0029  DOSF_PARSE      EQU 41          ; Parse file name
= 002A  DOSF_GDATE      EQU 42          ; Get date
= 002B  DOSF_SDATE      EQU 43          ; Set date
= 002C  DOSF_GTIME      EQU 44          ; Get time
= 002D  DOSF_STIME      EQU 45          ; Set time
= 002E  DOSF_CVERF      EQU 46          ; Set/Reset verify flag
```

```
                ;
                ; Define the interrupts
                ;

= 0020  DOSI_TERM       EQU 20H         ; Program terminate
= 0021  DOSI_FUNC       EQU 21H         ; Perform a function
= 0022  DOSI_TADDR      EQU 22H         ; Terminate address
= 0023  DOSI_CADDR      EQU 23H         ; ^C Exit address
= 0024  DOSI_FERADDR    EQU 24H         ; Fatal error exit addr
= 0025  DOSI_ADREAD     EQU 25H         ; Absolute disk read
= 0026  DOSI_ADWRITE    EQU 26H         ; Absolute disk write
= 0027  DOSI_TERMR      EQU 27H         ; Program terminate, but stay resident

                ;
                ; Define the program header
                ;

= 0000  PHD_TERM        EQU 000H        ; Termination point (has INT 20H)
= 0002  PHD_MEMSIZE     EQU 002H        ; Memory size (first seg num
                                        ;   after end of mem)
= 0005  PHD_AFUNC       EQU 005H        ; Alternate function entry point
= 000A  PHD_EXADDR      EQU 00AH        ; Exit handler addr
= 000E  PHD_ABADDR      EQU 00EH        ; ^C handler addr
= 0012  PHD_FEADDR      EQU 012H        ; Fatal error handler addr
= 005B  PHD_STACK       EQU 05BH        ; End of stack area
= 005C  PHD_FCB1        EQU 05CH        ; First program argument
= 006C  PHD_FCB2        EQU 06CH        ; Second program argument
= 0080  PHD_DIOA        EQU 080H        ; Default disk transfer area
= 0100  PHD_CODESTART   EQU 100H        ; Start of code (Size of a PHD)

                ;
                ; Define the "User" File control block (FCB)
                ;

= 0000  FCB_DRIVE       EQU 0           ; Drive number
= 0001  FCB_FNAME       EQU FCB_DRIVE+1 ; File name
= 0009  FCB_EXT         EQU FCB_FNAME+8 ; Extension to file name
= 000C  FCB_CURBLK      EQU FCB_EXT+3   ; Current block
= 000E  FCB_RECSZ       EQU FCB_CURBLK+2 ; Record size
= 0010  FCB_FILSZ       EQU FCB_RECSZ+2 ; File size
```

```
= 0014  FCB_DATE        EQU FCB_FILSZ+4 ; Date file modified
= 0016  FCB_TIME        EQU FCB_DATE+2  ; Time file modified
= 0018  FCB_RES         EQU FCB_TIME+2  ; Reserved
= 0020  FCB_CURREC      EQU FCB_RES+8   ; Current record(in block)
= 0021  FCB_RANREC      EQU FCB_CURREC+1 ; Random record number
= 0025  FCB_SIZE        EQU FCB_RANREC+4 ; Size of a FCB

                ;
                ; Define the extended file control block
                ;

= 0000  XFCB_FLAG       EQU 0           ; Flag field
= 0001  XFCB_RES        EQU XFCB_FLAG+1 ; Reserved
= 0006  XFCB_ATTR       EQU XFCB_RES+5  ; Attribute byte
= 0002    XFCBA_HID       EQU 02H         ; Hidden files
= 0004    XFCBA_SYS       EQU 04H         ; System files
= 0007  XFCB_FCB        EQU XFCB_ATTR+1 ; Normal FCB
= 002C  XFCB_SIZE       EQU XFCB_FCB+FCB_SIZE ; Size of a XFCB

                ;
                ; Define the directory entries
                ;

= 0000  DE_FNAME        EQU 0           ; File name
= 0008  DE_EXT          EQU DE_FNAME+8  ; Extension to file name
= 000B  DE_ATTR         EQU DE_EXT+3    ; File attribute
= 0002    DEA_HID         EQU 02H         ; Hidden file
= 0004    DEA_SYS         EQU 04H         ; System file
= 000C  DE_RES          EQU DE_ATTR+1   ; Reserved
= 0016  DE_TIME         EQU DE_RES+10   ; Time the file was modified
= 0018  DE_DATE         EQU DE_TIME+2   ; Date the file was modified
= 001A  DE_START        EQU DE_DATE+2   ; Starting sector of file
= 001C  DE_FSIZE        EQU DE_START+2  ; File size
= 0020  DE_SIZE         EQU DE_FSIZE+4  ; Size of a DE (should be 32)
```

```
                ;
                ; Define the "Drive parameter table" for MS-DOS usage
                ;  (Used only by the BIOS at init time)

= 0000  DPT_SECIZ       EQU 0           ; Size in bytes of a physical sector
= 0002  DPT_CLUSIZ      EQU DPT_SECSIZ+2 ; Number of sectors in an
                                         ;   allocation unit
= 0003  DPT_RESSEC      EQU DPT_CLUSIZ+1 ; Number of reserved sectors at
                                         ;   start of disk
= 0005  DPT_FATCNT      EQU DPT_RESSEC+2 ; Number of FAT's
= 0006  DPT_MAXENT      EQU DPT_FATCNT+1 ; Number of directory entries
= 0008  DPT_DSKSIZ      EQU DPT_MAXENT+2 ; Number of physical sectors on
                                         ;   the disk
= 000A  DPT_SIZE        EQU DPT_DSKSIZ+2 ; Size of a DPT

                ;
                ; Define the disk errors
                ;

= 0000  DSKE_WRITEP     EQU 0           ; Write protect
= 0002  DSKE_NREADY     EQU 2           ; Not ready
= 0004  DSKE_DATA       EQU 4           ; Data error
= 0006  DSKE_SEEK       EQU 6           ; Seek
= 0008  DSKE_SECT       EQU 8           ; Sector not found
= 000A  DSKE_WFAULT     EQU 10          ; Write fault
= 000C  DSKE_OTHER      EQU 12          ; Anything else
------------------------------------------------------------------------
```

APPENDIX I

Interrupts, Function Calls and Entry Points

Entry Points Defined

BIOS_INIT—System initialization

BIOS_INIT is the entry point used by the boot loader to pass control to the BIOS. This entry point may be called only from the boot loader during system initialization. The following functions are performed by BIOS_INIT:

1. All devices (except for the video) are initialized.
2. The disk drive tables are fixed up to account for characteristics of the boot device.
3. The DOS is read in from the disk.
4. The DOS initialization routine is called.
5. The character font table and the keyboard mapping are setup.
6. The program "COMMAND.COM" is loaded and control is passed to it.

BIOS_STATUS—Console input status

This routine checks to see if a character is ready at the console. If so, that character is returned. Once a character has been returned with this call, that same character is returned every time the call is made until a BIOS_CONIN call is made to read the character.

Returns:

"Z" set—no character ready
"Z" clear—AL = first character in input queue

Uses:

No registers are modified.

BIOS_CONIN—Console input

This routine waits for a character from the console.

Returns:

AL = character from the console

Uses:

No registers are modified.

BIOS_CONOUT—Console output

This routine waits for the console to be ready and then outputs a character to it.

Call with:

AL = character to output

Uses:

No registers are modified.

BIOS_PRINT—Printer output

This routine waits for the printer to be ready and then outputs a character to it.

Call with:

AL = character to output.

Uses:

No registers are modified.

BIOS_AUXIN—Aux input

This routine waits for a character to be ready at the auxiliary device and then reads it.

Returns:

AL = character read.

Uses:

No registers are modified.

BIOS_AUXOUT—Aux output

This routine waits for the auxiliary device to be ready and then outputs a character to it.

Call with:

AL = character to write.

Uses:

No registers are modified.

BIOS_READ—Disk input

This routine is used to read sectors from a specified disk device. Up to one segment of data may be read in one call.

Call with:

AL = Device number
CX = Number of sectors to read
DS = First logical sector to read
DS:BX = buffer to place data

Returns:

CX = Number of sectors not read
"CY" clear—operation succeeded
"CY" set—operation failed, AL = error code (see above in file DEFDSK.ASM)

Uses:

All registers may be modified (other than segment registers)

BIOS_WRITE—Disk output

This routine is used to write sectors from a specified disk device. Up to one segment of data may be written in one call.

Call with:

AL = Device number
AH = Verify flag: 0 = no verify, 1 = verify after write (not currently implemented)
CX = Number of sectors to write
DX = First logical sector to write
DS:BX = buffer to get data from

Returns:

CX = Number of sectors not written
"CY" clear—operation succeeded
"CY" set—operation failed, AL = error code (see above in file DEFDSK.ASM

Uses:

All registers may be modified (other than segment registers)

BIOS_DSKCHG—Disk change status

Check if the specified disk has been changed.

Call with:

AL = drive number

Returns:

CY clear (normal exit)
AH = −1, if disk has been changed
AH = 0, if it is not known if disk changed
AH = +1, if disk could not have changed
AL = drive number (can change or remain the same)
CY set (error exit)
AL = error code

Uses:

No other registers are modified.

BIOS_SETDATE—Set current date

Sets the current date as days since Jan 1, 1980.

Call with:

AX = the number of days since 1/1/80.

Uses:

No registers are modified.

BIOS_SETTIME—Set current time

Sets the current time of day.

Call with:

CH = hours (0-23)
CL = minutes (0-59)
DH = seconds (0-59)
DL = hundredths of seconds (0-99)

Uses:

No registers are modified.

BIOS_GETDATE—Get date and time

Returns the current date and time.

Returns:

AX = count of days since 1/1/80
CH = hours (0-23)
CL = minutes (0-59)
DH = seconds (0-59)
DL = hundredths of seconds (0-99)

Uses:

No other registers are modified.

BIOS_FLUSH—Flush keyboard input buffer

The input character queue associated with the console device is flushed.

Uses:

No registers are modified.

BIOS_MAPDEV—Map disk

Maps a disk driver given the device number and FAT ID.

Call with:

AL = I/O driver number
AH = First byte of FAT (range F8 to FF)

Returns:

AL = I/O driver for given media and drive

Uses:

No other registers are modified.

BIOS_DSKFUNC—Disk function

Used to execute an arbitrary disk function.

Call with:

The register pair ES:BX points to a parameter block (for all functions except GBIOSVEC and MAPDSK). The Parameter block has the following fields:

DSKPR_DRIVE (byte):
Logical drive number (– 1 to number of drives supported).
DSKPR_SECTOR (word):
Logical sector number. (On read/write track, used as side flag: 0 = side zero, 1 = side one).
DSKPR_COUNT (word):
Sector transfer count. (Must fit in segment)
DSKPR_BUFF (double word):
Address of I/O buffer. The first word is the offset and the second word is the segment.

AL = function to perform
AL = DSK_RESET—Reset the disk (home head)
AL = DSK_STATUS—Get disk status
AL = DSK_STEPIN—Step in head
AL = DSK_READ—Read sectors from the disk
AL = DSK_WRITE—Write sectors to the disk
AL = DSK_VERIFY—Not implemented
AL = DSK_FORMAT—Format track (write track)
AL = DSK_READTRK—Read track
AL = DSK_GBIOSVEC—Get addr of disk table vector
AL = DSK_MAPDSK—Maps drive number in AH
AL = DSK_SETFDC—Indicates drive has been formatted

Returns:

AX = Status of operation
CY clear—operation succeeded
CY failure—operation failed
For DSK_READ and DSK_WRITE:
DSKPR_COUNT = number of sectors nor read/written
DSKPR_BUFF = updated to next addr
For DSK_GBIOSVEC:
ES:BX -> vector of disk table addresses
For DSK_STATUS:
AH = aux status, AL = status
For DSK_MAPDSK:
AL = mapped device

Uses:

All registers may be used.

APPENDIX I

Interrupts, Function Calls and Entry Points

File: DEFDSK.ASM

```
------------------------------------------------------------------
------------------------------------------------------------------
        ;
        ;  Define Functions performed by Disk driver routines
        ;

= 0000  DSK_RESET       EQU 0                   ; Reset function
= 0001  DSK_STATUS      EQU DSK_RESET+1          ; Status function
= 0002  DSK_READ        EQU DSK_STATUS+1         ; Read function
= 0003  DSK_WRITE       EQU DSK_READ+1           ; Write function
= 0004  DSK_VERIFY      EQU DSK_WRITE+1          ; Verify function
= 0005  DSK_FORMAT      EQU DSK_VERIFY+1         ; Format(write track) function
= 0006  DSK_STEPIN      EQU DSK_FORMAT+1         ; Step in function
= 0007  DSK_READTRK     EQU DSK_STEPIN+1         ; Read track function
= 0008  DSK_GBIOSVEC    EQU DSK_READTRK+1        ; Get BIOS disk vector addr
= 0009  DSK_MAPDSK      EQU DSK_GBIOSVEC+        ; Get Logical to physical mapping
= 000A  DSK_SETFDC      EQU DSK_MAPDSK+1         ; Show that disk has been formatted
=       DSK_FMAX        EQU DSK_SETFDC+1         ; Max function value

        ;
        ;  Define the disk info block (one is needed for each drive)
        ;

= 000F  MAXDSK          EQU 15          ; Maximum number of disks
= 0002  MAXDSK5         EQU 2           ; Maximum 5 inch drives (0-1)
= 0004  MAXDSK8         EQU 4           ; Maximum 8 inch drives (2-3)
= 0000  DSK_STA         EQU 0           ; Status of last operaton
= 0100    DSKST_FNERR     EQU   0100H     ; Invalid function
= 0200    DSKST_ORERR     EQU   0200H     ; Improper order of function
= 0300    DSKST_DNERR     EQU   0300H     ; Invalid disk number
= 0400    DSKST_DTERR     EQU   0400H     ; Invalid disk type
= 0500    DSKST_NIERR     EQU   0500H     ; Function not implemented
= 0600    DSKST_NDERR     EQU   0600H     ; No disk in drive
= 0002  DSK_TYPE        EQU DSK_STA+2   ; Disk type
= 0000    DSK_TZ207       EQU 0           ; Z-207 type disk
= 0003  DSK_LTRK        EQU DKS_TYPE+   ; Last track
= 0004  DSK_LOPT        EQU DKS_LTRK+   ; Last operation
= 0001    DSK_OWR         EQU   01H       ; Write was last op
```

APPENDIX I

Interrupts, Function Calls and Entry Points

```
= 0002    DSK_ORD        EQU   02H       ; Read was last op
= 0004    DSK_ORS        EQU   04H       ; Reset was last op
= 0008    DSK_OSI        EQU   08H       ; Step in was last op
= 0010    DSK_OFT        EQU   10H       ; Format was last op
= 0020    DSK_ORT        EQU   20H       ; Read track was last op
= 0080    DSK_OUK        EQU   80H       ; Track is unknown
= 0005  DSK_FLAG       EQU DSK_LOPT+1  ; Flags
= 0001    DSK_FDS        EQU   01H       ; Disk is double sided
= 0002    DSK_FFS        EQU   02H       ; Drive can be fast stepped
= 0004    DSK_FDP        EQU   04H       ; Disk is 48 tpi and should be double stepped
= 0008    DSK_FWP        EQU   08H       ; Disk is software write protected
= 0010    DSK_FDC        EQU   10H       ; Force Disk has Changed next time
= 0020    DSK_FSL        EQU   20H       ; Skip head load on select
= 0006  DSK_SEL        EQU DSK_FLAG+   ; Command to select drive
= 0007  DSK_RS         EQU DSK_SEL+    ; Command to reset drive
= 0008  DSK_SPHI       EQU DSK_RS+1    ; Command to step in
= 0009  DSK_FMT        EQU DSK_SPHI+1  ; Command to format(write) a track
= 000A  DSK_RD         EQU DSK_FMT+1   ; Command to read a sector
= 000B  DSK_WR         EQU DSK_RD+1    ; Command to write a sector
= 000C  DSK_SK         EQU DSK_WR+1    ; Command to seek to a track
= 000D  DSK_SERR       EQU DSK_SK+1    ; Number of "soft" errors
= 000F  DSK_MAXT       EQU DSK_SERR+2  ; Maximum track number of drive
= 0010  DSK_NRETRY     EQU DSK_MAXT+1  ; Maximum retry count
= 0011  DSK_SPT        EQU DSK_NRETRY+1; Sectors per track
= 0012  DSK_BPS        EQU DSK_SPT+1   ; Number of bytes per sector
= 0014  DSK_BPWT       EQU DSK_BPS+2   ; Number of bytes per write track operation
= 0016  DSK_BPRT       EQU DSK_BPWT+2  ; Number of bytes per read track operation
= 0018  DSK_DELAY      EQU DSK_BPRT+2  ; Counter value for short delay
= 001A  DSK_LDELAY     EQU DSK_DELAY+2 ; Counter value for a long delay
= 001C  DSK_PORT       EQU DSK_LDELAY+2; Base Port number
= 001E  DSK_RDT        EQU DSK_PORT+2  ; Read track command
= 001F  DSK_IMGFLG     EQU DSK_RDT+1   ; Imaginary drive flag
= 0080    DSKIF_ID       EQU 80H         ; (0 - real drive; 1 - imaginary drive)
= 0040    DSKIF_DV       EQU 40H         ; (0 - disk is not in drive; 1 - disk is in drive)
= 0020    DSKIF_NM       EQU 20H         ; (0 - can map imag to drive; 1 - can't)
= 000F    DSKIF_DN       EQU 0FH         ; (Mask for disk in drive)
= 0020  DSK_TDSEL      EQU DSK_IMGFLG+1 ; Time to wait before deselecting drive(in 100ths of secs)
= 0022  DSK_SIZE       EQU DSK_TDSEL+2 ; Size of DSK
```

```
                ;
                ;  Define the parameter table passed to the disk drive routines
                ;

= 0000  DSKPR_DRIVE     EQU 0             ; Logical drive number
= 0001  DSKPR_SECTOR    EQU DSKPR_DRIVE+1 ; Logical sector number
= 0003  DSKPR_COUNT     EQU DSKPR_SECTOR+2 ; Sector transfer count
= 0005  DSKPR_BUFF      EQU DSKPR_COUNT+2 ; Buffer addr (offset,paragraph)
= 0009  DSKPR_SIZE      EQU DSKPR_BUFF+4 ; Size of this thing

-------------------------------------------------------------------------
```

BIOS_PRNFUNC—PRN function
BIOS_AUXFUNC—AUX function
BIOS_CONFUNC—CON function

These three entry points are used to perform any of five functions on the device after which they are named. The functions are write a character, read a character, get status, perform control type operation, and read a character, but leave it in the input queue (nondestructive read). The file DEFCHR.ASM, which is included below, has the needed definitions to use these entry points.

Call with:

AH = function to perform

AH = CHR_WRITE— Write character function
AL = character to write
AH = CHR_READ— Read character function
AH = CHR_STATUS— Status function
AL = Subfunction
AL = CHR_SFGS— Get status
AL = CHR_SFGC— Return configuration info to ES:BX
AH = CHR_CONTROL— Control function
AL = subfunction
AL = CHR_CFSU— Set up using new configuration info at ES:BX
AL = CHR_CFCI— Clear input
AL = CHR_CFCO— Clear output
AH = CHR_LOOK— Nondestructive read function

Returns:

CY clear (normal exit)
Write— nothing returned
Read— AL = character read
Status
Get status— AH = status, AL = raw status,
BH = input queue size, BL = chars in queue
Get config info— Config info copied to ES:BX
Control— nothing returned
Look— AL = first character in input queue

CY set (error exit)
AX ± error code

```
 File: DEFCHR.ASM
----------------------------------------------------------------

        ;
        ; DEFCHR - Definitions for the character devices (CON:, AUX:, and PRN:)
        ;

        ;  Define functions of BIOS_CONFUNC, BIOS_PRNFUNC, and BIOS_AUXFUNC

= 0000  CHR_WRITE       EQU 0           ; Write function
= 0001  CHR_READ        EQU CHR_WRITE+  ; Read function
= 0002  CHR_STATUS      EQU CHR_READ+   ; Status function
= 0000    CHR_SFGS        EQU 0           ; Get status subfunction
= 0001      CHRS_WA         EQU 00000001B   ; <ETX> sent, waiting for <ACK>
= 0002      CHRS_WD         EQU 00000010B   ; <DC3> seen, waiting for <DCI>
= 0004      CHRS_SN         EQU 00000100B   ; Sending nulls
= 0080      CHRS_TXR        EQU 10000000B   ; Transmitter ready to send data
= 0040      CHRS_RXR        EQU 01000000B   ; Receiver has data
= 0020      CHRS_RXOF       EQU 00100000B   ; Receiver queue overflow
= 0010      CHRS_RXE        EQU 00010000B   ; Other type of reciver error
= 0008      CHRS_TXE        EQU 00001000B   ; Transmitter error
= 0001    CHR_SFGC        EQU CHR_SFGS+1  ; Get configuration info subfunction
= 0003  CHR_CONTROL     EQU CHR_STATUS+1 ; Control function
= 0000    CHR_CFSU        EQU 0           ; Setup new configuration parms subfunction
```

```
= 0001    CHR_CFCI        EQU CHR_CFSU+1  ; Clear input subfunction
= 0002    CHR_CFCO        EQU CHR_CFCI+1  ; Clear output subfunction
= 0001  CHR_LOOK        EQU CHR_CONTROL+1; Nondestructive read function
=       CHR_FMAX        EQU CHR_LOOK    ; Maximum function number

        ; Configuration information packet

= 0000  CHRD_CLASS      EQU 0           ; Device class
= 0000    CHRDCL_CRT      EQU 0           ;Internal keyboard/display
= 0001    CHRDCL_SER      EQU CHRDCL_CRT+1; 2661 serial port
= 0002    CHRDCL_PAR      EQU CHRDCL_SER+1; PIA parallel port
=         CHRDCL_MAX      EQU CHRDCL_PAR ; Maximum class value
= 0001  CHRD_ATTR       EQU CHRD_CLASS+1 ; Attributes
= 0001    CHRDA_SPI       EQU 00000001B   ; Strip parity on input
= 0002    CHRDA_SPO       EQU 00000010B   ; Strip parity on output
= 0004    CHRDA_MLI       EQU 00000100B   ; Map lower to upper case on input
= 0008    CHRDA_MLO       EQU 00001000B   ; Map lower to upper case on output

        ; The remaining fields are used only with the 2661 serial ports
        ;  (except CHRD_NCHR and CHRD_NCNT which can be used by parallel printer)

= 0002  CHRD_PORT       EQU CHRD_ATTR+1 ; Port number
= 0004  CHRD_BAUD       EQU CHRD_PORT+2 ; Baud rate
= 0000    BD455           EQU 0           ; 45.5
= 0001    BD050           EQU 1           ; 50
= 0002    BD075           EQU 2           ; 75
= 0003    BD110           EQU 3           ; 110
= 0004    BD134           EQU 4           ; 134.5
= 0005    BD150           EQU 5           ; 150
= 0006    BD300           EQU 6           ; 300
= 0007    BD600           EQU 7           ; 600
= 0008    BD120           EQU 8           ; 1200
= 0009    BD180           EQU 9           ; 1800
= 000A    BD200           EQU 10          ; 2000
= 000B    BD240           EQU 11          ; 2400
= 000C    BD480           EQU 12          ; 4800
= 000D    BD960           EQU 13          ; 9600
= 000E    BD192           EQU 14          ; 19200
```

```
= 000F    BD384          EQU 15          ; 38400
=         BDMAX          EQU BD384       ; Maximum valid baud rate
= 0005  CHRD_HSHK      EQU CHRD_BAUD+1 ; Handshaking protocol
= 0000    CHRDH_NO       EQU 0           ; None
= 0001    CHRDH_EAH      EQU CHRDH_NO+1  ; <EXT>/<ACK>
= 0002    CHRDH_DCH      EQU CHRDH_EAH+1 ; <DC3>/<DC1> (CTRL-S/CTRL-Q)
= 0003    CHRDH_DCDH     EQU CHRDH_DCH+1 ; DCD(data carrier detect) high
= 0004    CHRDH_DCDL     EQU CHRDH_DCDH+1; DCD low
= 0005    CHRDH_DSRH     EQU CHRDH_DCDL+1; DSR(data set ready) high
= 0006    CHRDH_DSRL     EQU CHRDH_DSRH+1; DSR low
=         CHRDH_MAX       EQU CHRDH_DSRL  ; Maximum valid value
= 0006  CHRD_BCTL      EQU CHRDH_HSHK+1 ; Stop bits/parity char length
                                         ; (2661 Mode register 1)
= 00C0    CHRDB_SB       EQU 11000000B   ; Stop bits
= 0040      CHRDB_SB1      EQU 040H        ; 1 stop bit
= 0080      CHRDB_SB15     EQU 080H        ; 1.5 stop bits
= 00C0      CHRDB_SB2      EQU 0C0H        ; 2 stop bits
= 0020    CHRDB_PT       EQU 00100000B   ; Parity type(0=odd, 1=even)
= 0010    CHRDB_PC       EQU 00010000B   ; Parity contr(0=disabled, 1=enabled)
= 000C    CHRDB_CL       EQU 00001100B   ; Character length
= 0000      CHRDB_CL5      EQU 00H         ; 5 bits
= 0004      CHRDB_CL6      EQU 04H         ; 6 bits
= 0008      CHRDB_CL7      EQU 08H         ; 7 bits
= 000C      CHRDB_CL8      EQU 0CH         ; 8 bits
= 0007  CHRD_ECNT      EQU CHRD_BCTL+1 ; If <ETX>/<ACK> used, chars to
                                       ;   send before <ETX> sent
= 0008  CHRD_NCNT      EQU CHRD_ECNT+1 ; Number of NULLs to send after
                                       ;   CHRD_NCHR seen
= 0009  CHRD_NCHR      EQU CHRD_NCNT+1 ; Character after which to send NULLS
= 000A  CHRD_RES       EQU CHRD_NCHR+1 ; Reserved for future use
= 0010  CHRD_SIZE      EQU CHRD_RES+6  ; Size of a CHRD
```

APPENDIX I

Interrupts, Function Calls and Entry Points

```
        ; Error codes that are returned

= 0000  CHRE_ILGFH      EQU 0           ; Illegal function code in AH
= 0001  CHRE_ILGFL      EQU CHRE_ILGFH+1; Illegal function code in AL
= 0002  CHRE_NWR        EQU CHRE_ILGFL+1; No writes allowed to device
= 0003  CHRE_NRD        EQU CHRE_NWR+1  ; No reads allowed to device
= 0004  CHRE_BSUP       EQU CHRE_NDR+1  ; Bad set up parameters
= 0005  CHRE_WRB        EQU CHRE_BSUP+1 ; Device busy on write
= 0006  CHRE_RDNR       EQU CHRE_WRB+1  ; Device not ready on read
= 0007  CHRE_HTO        EQU CHRE_RDNR+1 ; Software handshake time out
= 0008  CHRE_ILR        EQU CHRE_HTO+1  ; Illegal response from device
= 0009  CHRE_IQE        EQU CHRE_ILR+1  ; Input queue empty
= 000A  CHRE_NIQ        EQU CHRE_IQE+1  ; Device has no input queue

       ;  Internal character device control table (It includes an embedded CHRD)

= 0000  CID_CHRD        EQU 0           ; A CHRD
= 0010  CID_CLASS       EQU CID_CHRD+CHRD_SIZE  ; Class of character device
                                        ;    (must be mult of 2)
= 0000    CIDCL_CRT       EQU CHRDCL_CRT*2 ; Internal video/keyboard
= 0002    CIDCL_SER       EQU CHRDCL_SER*2 ; 2661 serial port
= 0004    CIDCL_PAR       EQU CHRDCL_PAR*2 ; PIA parallel port
= 0012  CID_TYPE        EQU CID_CLASS+2 ; Special types
= 0000    CIDTY_NORM      EQU 0           ; Normal type
= 0001    CIDTY_CSP       EQU CIDTY_NORM+1 ; Special CRT
= 0014  CID_IPORT       EQU CID_TYPE+2  ; Input port
= 0016  CID_OPORT       EQU CID_IPORT+2 ; Output port
= 0018  CID_SPORT       EQU CID_OPORT+2 ; Status port
= 001A  CID_CPORT       EQU CID_SPORT+2 ; Control port
= 001C  CID_ST          EQU CID_CPORT+2 ; Status(see CHRD_SFGS for values)
= 001D  CID_IRM         EQU CID_ST+1    ; Input ready mask
= 001E  CID_IPM         EQU CID_IRM+1   ; Input polarity mask
= 001F  CID_ORM         EQU CID_IPM+1   ; Output ready mask
= 0020  CID_OPM         EQU CID_ORM+1   ; Output polarity mask
= 0021  CID_ECTR        EQU CID_OPM+1   ; Char counter for sending<ETX>
= 0022  CID_NCTR        EQU CID_ECTR+1  ; Null down countr
= 0023  CID_SIZE        EQU CID_NCTR+1  ; Size of the CID
```

```
                ;
                ; Define input queue for character devices
                ;

= 0000  CQ_SADDR        EQU 0           ; Addr of start of queue
= 0002  CQ_EADDR        EQU CQ_SADDR+2  ; Addr of end of queue
= 0004  CQ_QSIZE        EQU CQ_EADDR+2  ; Size of queue
= 0006  CQ_ELMTS        EQU CQ_QSIZE+2  ; Number of elements currently in queue
= 0008  CQ_STATUS       EQU CQ_ELMTS+2  ; Status (as defined under CHR_STATUS)
= 0009  CQ_FRONT        EQU CQ_STATUS+1 ; Addr of first element in queue
= 000B  CQ_REAR         EQU CQ_FRONT+2  ; Addr of last element in queue
= 000D  CQ_SIZE         EQU CQ_REAR+2   ; Size of a CQ

-------------------------------------------------------------------------------
```

```
  File: DEFCONFG.ASM
------------------------------------------------------------------------------
        ;
        ; Configuration type info
        ;

= 00B0  Z207A           EQU 0B0H        ; First Z-207 disk controller base port
          ; (See DEFZ207 to program controller)
= 00AE  Z217A           EQU 0AEH        ; Reserved for future use
= E000  ZGRNSEG         EQU 0E000H      ; Segment of green video plane
= D000  ZREDSEG         EQU 0D000H      ; Segment of red video plane
= C000  ZBLUSEG         EQU 0C000H      ; Segment of blue video plane
= 00D8  ZVIDEOG         EQU 0D8H        ; Video 68A21 port
          ; PA0 -> enable red display
          ; PA1 -> enable green display
          ; PA2 -> enable blue display
          ; PA3 -> not flash screen
          ; PA4 -> not write multiple red
          ; PA5 -> not write multiple green
          ; PA6 -> not write multiple blue
          ; PA7 -> disable video RAM
          ; PA7-PB0 -> LA15-LA8
          ; CA1 - not used
          ; CA2 -> clear screen
          ; CB1 - not used
          ; CB2 -> value to write (0 or 1) on clear screen
          ; (see DEF6821 to program the 6821
= 00DC  ZCRTC           EQU 0DCH        ; Video 6845 CRT-C port
          ; (see DEF6845 to program the 6845)
= 00DE  ZLPEN           EQU 0DEH        ; Light pen latch
= 0007    ZLPEN_BIT       EQU 00000111B   ; Bit hit by pen
= 00F0    ZLPEN_ROW       EQU 11110000B   ; Row hit by pen
= 00E0  ZPIA          EQU 0E0H        ; Parallel printer plus light pen and
                                        ;   video vertical retrace 68A21 port
          ; PA0 -> PDATA1
          ; PA1 -> PDATA2
          ; PA2 -> not STROBE
          ; PA3 -> not INIT
          ; PA4 <- VSYNC
```

```
                ; PA5 -> clear VSYNC flip flop
                ; PA6 <- light pen switch
                ; PA7 -> clear light pen flip flop
                ; PB0 <- BUSY
                ; PB1 <- not ERROR
                ; PB2 -> PDATA3
                ; PB3 -> PDATA4
                ; PB4 -> PDATA5
                ; PB5 -> PDATA6
                ; PB6 -> PDATA7
                ; PB7 -> PDATA8
                ; CA1 <- light pen hit (from flip flop)
                ; CA2 <- VSYNC (from flip flop)
                ; CB1 <- not ACKNLG
                ; CB2 <- BUSY
                ; (See DEF6821 to program the PIA)

= 00E4  ZTIMER          EQU 0E4H        ; Timer 8253 port
= 61A8    ZTIMEVAL        EQU 25000       ; 100ms divide by N value
                ; (See DEF8253 to program the 8253)
= 00FB  ZTIMERS         EQU 0FBH        ; Timer interrupt status port
= 0001    ZTIMERS0        EQU 001H        ; Timer 0 interrupt
= 0002    ZTIMERS2        EQU 002H        ; Timer 2 interrupt
= 00E8  ZSERA           EQU 0E8H        ; First 2661-2 serial port
= 00EC  ZSERB           EQU 0ECH        ; Second 2661-2 serial port
                ; (See DEFEP2 to program 2661-2)
= 00F2  ZM8259A         EQU 0F2H        ; Master 8259A interrupt controller port
= 0000    ZINTEI          EQU 0           ; Parity error or S-100 pin 98 interrupt
= 0001    ZINTPS          EQU 1           ; Processor swap interrupt
= 0002    ZINTTIM         EQU 2           ; Timer interrupt
= 0003    ZINTSLV         EQU 3           ; Slave 8259A interrupt
= 0004    ZINTSA          EQU 4           ; Serial port A interrupt
= 0005    ZINTSB          EQU 5           ; Serial port B interrupt
= 0006    ZINTKD          EQU 6           ; Keyboard, Display, or Light pen interrupt
= 0007    ZINTPP          EQU 7           ; Parallel port interrupt
                ; (See DEF8259A to program the 8259A)
= 0040    ZM8259AI        EQU 64            ; Base interrupt number for master
= 00F0  ZS8259A         EQU 0F0H        ; Secondary 8259A interrupt controller port
= 0048    ZS8259AI        EQU 72            ; Base interrupt number for slave
```

APPENDIX I

Interrupts, Function Calls and Entry Points

```
= 00F4  ZKEYBRD         EQU 0F4H        ; Keyboard port
= 00F4    ZKEYBRDD        EQU ZKEYBRD+0   ; Keyboard data port
= 00F5    ZKEYBRDC        EQU ZKEYBRD+1   ; Keyboard command port
= 0000      ZKEYRES         EQU 0           ; Reset command
= 0001      ZKEYARD         EQU 1           ; Autorepeat on command
= 0002      ZKEYARF         EQU 2           ; Autorepeat off command
= 0003      ZKEYKCO         EQU 3           ; Key click on command
= 0004      ZKEYKCF         EQU 4           ; Key click off command
= 0005      ZKEYCF          EQU 5           ; Clear keyboard FIFO command
= 0006      ZKEYCLK         EQU 6           ; Generate a click sound command
= 0007      ZKEYBEP         EQU 7           ; Generate a beep sound command
= 0008      ZKEYEK          EQU 8           ; Enable keyboard command
= 0009      ZKEYDK          EQU 9           ; Disable keyboard command
= 000A      ZKEYUDM         EQU 10          ; Enter UP/DOWN mode command
= 000B      ZKEYNSM         EQU 11          ; Enter normal scan mode command
= 000C      ZKEYEI          EQU 12          ; Enable keyboard interrupts command
= 000D      ZKEYDI          EQU 13          ; Disable keyboard interrupts command
= 00F5    ZKEYBRDS        EQU ZKEYBRD+1   ; Keyboard status port
= 0001      ZKEYOBF         EQU 001H        ; Output buffer not empty
= 0002      ZKEYIBF         EQU 002H        ; Input buffer full
= 00FC  ZMCL            EQU 0FCH        ; Memory control latch
= 0003    ZMCLMS          EQU 00000011B   ; Map select mask
= 0000      ZSM0            EQU 0           ; Map select 0
= 0001      ZSM1            EQU 1           ; Map select 1
= 0002      ZSM2            EQU 2           ; Map select 2
= 0003      ZSM3            EQU 3           ; Map select 3
= 000C    ZMCLRM          EQU 00001100B   ; Monitor ROM mapping mask
= 0000      ZRM0            EQU 0*4         ; Power up mode - ROM everywhere on reads
= 0004      ZRM1            EQU 1*4         ; ROM at top of every 64K page
= 0008      ZRM2            EQU 2*4         ; ROM at top of 8088's addr space
= 000C      ZRM3            EQU 3*4         ; Disable ROM
= 0010    ZMCLPZ          EQU 00010000B   ; 0=Set Parity to the zero state
= 0020    ZMCLPK          EQU 00100000B   ; 0=Disable parity checking circuity
```

APPENDIX I

```
= 00FD  ZHAL          EQU 0FDH        ; Hi-address latch
= 00FF    ZHAL85        EQU 0FFH        ; 8080 Mask
= 000F    ZHAL88        EQU 0F0H        ; 8088 Mask
= 00FE  ZPSP          EQU 0FEH        ; Processor swap port
= 0080    ZPSPPS        EQU 10000000B   ; Processor select (0=8085, 1=8088)
= 0000    ZPSPPS5       EQU 00000000B   ; Select 8085
= 0080    ZPSPPS8       EQU 10000000B   ; Select 8088
= 0002    ZPSPSI        EQU 00000010B   ; Generate interrupt on swapping
= 0001    ZPSPI8        EQU 00000001B   ; 8088 processes all interrupts
= 00FF  ZDIPSW        EQU 0FFH        ; Configuration dip switches
= 0007    ZDIPSWBOOT    EQU 00000111B   ; Boot device field
= 0008    ZDIPSWAB      EQU 00001000B   ; 1=Auto boot(0=Manual boot)
= 0070    ZDIPSWRES     EQU 01110000B   ; Reserved
= 0080    ZDIPSWHZ      EQU 10000000B   ; 0=60Hz(1=50HZ)

---------------------------------------------------------------
```

MONITOR-100 Subroutine Entry Points:

MTR-100 Global Subroutine Vectors (Address Offsets from FE000H Base)

Monitor Subroutine Vectors

File: DEFMTR.ASM

```
----------------------------------------------------------------

        ;
        ; Definitions for the Monitor ROM
        ;

        ; Entry points to ROM monitor

0000    MTR_SEG SEGMENT AT 0FE01H       ; Segment addr for Monitor ROM calls
0000            ORG     000H            ; Reset function
0000      MTR_RES        LABEL   FAR    ; Reset function
0005            ORG     005H
0005      MTR_MON        LABEL   FAR    ; Monitor call
000A            ORG     00AH
000A      MTR_SWIM       LABEL   FAR    ; Trace/breakpoint handler
000F            ORG     00FH
000A      MTR_DCRT       LABEL   FAR    ; Dumb display output
0014            ORG     014H
0014      MTR_DKBD       LABEL   FAR    ; Dumb keyboard handler
0019            ORG     019H
0019      MTR_SCRT       LABEL   FAR    ; Smart display output
001E            ORG     01EH
001E      MTR_SKBD       LABEL   FAR    ; Smart keyboard input
0023            ORG     023H
0023      MTR_TTY_INTR  LABEL   FAR     ; Vertical retrace interrupt handler
0023    MTR_SEG ENDS
```

```
0000    MTR_D_SEG SEGMENT AT 0          ; ROM Monitor data segment(not really located at 0)
0000            ORG     000H

        ; Monitor Parameters

0000      MTR_WIP       LABEL   FAR     ; Far jump to wild interrupt handler
0000                    DB 5 DUP(?)     ; the far jump
0005      MTR_VER       DB ?            ; BCD version of ROM monitor
= 0001      MTR_CVER    EQU 01H         ; Lowest version BIOS can run on
0006      MTR_DS_SIZE   DW ?            ; Size of the ROM monitor data segment
```

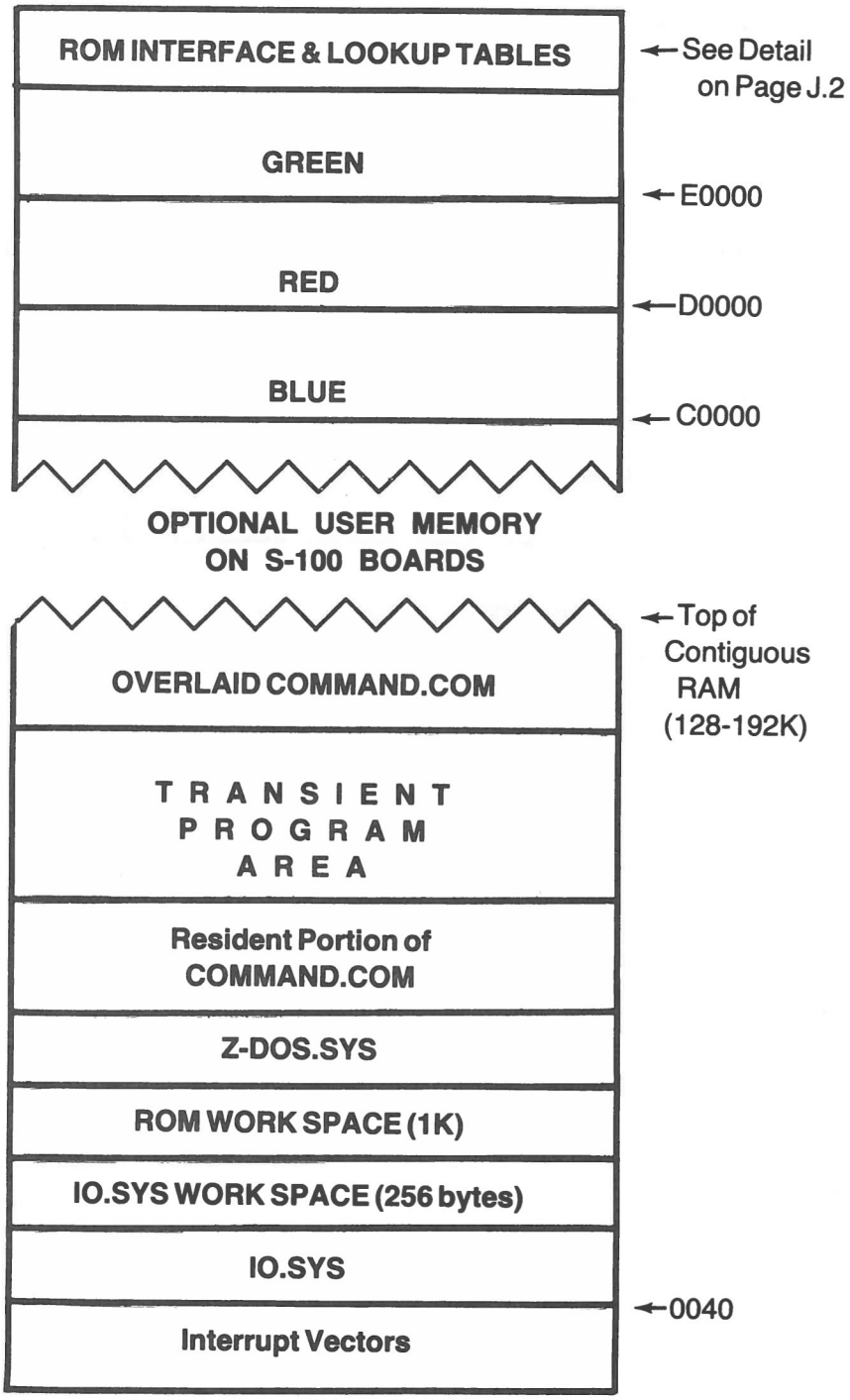
APPENDIX I
Interrupts, Function Calls and Entry Points

```
        ; Boot parameters

0008      MTR_BINDX     DB ?            ; Boot device index
0009      MTR_BPORT     DB ?            ; Boot device base port number
000A      MTR_BSTRING   DB 80 DUP(?)    ; Boot string
005A      MTR_BUNIT     DB ?            ; Boot unit number

        ; Pointers to All sorts of things

005B      MTR_DCI       DD ?            ; Addr of Display Character Initialization Routine
005F      MTR_DFC       DD ?            ; Addr of Display Font Character Routine
0063      MTR_DXMTC      DD ?           ; Addr of Dumb Keyboard Transmit Character Routine
0067      MTR_EDC        DD ?           ; Addr of Erase Display Character Routine
006B      MTR_EMEC       DD ?           ; Addr of Extended-Mode Escape Character Handler Routine
006F      MTR_FONT       DD ?           ; Addr of Character Font table
= 07E0      MTR_FNT_SIZE  EQU 9*(235-' ') ; Size of reserved font table(number of bytes copied from rom fon

0073      MTR_MDC       DD ?            ; Addr of Move Display Characters Routine
0077      MTR_MDL       DD ?            ; Addr of Display Line Routine
007B      MTR_PROMPT    DD ?            ; Addr of Display ROM Monitor Prompt Routine
007F      MTR_RDC       DD ?            ; Addr of Read Displayed Character Routine
0083      MTR_SXMTC     DD ?            ; Addr of Smart Keyboard Transmit Character Routine
0087      MTR_UIES      DD ?            ; Addr of Unimplemented Escape Sequence Handler Routine
008B      MTR_XCA       DD ?            ; Addr of Transmit Character Attributes Routine

        ; If version = 1, next word is not present, and all references must have
        ;       -2 added to them for labels beyond this point

008F      MTR_FNTSIZ    DW ?            ; Size of FONT in bytes (If version > 1)
0091      MTR_KYB       DW 256 DUP (?)  ; Keyboard map table
0191      MTR_CHR       DW 256 DUP (?)  ; Display map table
0291      MTR_HORP      DW ?            ; Horizontal position of cursor (column)
0292      MTR_VERP      DW ?            ; Vertical position of cursor (row)
0293    MTR_D_SEG ENDS

0000    IPAGE_SEG SEGMENT AT 0          ; The interrupt area page
03FE            ORG     03FEH
03F3    MTR-DS          LABEL WORD       ; Location that contains monitor DS value

03FE    IPAGE_SEG ENDS

-----------------------------------------------------------------
```

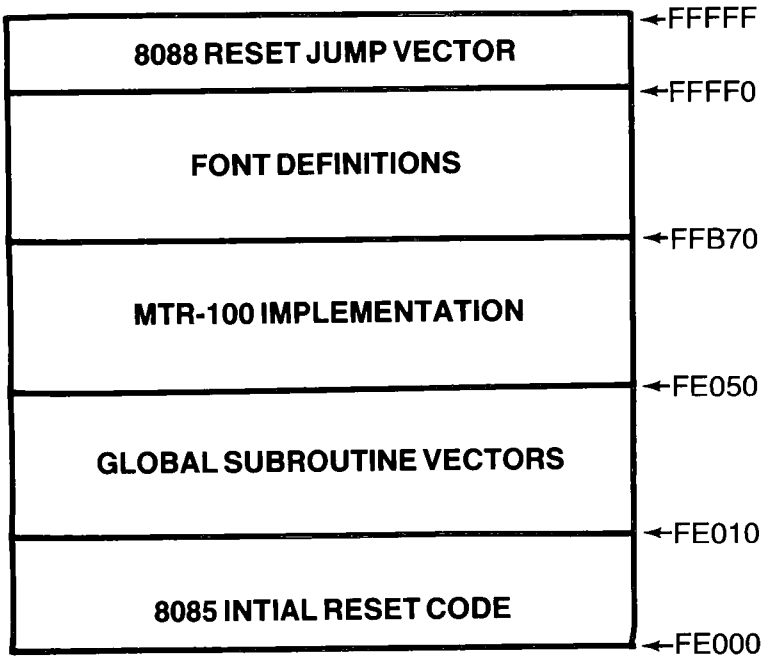
APPENDIX J

System Structure and Memory Maps

System Memory Map

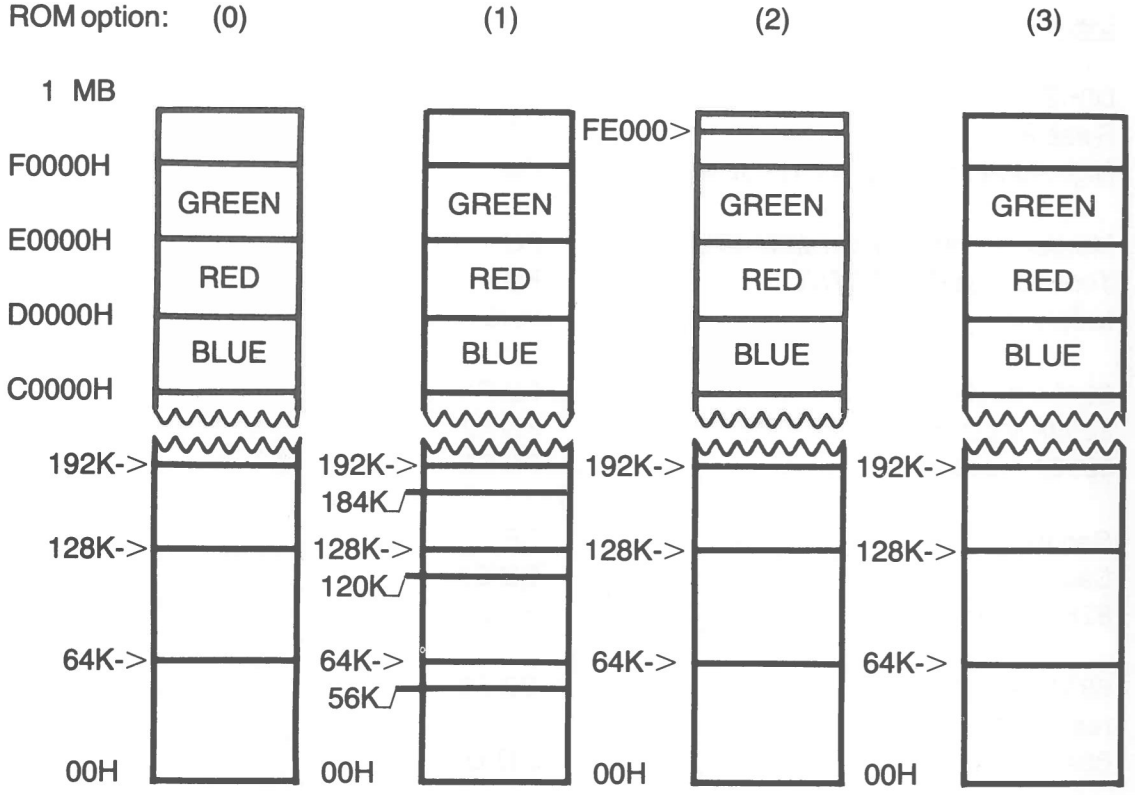


ROM Interface & Lookup Tables



APPENDIX J
System Structure and Memory Maps

Z-100 Memory Map by ROM Option



MEMCTL:

(0)	(1)	(2)	(3)
Bit 3 = 0 Bit 2 = 0 (All ROM)	Bit 3 = 0 Bit 2 = 1 (8K at Top of each Page)	Bit 3 = 1 Bit 2 = 0 (8K at Top 8088 Address)	Bit 3 = 1 Bit 2 = 1 (No ROM)

RAM MEMCTL: Bit 0 = MAPSEL0 Bit 1 = MAPSEL1

Option:	(0)	(1)	(2)
	Bit 1 = 0 Bit 0 = 0 (Power Up Contiguous Addresses)	Bit 1 = 0 Bit 0 = 1 (Replaces 0-48K with Seg 64K-112K)	Bit 1 = 1 Bit 0 = 0 (Replaces 0-48K with Seg 112K-160K)

(These keep 48K-64K constant)

I/O Port Assignments

Device Name	Port Address
DIP Switch	FF
Swap Port (PSP)	FE
High Address Latch (HIGHADDR)	FD
Memory Control Latch (MEMCTL)	FC
Timer Status (TIMRSTAT)	FB
reserved	fa-f6
8041A Keyboard	F5-F4
8259A Master	F3-F2
8259A Slave	F1-F0
Serial B	EF-EC
Serial A	EB-E8
8253 Timer	E7-E4
68A21 Parallel	E3-E0
reserved	df-de
6845 CRTC	DD-DC
Video 68A21	DB-D8
reserved	d7-c0
Secondary Z-207	BF-B8
Primary Z-207	B7-B0
reserved	af-a8

Keyboard Port Addresses and Command Summary

Port	Address
Command Port	D5
Status Port	D5
Data Port	D4

Command	Code
Reset	00
Autorepeat On	01
Autorepeat Off	02
Key Click On	03
Key Click Off	04
Clear FIFO	05
Click	06
Beep	07
Enable Keyboard	08
Disable Keyboard	09
Key Up/Down Mode	0A
Normal Scan Mode	0B

MACRO-86 Table of Directives

Memory Directives

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

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

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

        <name>  PROC [NEAR]
        <name>  PROC [FAR]

<proc-name>     ENDP

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

        <name>  GROUP <segment-name>[,...]
        <name>  SEGMENT [<align>][<combine>][<class>]

 <seg-name>     ENDS
                EVEN
                ORG <exp>

        <name>  STRUC

<struc-name>    ENDS
```

Macro Directives

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

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

Conditional Directives

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

Listing Directives

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

Attribute Operators

Override operators

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

Value Returning Operators

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

Record Specific operators

Shift-count — <record-fieldname>
    <record-fieldname>
MASK
    MASK <record-fieldname>
WIDTH
    WIDTH <record-fieldname>
    WIDTH <record>

Precedence of Operators

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

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside:
   parenthesis ( )
   angle brackets < >
   square brackets [ ]
   structure variable operand: <variable>.<field>

2. segment override operator: colon (:)

3. PTR, OFFSET, SEG, TYPE, THIS

4. HIGH, LOW

5. *, /, MOD, SHL, SHR

6. +, − (both unary and binary)

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

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT,.TYPE

8088 (8086) Instructions (Alphabetic)

The mnemonics are listed alphabetically with their full names. The 8086 instructions are also listed in groups based on the type of arguments the instruction takes, (see Appendix M).

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

Mnemonic	Full Name
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JCXZ	Jump on CX zero
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus

Mnemonic	Full Name
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero
LOOPZ	LOOP while zero
MOV	Move
MOVS	Move byte or word (of string)
MOVBS	Move byte (string)
MOVSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
NOT	NOT
OR	OR
OUT	Output byte or word
POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
REPE	Repeat on equal
REPZ	Repeat on zero
REPNE	Repeat on not equal
REPNZ	Repeat on not zero
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SCAS	Scan byte or word (of string)

Mnemonic	Full Name
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Subtract
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

APPENDIX M

8088 (8086) Instructions (by Argument)

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

The following abbreviations are used in these lists:

OP	=	opcode; instruction mnemonic
reg	=	byte register (AL,AH,BL,BH,CL,CH,DL,DH)
or		word register (AX,BX,CX,DX,SI,DI,BP,SP)
r/m	=	register or memory address or indexed and/or based
accum	=	AX or AL register
immed	=	immediate
mem	=	memory operand
segreg	=	segment register (CS,DS,SS,ES)

General Operand Instructions

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

CALL and JUMP Type Instructions

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

Relative jumps

Argument Type

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

Mnemonics

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

Loop instructions

Same as Relative Jumps

Mnemonics

LOOP LOOPE LOOPZ LOOPNE LOOPNZ

Return Instruction

Mnemonic	Argument Type
RET	[immed] (optional, number of words to POP)

No Operand Instructions

Mnemonics

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

Load Instructions

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

Move Instructions

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

Push and Pop Instructions

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

Shift/Rotate Type Instructions

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

Input/Output Instructions

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

Increment/Decrement Instructions

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

Arithmetic – Multiply/Division/Negate/Not

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

Interrupt Instruction

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

Exchange instruction

Mnemonic	Argument Types
XCHG	XCHG accum,reg XCHG reg,accum XCHG reg,r/m XCHG r/m,reg

Miscellaneous Instructions

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

String Primitives

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

Mnemonics	Argument Types
CMPS	CMPS byte-word,byte-word (CMPS right operand is ES)
LODS	LODS byte/word,byte/word (LODS one argument = no ES)
MOVS	MOVS byte/word,byte/word (MOVS left operand is ES)
SCAS	SCAS byte/word,byte/word (SCAS one argument = ES)
STOS	STOS byte/word,byte/word (STOS one argument = ES)

Repeat Prefix to String Instructions

Mnemonics

LOCK REP REPE REPZ REPNE REPNZ

APPENDIX N

Character Font Files

To make the Z-100 more useful, seven alternate character fonts have been provided – Danish, English, French, German, Italian, Spanish, and Swedish. These are the files on Distribution Disk II that have .CHR extensions and a filename of the language (e.g., FRENCH.CHR is the French Character Font).

To use the alternate font, boot your Z-100 with a Z-DOS system disk. After the system has booted, rename the correct font file of the language you need to ALTCHAR.SYS, like:

A: **COPY <filename>.CHR=ALTCHAR.SYS RETURN**

The next time that you boot from this disk, the new font and keyboard mapping will be reconfigured to match the language set that you have chosen. For instance, for German, you enter:

A: **COPY GERMAN.CHR=ALTCHR.SYS RETURN**

The alternate font is implemented by IO.SYS after it has initialized the hardware, Z-DOS and the disks. It then looks at the disk to see if there is a file named ALTCHAR.SYS on the booted disk.

If no ALTCHAR.SYS file is found, IO.SYS continues its regular functions and loads COMMAND.COM.

If ALTCHAR.SYS is found, IO.SYS read the file and changes the mapping for both the font and the key codes.

The format of the alternate font file is in two basic parts – keyboard map and the font index. The keyboard mapper occurs first in the form:

Keyboard	Map
<code>	<swap>
.	.
.	.
.	.
FF	FF

where code is the value generated by the keyboard processor; swap is the value that code is to be mapped to; and FFH is the terminator for both code and swap data.

The font index follows the keyboard map and is a one byte font index followed by a nine byte description. The font information terminates with FFH as the font index, or an End-of-file (EOF) terminator.

ASCII Character and Escape Sequence Codes

Control Characters

Dec	Hex	ASCII Char	Control Character	Z-100 KEY	Z-DOS Usage Description
000	00H	NUL	CTRL-@		
001	01H	SOH	CTRL-A		
002	02H	STX	CTRL-B		
003	03H	ETX	CTRL-C		Aborts current command.
004	04H	EOT	CTRL-D		
005	05H	ENQ	CTRL-E		
006	06H	ACK	CTRL-F		
007	07H	BEL	CTRL-G		
008	08H	BS	CTRL-H	BACK SPACE	Removes last character from command line, and erases character from video screen.
009	09H	HT	CTRL-I	TAB	
010	0AH	LF	CTRL-J	LINE FEED	Inserts physical end-of-line, but does not empty command line. Uses LINE FEED to extend the current logical line beyond the physical limitation of one terminal line.
011	0BH	VT	CTRL-K		
012	0CH	FF	CTRL-L		
013	0DH	CR	CTRL-M	RETURN	
014	0EH	SO	CTRL-N		Cancels echoing of output to line printer.
015	0FH	SI	CTRL-O		
016	10H	DLE	CTRL-P		Echoes terminal output to line printer.
017	11H	DC1	CTRL-Q		
018	12H	DC2	CTRL-R		

APPENDIX

ASCII Character and Escape Sequence Codes

Dec	Hex	ASCII Char	Control Character	Z-100 KEY	Z-DOS Usage Description
019	13H	DC3	CTRL-S		Suspends display of output to terminal screen. (Any other key resumes display.)
020	14H	DC4	CTRL-T		
021	15H	NAK	CTRL-U		
022	16H	SYN	CTRL-V		
023	17H	ETB	CTRL-W		
024	18H	CAN	CTRL-X		Cancels the current line, empties the command line, and then outputs a back slash (\), RETURN and LINE FEED. The template used by special editing commands is not affected.
025	19H	EM	CTRL-Y		
026	1AH	SUB	CTRL-Z		
027	1BH	ESC	CTRL-[	ESCAPE	
028	1CH	FS	CTRL-\		
029	1DH	GS	CTRL-]		
030	1EH	RS	CTRL-^		
031	1FH	US	CTRL--		

Printable Characters

Dec	Hex	CHR	Name
032	20H	SP	Space
033	21H	!	Exclamation point
034	22H	"	Quotation mark
035	23H	#	Number sign
036	24H	$	Dollar sign
037	25H	%	Percent sign
038	26H	&	Ampersand
039	27H	'	Acute accent or Apostrophe
040	28H	(	Opening parenthesis
041	29H	)	Closing parenthesis
042	2AH	*	Asterisk
043	2BH	+	Plus sign

Dec	Hex	CHR	Name
044	2CH	,	Comma
045	2DH	-	Hyphen or Minus sign
046	2EH	.	Period or Decimal point
047	2FH	/	Slash
048	30H	0	Number zero
049	31H	1	Number one
050	32H	2	Number two
051	33H	3	Number three
052	34H	4	Number four
053	35H	5	Number five
054	36H	6	Number six
055	37H	7	Number seven
056	38H	8	Number eight
057	39H	9	Number nine
058	3AH	:	Colon
059	3BH	;	Semicolon
060	3CH	<	Less than or Left angle bracket
061	3DH	=	Equal sign
062	3EH	>	Greater than or Right angle bracket
063	3FH	?	Question mark
064	40H	@	At sign
065	41H	A	Letter A
066	42H	B	Letter B
067	43H	C	Letter C
068	44H	D	Letter D
069	45H	E	Letter E
070	46H	F	Letter F
071	47H	G	Letter G
072	48H	H	Letter H
073	49H	I	Letter I
074	4AH	J	Letter J
075	4BH	K	Letter K
076	4CH	L	Letter L
077	4DH	M	Letter M
078	4EH	N	Letter N
079	4FH	O	Letter O
080	50H	P	Letter P
081	51H	Q	Letter Q
082	52H	R	Letter R

Dec	Hex	CHR	Name
083	53H	S	Letter S
084	54H	T	Letter T
085	55H	U	Letter U
086	56H	V	Letter V
087	57H	W	Letter W
088	58H	X	Letter X
089	59H	Y	Letter Y
090	5AH	Z	Letter Z
091	5BH	[	Left bracket
092	5CH	\	Back slash
093	5DH	]	Right bracket
094	5EH	^	Caret
095	5FH	_	Underscore
096	60H	`	Grave Accent
097	61H	a	Letter a
098	62H	b	Letter b
099	63H	c	Letter c
100	64H	d	Letter d
101	65H	e	Letter e
102	66H	f	Letter f
103	67H	g	Letter g
104	68H	h	Letter h
105	69H	i	Letter i
106	6AH	j	Letter j
107	6BH	k	Letter k
108	6CH	l	Letter l
109	6DH	m	Letter m
110	6EH	n	Letter n
111	6FH	o	Letter o
112	70H	p	Letter p
113	71H	q	Letter q
114	72H	r	Letter r
115	73H	s	Letter s
116	74H	t	Letter t
117	75H	u	Letter u
118	76H	v	Letter v
119	77H	w	Letter w
120	78H	x	Letter x

Dec	Hex	CHR	Name
121	79H	y	Letter y
122	7AH	z	Letter z
123	7BH	{	Left brace
124	7CH	¦	Stile
125	7DH	}	Right brace
126	7EH	~	Tilde
127	7FH	DEL	DELETE

Z-100 Escape Sequence Functions

Sequence	Description
ESC ^	Transmit Current Line
ESC –	Transmit Character at Cursor
ESC i 0	Zenith Identify Terminal Type

Z-100 Responses:

```
ESC i E <banks> <size>
```

where <banks> is either 1 (1 Bank of VRAM) or 3 (3 banks of VRAM)
where <size> is A (32K VRAM parts) or B (64K VRAM parts)

3.2 Characteristic Escape Sequences

ESC m <fore> <back>

Set Foreground and Background Colors where fore is foreground color; where back is background color; and the color is in the range 0 through 7:

0 = Black
1 = Blue
2 = Red
3 = Magenta
4 = Green
5 = Cyan
6 = Yellow
7 = White

ESC x ;	Set Non-Blinking Cursor
ESC x <	Disable Keyboard Autorepeat
ESC x ?	Enable Key Expansion (Keyboard generates Escape Sequences)

Sequence	Description
ESC y ;	Set Blinking Cursor
ESC y <	Enable Keyboard Autorepeat
ESC y ?	Disable Key Expansion (Keyboard generates 8-bit characters)

Z-100	H/Z-19 KEY	Normal	Shifted
F0	ERASE	ESC J	ESC E
F1	F1	ESC S	ESC 1 A
F2	F2	ESC T	ESC 1 B
F3	F3	ESC U	ESC 1 C
F4	F4	ESC V	ESC 1 D
F5	F5	ESC W	ESC 1 E
F6	BLUE	ESC P	ESC 1 F
F7	RED	ESC Q	ESC 1 G
F8	GRAY	ESC R	ESC 1 H
F9	---	ESC O I	ESC 1 I
F10	---	ESC O J	ESC 1 J
F11	---	ESC O K	ESC 1 K
F12	---	ESC O L	ESC 1 L
I/D Char	IC/DC	ESC @/	ESC N
I/D Line	IL/DL	ESC L	ESC M

Z-100	H/Z-19 KEY	Normal	Shifted
Up Arrow	Up Arrow	ESC A	ESC A
Dn Arrow	Dn Arrow	ESC B	ESC B
Rt Arrow	Rt Arrow	ESC C	ESC C
L + Arrow	L + Arrow	ESC D	ESC D
HOME	HOME	ESC H	ESC H
BREAK	---	ESC ¦	ESC ¦
HELP	---	ESC ~	ESC ~

Escape Sequences for H/Z-19 and Z-100 Terminals

Sequence	Function	Used By: H/Z-19(h)/Z-100(z)
ESC #	Transmit page	hz
ESC / K	Response: VT52 identify	hz
ESC 0 I	Function key #9	z
ESC 0 J	Function key #10	z
ESC 1 A	Shift function key #1	z
ESC 1 B	Shift function key #2	z
ESC 1 C	Shift function key #3	z
ESC 1 D	Shift function key #4	z
ESC 1 E	Shift function key #5	z
ESC 1 F	Shift function key #6	z
ESC 1 G	Shift function key #7	z

Sequence	Function		Used By: H/Z-19(h)/Z-100(z)
ESC 1 H	Shift function key #8		z
ESC 1 I	Shift function key #9		z
ESC =	Enter alternate keypad mode		hz
ESC >	Exit alternate keypad mode		hz
ESC ? M	"ENTER"		hz
ESC ? n	"."	Key Names	hz
ESC ? p	"0"		hz
ESC ? q	"1"		hz
ESC ? r	"2"		hz
ESC ? s	"3"		hz
ESC ? t	"4"		hz
ESC ? u	"5"		hz
ESC ? v	"6"		hz
ESC ? w	"7"		hz
ESC ? x	"8"		hz
ESC ? y	"9"		hz
ESC @	Enter insert character mode		hz
ESC A	Cursor up		hz
ESC B	Cursor down		hz

Sequence	Function	Used By: H/Z-19(h)/Z-100(h)
ESC C	Cursor right	hz
ESC D	Cursor left	hz
ESC E	Clear entire display	hz
ESC F	Enter graphic mode	hz
ESC G	Exit graphic mode	hz
ESC H	Move cursor to home position	hz
ESC I	Reverse index (reverse scroll)	hz
ESC J	Erase from cursor position to end of screen	hz
ESC K	Erase from cursor position to end of line	hz
ESC L	Perform an insert line	hz
ESC M	Perform a delete line	hz
ESC N	Perform a delete character	hz
ESC O	Exit insert character mode	hz
ESC P	Function key #6 (BLUE)	hz
ESC Q	Function key #7 (RED)	hz
ESC R	Function key #8 (WHITE)	hz
ESC S	Function key #1	hz
ESC T	Function key #2	hz

sequence	Function	Used By: H/Z-19(h)/Z-100(z)
ESC U	Function key #3	hz
ESC V	Function key #4	hz
ESC W	Function key #5	hz
ESC Y <ln#> <col#>	Cursor addressing	hz
ESC Z	Identify as VT52 (ESC / K)	hz
ESC [	Enter hold screen mode	hz
ESC \	Exit hold screen mode	hz
ESC]	Transmit status line	hz
ESC ^	Transmit current line	z
ESC _	Transmit character at cursor	z
ESC i 0	Request terminal type	z
ESC i E	Response: for Z-100	z
ESC j	Save current cursor position	hz
ESC k	Restore cursor position	hz
ESC l	Erase entire line	hz
ESC m <fore> <back>	Set foreground and background colors	z
ESC n	Cursor position report	hz
ESC o	Erase from beginning of line to cursor	hz

Sequence	Function	Used By: H/Z-19(h)/Z-100(z)
ESC p	Enter reverse video mode	hz
ESC q	Exit reverse video mode	hz
ESC t	Enter keypad shifted mode	hz
ESC u	Exit keypad shifted mode	hz
ESC v	Enter wrap at end of line mode	hz
ESC w	Exit wrap at end of line mode	hz
ESC x 1	Enable 25th line	hz
ESC x 2	Disable keyboard click	hz
ESC x 3	Enter hold screen mode	hz
ESC x 4	Set "block" cursor	hz
ESC x 5	Disable cursor	hz
ESC x 6	Enter keypad shifted mode	hz
ESC x 7	Enter keypad alternate mode	hz
ESC x 8	Enable auto line feed on carriage return	hz
ESC x 9	Enable auto carriage return on line feed	hz
ESC x ;	Set "non-blinking" cursor	z
ESC x <	Disable keyboard auto repeat	z
ESC x ?	Enable key expansion	z

Sequence	Function	Used By: H/Z-19(h)/Z-100(z)
ESC y 1	Disable 25th line	hz
ESC y 2	Enable keyboard click	hz
ESC y 3	Exit hold screen mode	hz
ESC y 4	Set "underline" cursor	hz
ESC y 5	Enable cursor	hz
ESC y 6	Exit keypad shifted mode	hz
ESC y 7	Exit keypad alternate mode	hz
ESC y 8	Disable auto linefeed on carriage return	hz
ESC y 9	Disable auto carriage return on line feed	hz
ESC y ;	Set "blinking" cursor	z
ESC y <	Enable keyboard auto repeat	z
ESC y ?	Disable key expansion	z
ESC z	Reset to power-up configuration	hz
ESC {	Enable keyboard	hz
ESC ¦	Break key was pressed	z
ESC }	Disable keyboard	hz
ESC ~	HELP key	z

APPENDIX P

Notes on Writing Z-DOS Programs

How to Structure a Program so it Will Run Under Z-DOS (MS-DOS)

There are two types of executable files in Z-DOS (MS-DOS). These are .EXE files and .COM files. Below are the advantages and disadvantages of each:

.COM

Advantages

- It is at least 512 bytes shorter than .EXE file.
- It takes a little less time to load than .EXE file.

Disadvantages

- It must be less than 64K bytes long.
- It must be position independent code.
- It can not be produced by high level compiler.
- It must use the 8080 segment model.

.EXE

Advantages

- It can be up to 384K bytes long (you must also have enough physical memory).
- It can use any segment model.
- It can have position dependent code (which is patched when the program is loaded).
- It can use subroutines produced by high level compilers.

Disadvantages

- It is larger by at least 512 bytes (header) than .COM files.
- It can take longer to load than .COM files.
- It is harder to use some OS functions.

For both .EXE and .COM files, a program header is contructed when the program is loaded. The format of the program header is defined in the include file DEFMS.ASM. The conditions that exist for both .COM and .EXE files at program startup are as follows:

- The disk transfer address (DTA) is set to 80H in the progrm header (the default I/O area).

- The file control blocks (FCB's) at 5CH and 6CH in the program header are formatted (i.e., converted to upper case and blanked after from the first two parameters entered on the command line).

- The unformatted parameter area at 81H in the program header contains all the characters entered after the command name (including leading and embedded delimiters), with location 80H set to the number of characters.

- The memory size (first paragraph number after the end of memory) is stored at location 2H in the program header.

- The exit handler address, the CTRL-C handler address, and the fatal error handler address are stored in locations 0AH, 0EH, and 12H respectively in the program header.

APPENDIX P

Notes on Writing Z-DOS Programs

For .COM files, the registers on entry are:

AX, BX, CX, DX, BP, SI, DI are undefined
SP = 0FFFFH or constrained by end of memory
CS,DS,ES = Segment address of program header
IP = 100H
A word of zeros is pushed on the stack

For .EXE files, the registers on entry are:

AX, BX, CX, DX, BP, SI, DI are undefined

If a STACK segment is used then
  SS = Segment address of that segment
  SP = size of stack segment
ELSE
  SS = CS (see below)
  SP = 0FFFFH or constrained by end of memory
DS, ES = Segment address of program header
CS:IP = Far address of label in "END" statement of the program

The following programs are short examples of source code for both types of excutable files.

APPENDIX P

Notes on Writing Z-DOS Programs

```
        TITLE   EXAM1 - Example .EXE program
        PAGE    ,132

        .XLIST
INCLUDE DEFASCII.ASM
INCLUDE DEFMS.ASM
        .LIST

STKSEG  SEGMENT STACK
        DB      100H DUP(?)
STKSEG  ENDS

PGMSEG  SEGMENT
        ASSUME  CS:PGMSEG,SS:STKSEG,DS:DATASEG,ES:NOTHING
        DB      'EXAM1 - (C) Copyright 1982 by Zenith Data Systems'

START:
        MOV     AX,DATASEG      ; Set up DS
        MOV     DS,AX
        MOV     WORD PTR RTADDR+2,ES ; Save program header segment addr

        MOV     DX,OFFSET MESG  ; Get message address
        MOV     AH,DOSF_OUTSTR  ; Get print string function code
        INT     DOSI_FUNC       ; Print string

        JMP     RTADDR          ; Terminate program
PGMSEG  ENDS

DATASEG SEGMENT
RTADDR  DD      0               ; return addr(segment to be filled in)
MESG    DB      'EXAM1',CC_CR,CC_LF,'$'
DATASEG ENDS

        END     START
```

To create and run EXAM1.EXE, first create the above source code with EDLIN. Then enter the following commands:

A:**MASM EXAM1;** **RETURN**
A:**LINK EXAM1;** **RETURN**
A:**EXAM1** **RETURN**

```
        TITLE   EXAM2 - Example .COM Program
        PAGE    ,132

        .XLIST
INCLUDE DEFASCII.ASM
INCLUDE DEFMS.ASM
        .LIST

PGMSEG  SEGMENT
        ASSUME  CS:PGMSEG,SS:PGMSEG,DS:PGMSEG,ES:NOTHING
        ORG     100H            ; Position after program header

START:
        JMP     SHORT S1        ; Skip over copyright
        DB      'EXAM2 - (C) Copyright 1982 by Zenith Data Systems'

S1:
        MOV     DX,OFFSET MESG  ; Get addr of messge
        MOV     AH,DOSF_OUTSTR  ; Get function to output message
        INT     DOSI_FUNC       ; Print message

        INT     DOSI_TERM       ; Terminate program

MESG    DB      'EXAM2',CC_CR,CC_LF,'$'

PGMSEG  ENDS
        END     START
```

To create and run EXAM2.COM, first create the above source code with EDLIN. Then enter the following commands:

```
A: MASM EXAM2;      RETURN
A: LINK EXAM2;      RETURN
A: EXE2BIN EXAM2.EXE .COM      RETURN
A: ERASE EXAM2.EXE      RETURN
A: EXAM2      RETURN
```

APPENDIX Q

A Procedure to Change Disk Parameters

The purpose of this section is to show you how to change floppy disk parameters. The technique will be illustrated by showing you how to change the step rate for 8 inch floppy disks.

The location of the disk portion of the operating system is defined by a table that contains the "configuration vectors" or "pointers" (addresses) for the various system routines. You may locate this table and the disk subsystem in the two assembly language source files, DEFDSK.ASM and DEFZ207.ASM. These files are automatically included during the assembly of the Z-DOS BIOS and contain information specific to the Z-207 disk controller card. You may change the information in the table or disk subsystem temporarily (in memory) or permanently (by modifying the system disk).

Since the location (offset address of the table in the BIOS) may change with later releases of different versions of the BIOS, a "pointer" has been placed at a fixed location (address) as a permanent reference point. This location contains the offset address of the table, regardless where it may be located in this or any other release of the Z-DOS BIOS. The location of the pointer will not be changed in future releases of Z-DOS.

You can find the location of the fixed pointer when you assemble the file DEFMS.ASM. A portion of the assembled listing follows:

```
0061    BIOS_CTADDR   LABEL WORD          ; Addr of configuration information
0063          ORG OFFSET BIOS_CTADDR+2

0063  BIOS_SEG ENDS
```

The pointer's label is BIOS_CTADDR. As you can see, the location of the pointer (the offset addrsss) is 61 hexadecimal (Hex). By examining this location in the BIOS, you can find the location (address) of the Configuration Vector table.

APPENDIX Q

You can also find the Configuration Vector table in the file DEFMS.ASM. A representative portion of the assembled file follows:

```
                ;
                ; Configurationvector
                ;

=0000   CONFG_DSK     EQU 0              ; Addr of disk vector
=0002   CONFG_PRN     EQU CONFG_DKS+2    ; Addr of PRN configuration table
=0004   CONFG_AUX     EQU CONFG_PRN+2    ; Addr of AUX configuration table
```

Since you want to change the disk configuration, you will need to look for CONFG_DSK. It contains the address of the disk vector table. As you can see, the offset address during assembly is 00H. The actual address will be placed in this location during the "linking" of the various machine language modules as part of the assembly of the BIOS. However, you do know (from the information in the file) that the address you are looking for is the first one found in the table. If you were looking for the address of the PRN configuration table, it would be the second. The AUX would be the third, and so on.

You will find that the disk vector table contains the locations of the disk tables. Currently, the tables are arranged as follows:

1 5.25 inch floppy (drive A)
2 5.25 inch floppy (drive B)
3 8 inch floppy (drive C)
4 8 inch floppy (drive D)

APPENDIX Q

The following is a partial listing of the assembled file DEFDSK.ASM. The portion shown illustrates the format of a disk table. You may modify the fields flagged with an asterisk (*).

```
 = 0000   DSK_STA        EQU 0              ; Status of last operation
 = 0100     DSKST_ENERR    EQU   0100H       ; Invalid function
 = 0200     DSKST_ORERR    EQU   0200H       ; Improper order of function
 = 0300     DSKST_DNERR    EQU   0300H       ; Invalid disk number
 = 0400     DSKST_DTERR    EQU   0400H       ; Invalid disk type
 = 0500     DSKST_NIERR    EQU   0500H       ; Function not implemented
 = 0600     DSKST_NDERR    EQU   0600H       ; No disk in drive
 = 0002   DSK_TYPE       EQU DSK_STA+2      ; Disk type
 = 0000     DSK_TZ207      EQU 0             ; Z-207 type disk
 = 0003   DSK_LTRK       EQU DSK_TYPE+1     ; Last track
 = 0004   DSK_LOPT       EQU DSK_LTRK+1     ; Last operation
 = 0001     DSK_OWR        EQU   01H         ; Write was last op
 = 0002     DSK_ORD        EQU   02H         ; Read was last op
 = 0004     DSK_ORS        EQU   04H         ; Reset was last op
 = 0008     DSK_OSI        EQU   08H         ; Step in was last op
 = 0010     DSK_OFT        EQU   10H         ; Format was last op
 = 0020     DSK_ORT        EQU   20H         ; Read track was last op
 = 0080     DSK_OUK        EQU   80H         ; Track is unknown
*= 0005   DSK_FLAG       EQU DSK_LOPT+1     ; Flags
 = 0001     DSK_FDS        EQU   01H         ; Disk is double sided
+= 0002     DSK_FFS        EQU   02H         ; Drive can be fast stepped
 = 0004     DSK_FDP        EQU   04H         ; Disk is 48 tpi and should be double stepped
+= 0008     DSK_FWP        EQU   08H         ; Disk is software write protected
 = 0010     DSK_FDC        EQU   10H         ; Force Disk has Changed next time
+= 0020     DSK_FSL        EQU   20H         ; Skip head load on select
+= 0040     DSK_FRS        EQU   40H         ; Restore fast, then slow
 = 0006   DSK_SEL        EQU DSK_FLAG+1     ; Command to select drive
*= 0007   DSK_RS         EQU DSK_SEL+1      ; Command to reset drive
*= 0008   DSK_SPHI       EQU DSK_RS+1       ; Command to step in
 = 0009   DSK_FMT        EQU DSK_SPHI+1     ; Command to format (write) a track
 = 000A   DSK_RD         EQU DSK_FMT+1      ; Command to read a sector
 = 000B   DSK_WR         EQU DSK_RD+1       ; Command to write a sector
```

APPENDIX Q

```
*= 000C   DSK_SK          EQU DSK_WR+1       ; Command to seek to a track
 = 000D   DSK_SERR        EQU DSK_SK+1       ; Number of "soft" errors
 = 000F   DSK_MAXT        EQU DSK_SERR+2     ; Maximum track number of drive
*= 0010   DSK_NRETRY       EQU DSK_MAXT+1     ; Maximum retry count
 = 0011   DSK_SPT         EQU DSK_NRETRY+1   ; Sectors per track
 = 0012   DSK_BPS         EQU DSK_SPT+1      ; Number of bytes per sector
 = 0014   DSK_BPWT        EQU DSK_BPS+2      ; Number of bytes per write track operation
 = 0016   DSK_BPRT        EQU DSK_BPWT+2     ; Number of bytes per read track operation
*= 0018   DSK_DELAY       EQU DSK_BPRT+2     ; Counter value for short delay
*= 001A   DSK_LDELAY      EQU DSK_DELAY+2    ; Counter value for a long delay
 = 001C   DSK_PORT        EQU DSK_LDELAY+2   ; Base Port number
 = 001E   DSK_RDT         EQU DSK_PORT+2     ; Read track command
 =001F    DSK_IMGFLG       EQU DSK_RDT+1      ; Imaginary drive flag
 = 0080     DSKIF_ID         EQU 80H            ; (0 - real drive; 1 - imaginary drive)
 = 0040     DSKIF_DV         EQU 40H            ; (0 - disk is not in drive; 1 - disk is in drive)
 = 0020     DSKIF_NM         EQU 20H            ; (0 - can map imag to drive; 1 - can't)
```

APPENDIX Q

Changing Step Rates for the 8 Inch Disk Drive

The (step) rates for the step-, restore-, and seek-operations are embedded in commands sent to the 1797 disk controller chip. These commands are considered "type 1" commands.

The step rate for 8 inch disk drives is one half of the value given in the file, DEFZ207.ASM. Also, two other characteristics (established by two flags if the DSK_FLAG field) may need attention when you change the step rates: DSK_FSL and DSK_FRS.

DSK_FSL either causes a delay which allows the disk head(s) to load (when set) or selects no head delay (when not set). The head(s) in some disk drives is (are) loaded whenever the drive door is shut and therefore need no delay to allow the head to load when that drive is selected.

DSK_FRS (when set) affects the rate that the head will be restored to the position over track zero. If a high step rate is used during the restore operation, the head may easily overshoot when it reaches track zero. To overcome this problem, this flag (when set) will reduce the step rate as the head approaches track zero. If the drives are set at slower step rates, the problem does not exist; the flag (when not set) will not affect the restore operation.

The following procedure may be used to permanently change the step rate of both 8 inch disk drives. The general procedure may be used to change other factors as well.

Any time you want to change some factor in the BIOS, it is best to start with a "clean" Z-DOS disk. To do so, you should first format and place the operating system on a "new" disk (the disk may have been previously used for something else; the format procedure, in effect, creates the "new" disk). The next step involves the actual "patching" of the BIOS. The third step is up to you; we suggest that you thoroughly test the newly created BIOS to make sure that it will perform as you want it to do. The final step is to place the new BIOS on any other disks. This can be done by using the SYS command of Z-DOS as explained elsewhere in this manual.

APPENDIX Q

☐ Create a new 5.25 inch system disk by typing:

```
FORMAT A:/S
```

and pressing the **RETURN** key. You will be prompted to place a blank disk in drive A and to press the RETURN key when ready. Follow the prompts. When the new disk has been formatted, replace it in drive A with your Z-DOS disk.

☐ Start DEBUG. Type:

```
DEBUG
```

and press the **RETURN** key. Now that DEBUG is in the system, you may replace the Z-DOS disk in drive A with your the disk that you will modify.

Now you are ready to start modifying the BIOS. In order to do so, you will first have to read a portion of the disk into memory where it can be modified. Then, after you have modified that portion, you will write it back out to the disk.

You can find the location of the starting data sector in the table "Z-DOS Disk Structures" on Page H.1 of Appendix H. For double-sided, 5.25 inch disks, the starting data sector is 10 (0AH).

Also, since the length of the BIOS can vary from release to release, you will want to read enough of the disk to make sure that you have all of the BIOS in memory. 32 (20H) 512-byte sectors should be sufficient (16K).

☐ Load twenty sectors into memory. Type:

```
L 1000:0 0 A 20
```

and press the **RETURN** key. Here is what the command means:

L — Load command (DEBUG)
1000:0 — starting memory location to place the BIOS
0 — hardware drive number (0 = A, 1 = B, 2 = C, 3 = D)
A — starting sector number to be read (in hexadecimal)
20 — the number of sectors to be read (in hexadecimal)

APPENDIX Q

When you pressed the RETURN key, the LED disk access indicator lit and the disk was read. You may now examine and change any part of the BIOS. But first, you must locate the part you want to change.

☐ Examine location 61H to find the address of the configuration pointers. Type:

```
D 1000:61 62
```

and press the **RETURN** key. Here is what the command means:

D — Dump command (DEBUG display command)
1000:61 62 — address range to "dump" (display)

The computer will display something like this:

```
1000:0061  7A 04
```

The least significant portion of the address is stored in address 61H. The most significant is found in 62H. The address you want is therefore (in this example) 047AH.

Now you want the location of the disk vector table. Its location is stored as an address in the two bytes, starting at the address you just obtained (047AH, in our example).

☐ Type:

```
D 1000:047A 047B
```

and press the **RETURN** key. The computer will display something like this:

```
1000:047A  D0 22
```

This is the beginning address of the disk vector table. You want the third address; drive C is the third entry in the table. Since each address takes two bytes, you will want to get the fourth and fifth digits.

APPENDIX Q

☐ Type:

```
D 1000:22D4 22D5
```

and press the **RETURN** key. The computer will display something like this:

```
1000:22D4  56 23
```

Now you have the address of the table for the first 8 inch disk drive. To change the step rate from 3 ms. to 15 ms., you will want to modify the DSK_FRS flag. The flag is offset 5 bytes in the field (235BH). To change the flag, use the DEBUG E (Enter) command.

☐ Type:

```
E 1000:235B
```

and press the **RETURN** key. The computer will display something like this:

```
1000:235B  51.
```

You will note that the cursor remained in position after the period, rather than going to the next line and displaying the DEBUG prompt. The computer is waiting for your entry. But first examine the DSK_FLAG section of DEFZ207.ASM. You will see that there are seven flags which are added together to form DSK_FLAG. DSK_FRS represents 40H when set. To turn off DSK_FRS, you will have to subtract 40H from the value in 235BH. That leaves you with 11H.

☐ Type:

```
11
```

and press the **RETURN** key. Next, you will need to modify the restore command (DSK_RS – offset 7 bytes: 235DH), the step command (DSK_SPHI – offset 8 bytes: 235EH), and the seek command (DSK_SK – offset 12 bytes: 2362H).

APPENDIX Q

☐ Use the Enter command and type in the appropriate amounts as follows:

```
E 1000:235D
1000:235D  08.B
E 1000:235E
1000:235E  58.5B
E 1000:2362
1000:2362  1C.1F
```

When you have completed all of the previous steps, you will have completed the modification for drive C. The modification for drive D will be done in a similar manner. That is, locate the disk table address in the disk vector table (at location 22D6H and 22D7H in our example); and then modify the restore flag, the restore command, the step command, and the seek command. Finally, prepare to write the BIOS back to the disk.

To write the BIOS to the disk, you will simply perform the reverse of the process that you used to read the BIOS into memory.

☐ Type:

```
W 1000:0 0 A 20
```

and press the **RETURN** key. The only difference between this and the read command is the W in place of the R; and the W is the DEBUG command to write.

To use the new BIOS, reset the computer and boot the new disk. Other portions of the BIOS may be similarly modified. The source files for your version of BIOS are contained on the Z-DOS Distribution Disk II. The BIOS used for the examples shown was version 1.00.

ZENITH | data systems ST. JOSEPH, MICHIGAN 49085

Dear Customer,

Thank you for purchasing Z-DOS, an extremely flexible and powerful Disk Operating System for the Z-100 Series of Desktop Computers. Since the original release of Z-DOS, many customers have asked for additional information on modifying the BIOS (Basic Input Output System) module of the operating system, particularly with regard to adjusting the step rate for 8 inch disk drives. We are therefore adding Appendix Q, Modifying the Z-DOS BIOS, which you should add to the back of Volume II of Z-DOS. You will also need to make a notation to the Table of Contents of both Volumes to reflect the additional material.

Thank you,

Zenith Data Systems

HEATH | ZENITH data systems

Memory Checking

Modern personal computers are extremely reliable. Your Z-100 operates at a speed of 5 million clock cycles per second and will typically operate for several thousand hours between service calls. However, computers are machines, and machines do, on occasion, develop problems. Among the problems which may occur in all computers are memory failures.

Memory failures may be classified as either "hard" or "soft". Soft memory errors occur randomly at an average frequency of roughly once every several thousand hours of operation and do not indicate the presence of a hardware problem. Hard memory failures are due to defective components within the computer, and will recur frequently until the defective component is replaced.

Your Z-100 computer has special circuitry and additional memory to automatically detect any memory failures (hard or soft) which do occur, through a technique known as "parity checking". When a "parity error" occurs, a display similar to the one below will appear on the screen:

```
ERROR - MEMORY OR BUSS

F =XXXX   IP=XXXX   CS=XXXX   DS=XXXX   ES=XXXX   SS=XXXX   SP=XXXX
AX=XXXX   BX=XXXX   CX=XXXX   DX=XXXX   DI=XXXX   SI=XXXX   BP=XXXX

SYSTEM HALT
```

The purpose of providing the parity error message is to let you know that you may have a problem which may not be readily apparent and which may require the attention of service personnel. Parity checking is an advanced feature found in only a few microcomputer systems. In systems without parity checking, the memory error usually goes unnoticed for a period of days or weeks until the amount of data destroyed becomes so large that it can no longer be ignored.

When an error occurs, the system will display the parity error message described above and halt. The system must then be reset and rebooted, with the consequence that all work in the computer's memory and any unclosed files on the disk will be permanently lost. It is generally best not to use the system any further until the memory test (supplied with your Z-DOS operating system) has been run (preferably from a write-protected disk!). If the memory test does not turn up problems after several hours of operation, you should resume normal operation. If subsequent memory failures occur, you should copy down all of the data presented on the screen by the parity error routine and retain it for use by service personnel. Also, record which program and operating system you were using at the time. If multiple errors occur within 30 days, a serviceman should be called, even if the memory test does not indicate the presence of problems.

INDEX

Z-DOS Index

INDEX

Z–DOS Index

INDEX

Z-DOS Index

INDEX

Z-DOS Index

INDEX

Z–DOS Index

INDEX

Z–DOS Index

INDEX

Z–DOS Index

Resident debugger, **7.4**
Run file, 11.2
<runfile>, 11.14

S

/S switch, 6.70, **6.75, 9.9**
S command, **7.32, 8.32**
.SALL, **10.162**
SEG, **10.64, 10.65**
Segment, **10.65, 10.117**
  definition of, **11.3**
  override, 10.16, 10.58
  Override (:) Colon, **10.56**
  registers, 10.49
<segment-name>, 10.60
<segment-register>, 10.60
semicolon (;), 11.13, 12.18, 13.5
Serial device, 6.32
Shift-count, **10.73**
SHORT, **10.61**
sign flag, 10.50
SIZE, **10.71**
Skip over, 4.15
Software development, 6.44, 7.3
Source code, 10.1
Source file, definition of, 10.18
Source operand, 10.24
SPANISH.CHR, descritpion of, 6.9
Special characters
  as delimiters, 10.16
  as operators, 10.16
Special libraries, 12.1
Special macro operators, 10.148
<ss>, 6.113
Stack pointer, 11.5
/STACK:<number> Switch, **11.12, 11.20**
Status report, directory, 6.20
Stop bits, 6.33
Storage, **1.4**
<string>
  DEBUG usage, 7.9
  EDLIN usage, 8.7
STRUC, **10.131**
SUBTTL, **10.166**
Supported drive names, **3.11**
SWEDISH.CHR, descritpion of, 6.9
Switch SW-101, 5.3
Switching default drives, 3.13
Symbolic names, 13.1
Syntax error, 6.45
SYS, 6.6, **6.112**
SYS.COM, description of, 6.10
SYSCOPY.DAT, **6.93**
SYSCOPY.DAT, description of, 6.10
%SYSTEM, **6.91**
System, **2.3**
  CRT, 6.30
  Prompt, 3.12, 4.1
  Memory, definition of, **2.2**
  Resident, 6.4
  Resident, commands, 4.1, **4.2**, 4.7
  Resources, **2.4, 2.8**

T

T command, **7.33**
Table of commands, 6.4
TBYTE, 10.67
Template, **4.8**, 4.17
.TFCOND, **10.162**