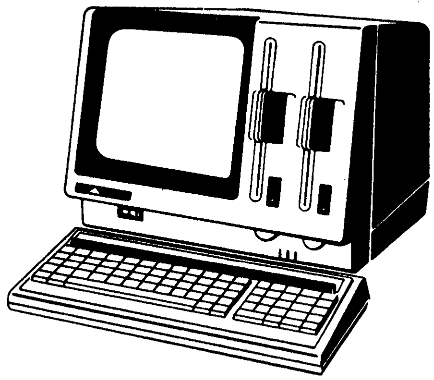
Advanced Personal Computer ™



MS™-DOS System Programmer's Guide

NEC

NEC Information Systems, Inc.

819-000104-3001 REV 00
9-83

Important Notice

PLEASE READ THE FOLLOWING TEXT CAREFULLY. IT CONSTITUTES A CONTINUATION OF THE PROGRAM LICENSE AGREEMENT FOR THE SOFTWARE APPLICATION PROGRAM CONTAINED IN THIS PACKAGE.

If you agree to all the terms and conditions contained in both parts of the Program License Agreement, please fill out the detachable postcard and return it to:

NEC Information Systems, Inc.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

LIABILITY

In no event shall the copyright holder, the original licensor nor any intermediate sublicensors of this software be responsible for any indirect or consequential damages or lost profits arising from the use of this software.

COPYRIGHT

The name of the copyright holder of this software must be recorded exactly as it appears on the label of the original diskette as supplied by NECIS on a label attached to each additional copy you make.

You must maintain a record of the number and location of each copy of this program.

All NECIS software programs and copies remain the property of the copyright holder, though the physical medium on which they exist is the property of the licensee.

MERGING, ALTERATION

Should this program be merged with or incorporated into another program, or altered in any way by the licensee, the terms of the Warranty contained herein are voided and neither NECIS nor the copyright holder nor any intermediate sublicensors will assure the conformity of this software to its specification nor refund the license fee for such nonconformity.

Upon termination of this license for any reason, any such merged or incorporated programs must be separated from the programs with which they have been merged or incorporated and any altered programs must be destroyed.

819-000102-8D01

Advanced Personal Computer™

NEC
NEC Information Systems, Inc.

Program Name (as it appears on diskette label)

____

Serial Number ______________________ Date Purchased ____________

Dealer Name and City ______________________________________

Your Name ____

Your Address ____

City ______________________ State ________ ZIP __________

"Warranty Requires Return of This Card"

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 386 LEXINGTON MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

Contents

Contents (cont'd)

Contents (cont'd)

Contents (cont'd)

Contents (cont'd)

Contents (cont'd)

Tables

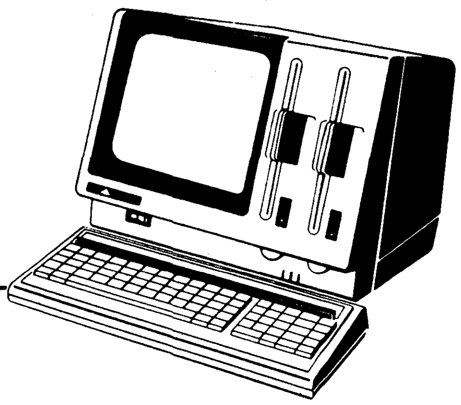
Illustrations

Preface

The *MS*TM*-DOS System Programmer's Guide* presents the system programming aspects of the MSTM-DOS operating system for the Intel 8086 and 8088 microprocessors, modified for the APC. This document assumes you are familiar with MS-DOS as it operates on the APC. If not, refer to the *MS*TM*-DOS System Reference Guide*. In addition to MS-DOS system characteristics, the reference guide describes utilities used in program development. Another reference is the *MS*TM*-DOS System User's Guide*, which contains descriptions of the MS-DOS file organization and commands that execute basic system routines and utilities.

Chapter 1

What Is In This Guide?



This guide presents several system utilities and programs available for program development on the APC running with the MS™-DOS disk operating system.

The assembler used with MS-DOS is Microsoft's MACRO-86 Macro Assembler™ for 8086-based computers. First, the elements of the assembly language subset utilized to code source files are described. Then, you are instructed on how to execute MACRO-86.

Designed to work with MACRO-86 are the MS-LINK Linker Utility™, MS-LIB Library Manager™, and MS-CREF Cross Reference Utility™. MS-LINK is a virtual linker that produces relocatable 8086 object modules. MS-LIB is a versatile utility for creating and maintaining program library files. MS-CREF converts a file optionally generated by MACRO-86 into a listing of source code symbols.

The DEBUG Debugging Program™ and the FC File Comparison Utility™ aid in program testing. DEBUG examines binary and executable object files for errors and corrects them. FC compares the contents of source and object files with like files to find dissimilarities. Files operated on by each of these programs can be in any high-level or Intel 8086-compatible assembly language usable in the MS-DOS system.

Two programs described in this guide direct you on how to access specific APC facilities from assembly language programs. The Auxiliary Character Generator displays alternate character sets on the APC screen from the random access memory reserved for this purpose. The Soft Key Definition Program loads into memory a soft key table containing the codes assigned to the APC's 16 dual-mode function keys.

Information is also included for programmers who want to install their own device drivers under MS-DOS. Chapter 11 tells you how to write character and block device drivers to perform input and output to peripheral devices.

DEVELOPING ASSEMBLY LANGUAGE PROGRAMS

A series of steps is typically followed in developing an assembly language program.

1. First, you use the EDLIN Line Editor™ or other 8086 editor compatible with your operating system, to create an 8086 assembly language source file. Give the source file the filename extension .ASM. (MACRO-86 recognizes .ASM as the default.)
2. Next, you assemble the source file with MACRO-86 which outputs an assembled object file with the default filename extension .OBJ. Assembled files, your program files, can be linked together in step 3.

   MACRO-86 optionally creates two types of listing file:

   - a normal listing file that shows assembled code with relative addresses, source statements, and full symbol table
   - a cross-reference file that MS-CREF can operate on to create a list showing the source line number of every symbol's definition and all references to it. When a cross reference file is created, the normal listing file (with the .LST filename extension) has line numbers placed into it as reference line numbers following the symbols in the cross reference listing.
3. Then, you link one or more .OBJ modules together, using MS-LINK, to produce an executable object file with the default filename extension .EXE.

   While developing your program, you may want to create a library file for MS-LINK to search to resolve external references. Use MS-LIB to create user library files from existing library files and/or user program object files.
4. Finally, you run your assembled and linked program, the .EXE file, under MS-DOS.

Figure 1-1 is a flow diagram of the program development process.

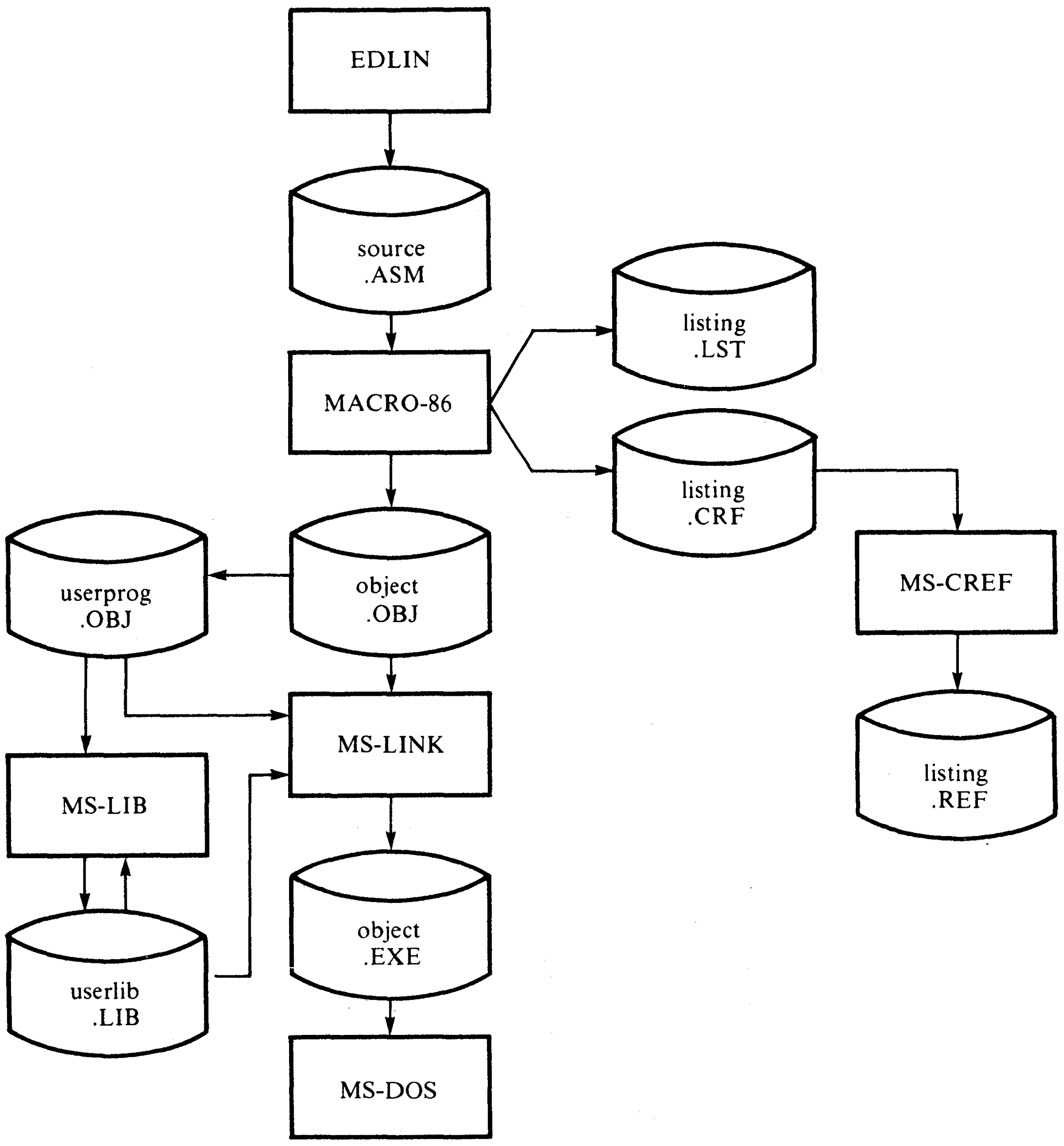


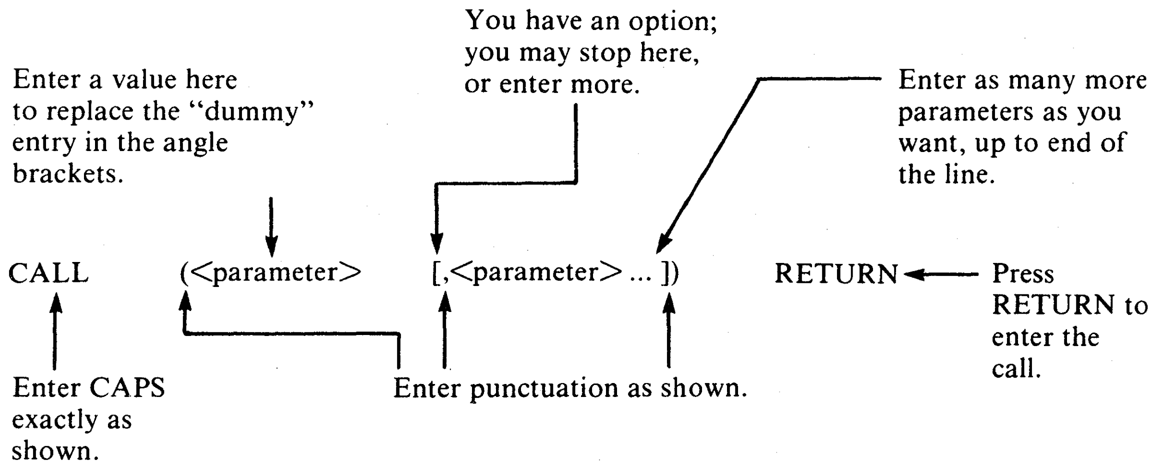
Figure 1-1 Program Development Flow Diagram

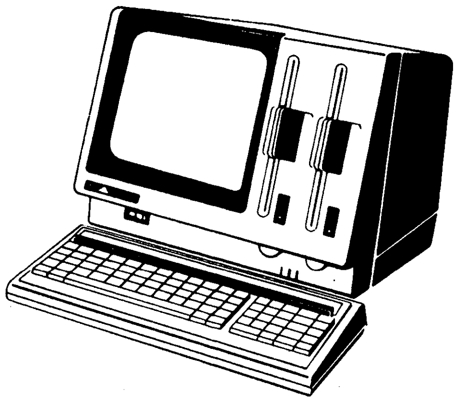
SYNTAX NOTATION

Certain symbols are used in this guide to describe the syntax of commands and statements entered to MS-DOS from the utilities and programs presented.

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. The angle brackets enclose lower case text that represents the entry to be made.
{ }	Braces indicate that you have a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered. Also, a key that must be pressed at the end of a command line or statement is indicated by its name in caps.

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


You have an option; you may stop here, or enter more.
Enter a value here to replace the "dummy" entry in the angle brackets.
Enter as many more parameters as you want, up to end of the line.
CALL (<parameter> [,<parameter> ...]) RETURN
Press RETURN to enter the call.
Enter CAPS exactly as shown.
Enter punctuation as shown.



Chapter 2
8086 Assembly Language Elements

The 8086 assembly language is used with the MACRO-86 Macro Assembler. The Line Editor (EDLIN) is the program that creates the source files for MACRO-86. These files are simply program files of source code statements. (If you are not familiar with EDLIN, refer to the *MS-DOS System User's Guide* for a description of this program.)

GENERAL FACTS ABOUT SOURCE FILES

Naming Your Source File

When you create a source file, you will need to name it. Like the other files in the MS-DOS operating system, a filename (symbol name) for an assembly language source file may consist of up to 8 legal characters. (Refer to the *MS-DOS System User's Guide* for a description of filenames.) MACRO-86 expects a specific three-character filename extension, .ASM. That is, whenever you run MACRO-86 to assemble your source file, MACRO-86 assumes that your source filename has this filename extension. However, you may name your source file with any extension you like. When you run MACRO-86, though, you must remember to specify the extension. Because of this action by MACRO-86, it is impossible to omit the filename extension. If you assemble a source file without a filename extension, MACRO-86 will assume that the extension is .ASM. MACRO-86 will then search the disk for the file, and not finding the correct file, will either assemble the wrong file or will return an error message stating that the file cannot be found.

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

What Is in a Source File?

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

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. Generally speaking, statements may appear in any order within the segments that creates a valid program. Some exceptions to random ordering do exist, which will be discussed under the affected assembler directives.

Every segment must end with an end segment statement (ENDS), every procedure must end with an end procedure statement (ENDP), and every structure must end with an end structure statement (ENDS). Likewise, the source file must end with an END statement, which contains an optional label, to tell MACRO-86 where program execution begins.

The section **Memory Organization** describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another. This information is important for developing your programs. It is presented as a prelude to the discussion of operands and operators.

Numeric Notation

The default input radix for all numeric values is decimal. The output radix for all listings is hexadecimal for code and data items, and decimal for line numbers. For more information, refer to the section Data Items.

Statement Line Format

Statements in source files follow a format that 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, 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 following discussion of names.

NAME FIELD

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 here apply to macro names, the discussion of macro names is better left to the section MACRO DIRECTIVES.

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 only inside its own segment)
- NAME LABEL FAR (for use outside its own segment)
- EXTRN NAME:NEAR (for use only outside its own module but inside its own segment)
- EXTRN NAME:FAR (for use outside its own module and segment.

To make a name represent data, use

- NAME LABEL <size> (BYTE, WORD, and so on)
- NAME Dx <exp>
- EXTRN NAME: <size> (BYTE, WORD, and so on).

To make a name represent a constant use

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

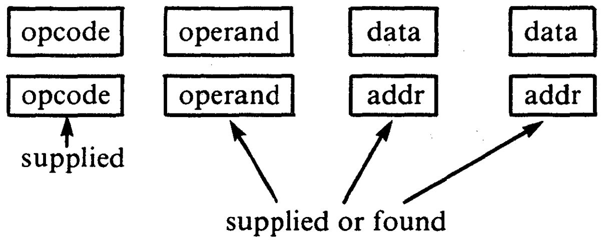
ACTION FIELD

The action field contains either an 8086 instruction mnemonic or a MACRO-86 assembler directive. Refer to the section **Instructions** for more information, including the list of 8086 instruction mnemonics. The MACRO-86 directives are described in detail in the section **Directives**.

If the name field is blank, the action field will be the first entry in the statement format. In this case, the action may appear starting in any column, 1 through the maximum line length (less columns for action and expression).

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

Example:



where: *supplied* is part of the instruction

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

opcode = the action part of an instruction.

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

EXPRESSION FIELD

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

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

```
opcode      dest-operand,      source-operand
```

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

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

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

```
directive      operand
```

A directive operand is a data operand, 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.

COMMENT FIELD

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

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

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

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

Table 2-1 lists the operands and operators MACRO-86 supports in the expression field. Operands and operators are shown in order of precedence.

Table 2-1 MACRO-86 Operands & Operators

OPERANDS	OPERATORS
Immediate (including symbols)	LENGTH, SIZE, WIDTH, MASK, FIELD [ ], ( ),
Register	segment override (:)
Memory	
label	
variables	PTR, OFFSET, SEG, TYPE, THIS,
simple	
indexed	HIGH, LOW
structures	

Table 2-1 MACRO-86 Operands & Operators (cont'd)

OPERANDS	OPERATORS
Attribute	*, /, MOD, SHL, SHR
override	
PTR	+, -(unary), -(binary)
:(seg)	
SHORT	EQ, NE, LT, LE, GT, GE
HIGH	
LOW NOT	
value returning	
OFFSET	AND
SEG	
THIS	OR, XOR
TYPE	
.TYPE	SHORT,.TYPE
LENGTH	
SIZE	
record specifying	
FIELD	
MASK	
WIDTH	

NOTE

Some operators can be used as operands or as part of an operand expression. Refer to the sections **Operands** and **Operators** for details.

NAMES: LABELS, VARIABLES, AND SYMBOLS

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

Names are symbolic representations of values. These 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.

MACRO-86 supports three types of names in statement lines: labels, variables, and symbols.

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:

- <name>:

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

  Examples:

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

- <name> LABEL NEAR
  <name> LABEL FAR

  Use the LABEL directive. Refer to the discussion of the LABEL directive in the section MEMORY DIRECTIVES.

  NEAR and FAR are discussed below under the type attribute.

  Examples:

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

- <name> PROC NEAR
  <name> PROC FAR

  Use the PROC directive. Refer to the discussion of the PROC directive in the section MEMORY DIRECTIVES.

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

  Examples:

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

- EXTRN <name>:NEAR
  EXTRN <name>:FAR

  Use the EXTRN directive.

  NEAR and FAR are discussed below under the type attribute.

  Refer to the discussion of the EXTRN directive in the section MEMORY DIRECTIVES.

  Example:

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

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

SEGMENT ATTRIBUTE

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. Therefore, the segment (or group) attribute of a symbol is the base address of the segment (or group) where it is defined.

OFFSET ATTRIBUTE

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 ATTRIBUTE

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

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

NEAR and FAR cause MACRO-86 to generate slightly different code. NEAR labels supply their offset attribute only (a 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 an address where a specified value may be found.

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

Variables are defined three ways:

-
  ```
  <name> <define-dir>                     ;no colon!
  <name> <struc-name> <expression>
  <name> <rec-name> <expression>
  ```

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

  Example:

  ```
  START_MOVE    DW     ?
  ```

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

  <rec-name> is a record name defined by the RECORD directive.

  Examples:

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

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

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

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

  See the DEFINE, STRUC, and RECORD directives in the section MEMORY DIRECTIVES.

- <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 the LABEL directive in the section MEMORY DIRECTIVES.

- EXTRN <name> : <size>

  Use the EXTRN directive with one of the size specifiers described above. See the EXTRN directive in the section MEMORY DIRECTIVES.

  Example:

  ```
  EXTRN   FOO:DWORD
  ```

As do labels, variables have three attributes: segment, offset, and type.

Segment and offset are the same for variables as for labels.

The type attribute is different. This 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 used in expressions as operands to instructions and directives.

Symbols are defined three ways:

- <name> EQU <expression>

  Use the EQU directive. See the EQU directive in the section MEMORY DIRECTIVES.

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

Examples:

```
FPP   EQI   7H
ZOO   EQU   FOO
```

- <name> = <expression>

  Use the Equal Sign directive (see the section MEMORY DIRECTIVES).

  <expression> may be any valid expression.

  Examples:

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

- EXTRN <name> : ABS

  Use the EXTRN directive with type ABSD (see the section MEMORY DIRECTIVES).

  Example:

  ```
  EXTRN   BAZ:ABS
  ```

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

Legal Characters for Symbol Names

The legal characters for symbol names in source files are

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

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

Additional special characters act as operators or delimiters.

:	(colon) segment override operator
.	(period) operator for field name of a record or structure. May be used in a filename only if it is the first character.

[ ]	(square brackets) around register names to indicate value in address in register not value (data) in register.
( )	(parentheses) operator in DUP expressions and operator to change precedence of operator evaluation.
< >	(angle brackets) operators used around initialization values for records or a structure, around parameters in IRP macro blocks, and to indicate literals.

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

EXPRESSIONS: OPERANDS AND OPERATORS

Basically, "expression" is the term used to indicate values on which an instruction or directive performs its functions.

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

The discussion of memory organization for a MACRO-86 program acts as a preface to the descriptions of operands and operators and as a link to names discussed in the preceding section.

Memory Organization

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

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

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

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

You have the option of placing 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 makes values relative to a group base. The value of grouping is that you can refer to data items without worrying about segment overrides and about changing segment registers often.

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

There are two kinds of references in a program:

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

- Segment — What segment the label belongs to
- Offset — The number of bytes from the beginning of its segment
- Type — NEAR or FAR
- CS ASSUME — The CS ASSUME that the label was coded under

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

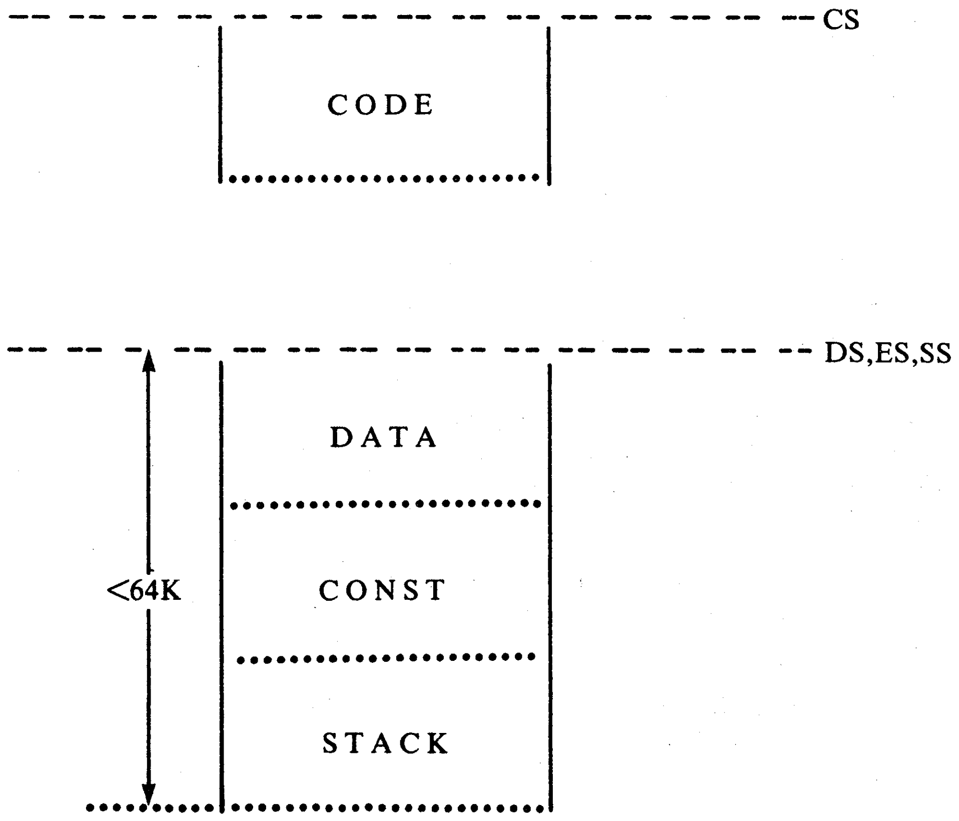
When you enter a FAR JMP or FAR CALL, you are changing both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

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

The program statements would be

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

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



Given this arrangement, a statement like

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

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

A statement like

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

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

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

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

MACRO-86 is a two-pass assembler. During pass 1, it builds a symbol table and calculates how much code is generated but does not produce object code. If undefined items are found (including forward references), assumptions are made about the reference so that the correct number of bytes are generated on pass 1. Only certain types of errors are displayed: errors involving items that must be defined on pass 1. No listing is produced unless you give a /D switch when you run the assembler. The /D switch produces a listing for both passes.

On pass 2, the assembler uses the values defined in pass 1 to generate the object code. Definitions of reference during pass 2 are checked against the pass 1 value, which is in the symbol table. Also, the amount of code generated during pass 1 must match the amount of code generated during pass 2. If either is different, MACRO-86 returns a phase error.

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

The following references must be known on pass 1:

- IF/IFE <expression>

  If expression is not known on pass 1, MACRO-86 does not know how to assemble the conditional block or which part to assemble if ELSE is used.

On pass 2, the assembler would know and would assemble, resulting in a phase error.

- <expression> DUP (...)

  This operand explicitly changes the relative offset, so <expression> must be known on pass 1. The value in parentheses need not be known because it does not affect the number of bytes generated.

- .RADIX <expression>

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

The biggest problem for the assembler is handling forward references. How can it know the kind of a reference when it still has not seen the definition? This is one of the main reasons for two passes. And, unless MACRO-86 can tell from the statement containing the forward reference what the size, the distance, or any other of its attributes are, the assembler can only take the safe route and generate the largest possible instruction in some cases (except for segment override or FAR). This results in extra code that does nothing. MACRO-86 figures this out by pass 2, but it cannot reduce the size of the instructions without causing an error, so it puts out NOP instructions (90H).

For this reason, MACRO-86 includes a number of operators to help the assembler. These operators tell MACRO-86 what size instruction to generate when it is faced with an ambiguous choice. As a benefit, you can also reduce the size of your program by using these operators to change the nature of the arguments to the instructions.

Examples:

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

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

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

This instruction 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 fewer 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 to save yourself a byte when using forward references. If you defined FOO as a forward constant, you might enter the statement:

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

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

```
MOV BYTE PTR [BX],FOO

MOV [BX], BYTE PTR FOO
```

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

Operands

An operand may be any one of three types: immediate, register, or memory operands. There is no restriction on combining the various types of operands.

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

- Immediate operands
  - Data items
  - Symbols

Register operands

Memory operands
- Direct
  - Labels
  - Variables
  - Offset (fieldname)

- Indexed
  - Base register
  - Index register
  - [constant]
  - ±displacement

- Structure

IMMEDIATE OPERANDS

Immediate operands are constant values that you supply when you 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 AS,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. Table 2-2 summarizes the numeric data forms.

Table 2-2 Numeric Data Forms

DATE FORM	FORMAT	EXAMPLE
Binary	xxxxxxxxB	01110001B
Octal	xxxO xxxQ	7350 (letter O) 412Q

Table 2-2 Numeric Data Forms (cont'd)

DATE FORM	FORMAT	EXAMPLE
Decimal	xxxxx xxxxxD	65535 (default) 1000D (when .RADIX changes input radix to nondecimal)
Hexadecimal	xxxxH	0FFFFH (first digit must be 0-9)
ASCII	'xx' "xx"	'OM'-more than two with DB only; "OM"-both forms are synonymous
10 real	xx.xxE±xx	25.23E-7 (floating point format)
16 real	x...xR	8F76DEA9R (first digit must be 0-9; the total number of digits must be 8, 16, or 20; or 9, 17, 21 if first digit is 0)

The output radix for all listings is hexadecimal for code and data items, and decimal for line numbers. The output radix can only be changed to octal radix by setting the /O switch when MACRO-86 is run (see the section Command Switches).

The input radix may be changed two ways:

- The .RADIX directive (see the section **Memory Directives)**
- Special notation characters can be appended to a numeric value. Table 2-3 designates these special notation characters.

Table 2-3 Notation for Changing the Input Radix

RADIX	RANGE	NOTATION	EXAMPLE
Binary	0-1	B	01110100B
Octal	0-7	Q or O (letter)	735Q 6210

Table 2-3 Notation for Changing the Input Radix (cont'd)

RADIX	RANGE	NOTATION	EXAMPLE
Decimal	0-9	(none) or D	9384 (default) 8149D (when .RADIX directive changes default radix to not decimal)
Hexadecimal	0-9,	H	OFFH 80H (first character must be numeral in range 0-9)

Symbols

Symbols names equated with some form of constant information (see the previous section **Symbols**) may be used as immediate operands. Using a symbol constant in a statement is the same as using a numeric constant. Therefore, using the sample statement above, you could enter

```
MOV AX,FOO
```

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

```
FOO EQU 9
```

REGISTER OPERANDS

The 8086 processor contains a number of registers. These registers are identified by two-letter symbols (reserved) that the processor recognizes.

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

Information about some of these registers is as follows:

- The general registers are two sizes: 8-bit and 16-bit. All other registers are 16-bit.
- The general registers are both 8-bit and 16-bit registers. Actually, the 16-bit general registers are composed of a pair of 8-bit registers, one for the low

byte (bits 0-7) and one for the high byte (bits 8-15). Note, however, that each 8-bit general register can be used independently from its mate. In this case, each 8-bit register contains bits 0-7.

- Segment registers are initialized by the user and contain segment base values. The segment register names (CS, DS, SS, ES) can be used with the colon segment override operator to inform MACRO-86 that an operand is in a different segment than specified in an ASSUME statement. (See the segment override operator in the section ATTRIBUTE OPERATORS.)
- The flag register is one 16-bit register containing nine 1-bit flags (six arithmetic flags and three control flags).

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

Register/Memory Field Encoding:

MOD=11 (Register Mode)

R/M	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

NOTE

The R/M bits are found within the machine instruction.

Effective Address Calculation:

R/M	MOD=00	MOD=-1	MOD-10
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16

011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16
100	[SI]	[SI]+D8	[SI]+D16
101	[DI]	[DI]+D8	[DI]+D16
110	DIRECT ADDRESS	[BP]+D8	[BP]+D16
111	[BX]	[BX]+D8	[BX]+D16

NOTE

D8 = a byte value; D16 = a word value.

Other Registers:

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

Flags:

6 1-bit arithmetic flags		3 1-bit control flags	
CF	carry flag	DF	direction flag
PF	parity flag	IF	interrupt-enable flag
AF	auxiliary flag	TF	trap flag
AF	zero flag		
SF	sign flag		

Note that 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 the section **Memory Operands**).

MEMORY OPERANDS

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

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

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

Direct Memory Operands

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

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

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

Indexed Memory Operands

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

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

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

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

The types of indexed memory operands are as follows:

Base registers	[BX] and [BP]. BP has SS as its default segment register; all others have DS as the default.
Index registers	[DI] and [SI]
[constant]	An immediate in square brackets. For example, [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 are

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

More examples of equivalent forms are

```
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.

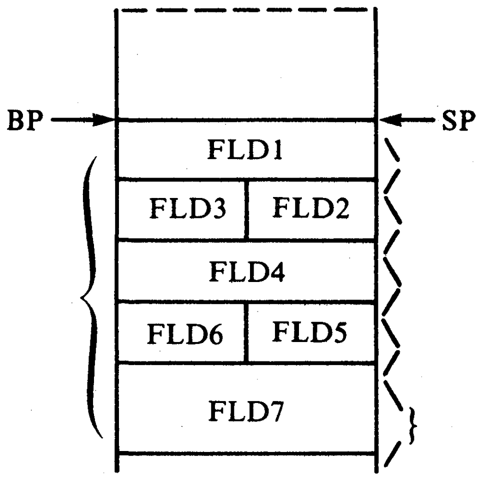
Be sure to include the period (.) in the operand.

Example:

```
ZOO      STRUC
GIRAFFE DE ?
ZOO      ENDS

LONG_NECK   ZOO   16

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

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

```
[BP].FLD6
```



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

Operators

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

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

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

ATTRIBUTE OPERATORS

Attribute operators used as operands perform one of three functions:

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

The following list shows all the attribute operators by type:

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

Value returning operators
  SEG
  OFFSET
  .TYPE
  LENGTH
  SIZE

Record specific operators
  Shift count (field name)
  WIDTH
  MASK

Override Operators

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

Pointer (PTR)

```
<attribute>  PTR  <expression>
```

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

<attribute> is the new attribute: the new type or new distance.

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

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

The second use of PTR is to access data by type other than the type in the variable definition. Most often this occurs in structures. If the structure is defined as WORD but you want to access an item as a byte, PTR is the operator to use. However, a much easier method is to enter a second statement that defines the structure in bytes too. This eliminates the need to use PTR for every reference to the structure. Refer to the LABEL directive in the section Memory Directives.

Examples:

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

ADD BYTE PTR FOO,9
```

Segment Override (:) (colon)

```
<segment-register> : <address-expression>
<segment-name> : <address-expression>
<group-name> : <address-expression>
```

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

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

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

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

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

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

Examples:

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

MOV CSEG;FAR_LABEL,AX

MOV AX,OFFSET DGROUP:VARIABLE
```

SHORT

```
SHORT <label>
```

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

Example:

```
JMP SHORT REPEAT

          .
          .
          .
```

REPEAT:

THIS

THIS <distance>
THIS <type>

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

The argument to THIS may be a distance (NEAR or FAR) or a type (BYTE, WORD, or DWORD).

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

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

Examples:

TAG EQU THIS BYTE same as TAG LABEL BYTE

SPOT_CHECK = THIS NEAR same as SPOT_CHECK LABEL NEAR

HIGH/LOW

HIGH <expression>
LOW <expression>

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

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

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

Examples:

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

MOV AL,LOW OFFFFH
```

Value Returning Operators

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

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

Because variables in MACRO-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 the size of the variable.
.TYPE	Isolates type of the variable.
LENGTH and SIZE	Isolate the memory allocation.

SEG

```
SEG <label>
SEG <variable>
```

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

Example:

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

OFFSET

```
OFFSET <label>
OFFSET <variable>
```

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

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

NOTE

OFFSET does not make the value a constant. Only MS-LINK can resolve the final value. Also, OFFSET is not required with uses of the DW or DO directives. The assembler applies an implicit OFFSET to variables in address expressions following DW and DO.

Example:

```
MOV BX,OFFSET FOO
```

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

```
TYPE <label>
TYPE <variable>
```

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

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

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

Examples:

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

.TYPE

.TYPE <variable>

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

The byte that is returned is configured so that the lower two bits are the mode. If the lower two bits are

0	the mode is absolute
1	the mode is program related
2	the mode is data related.

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

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

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

Example:

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

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

LENGTH

    LENGTH <variable>

LENGTH accepts only one variable as its argument.

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

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

If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples:

```
FOO DW 100 DUP(1)

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

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

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

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE

SIZE <variable>

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

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

Example:

```
FOO DW 100 DUP(1)

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

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

Record Specific Operators

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

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

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

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

- FOO — a record defined by the RECORD directive FOO RECORD FIELD1:3,FIELD2:6,FIELD3:7

- BAZ — a variable used to allocate FOO BAZ FOO < >
- FIELD1, FIELD2, and FIELD3 — the fields of the record FOO

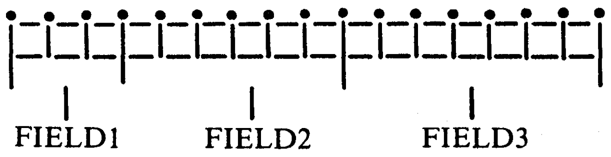
Shift-count — (Record fieldname)

<record-fieldname>

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

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

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



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

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

Example:

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

FIELD2 is now right shifted, ready for access.

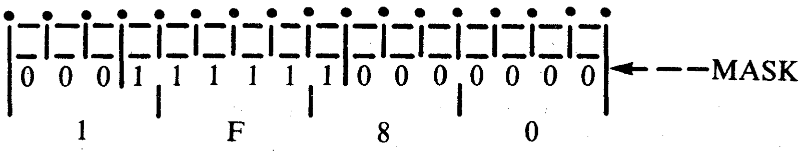
MASK

MASK <record-fieldname>

MASK accepts a field name as its only argument.

MASK returns a bit mask defined by 1 for 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.

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



The MASK of FIELD2 equals 1F80H.

Example:

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

FIELD2 is now isolated.

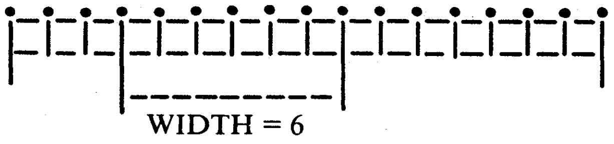
WIDTH

```
WIDTH <record-fieldname>
WIDTH <record>
```

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

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

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



Where: the WIDTH of FIELD1 equals 3
the WIDTH of FIELD2 equals 6
the WIDTH of FIELD3 equals 7.

Example:

```
    MOV CL, WIDTH FIELD2
```

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

ARITHMETIC OPERATORS

Eight arithmetic operators provide the common mathematical functions (add, subtract, divide, multiply, modulo, 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.

Table 2-4 lists the arithmetic operators with appropriate examples.

Table 2-4 Arithmetic Operators

OPERATOR	OPERATION	EXAMPLE
*	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.	MOV AX,100 MOD 17 The value moved into AX will be OFH (decimal 15).

Table 2-4 Arithmetic Operators (cont'd)

OPERATOR	OPERATION	EXAMPLE
SHR	Shift Right. SHR is followed by an integer that specifies the number of bit positions the value is to be right shifted.	MOV AX,1100000B SHR 5 The value moved into AX will be 11B (03).
SHL	Shift Left. SHL is followed by an integer that specifies the number of bit positions the value is to be lift shifted.	MOV AX,0110B SHL 5 The value moved into AX will be 011000000B (OCOH).
– (Unary Minus)	Indicates that following value is negative, as in a negative integer.	
+	Add. One operand must be a constant; one may be a non-constant.	
–	Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be non-constants. But, the right may be a nonconstant only if the left is also a nonconstant and in the same segment.	

RELATIONAL OPERATORS

Relational operators compare two constant operands.

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

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

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

Table 2-5 summarizes the six relational operators.

Table 2-5 Relational Operators

OPERATOR	OPERATION
EQ	Equal. Returns true if the operands equal each other.
NE	Not equal. Returns true if the operands are not equal to each other.
LT	Less than. Returns true if the left operand is less than the right operand.
LE	Less than or equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater than. Returns true if the left operand is greater than the right operand.
GE	Greater than or equal. Returns true if the left operand is greater than or equal to the right operand.

LOGICAL OPERATORS

Logical operators compare two constant operands bitwise.

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

Logical operators can be used in two ways:

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

The four logical operators are summarized in Table 2-6.

Table 2-6 Logical Operators

OPERATOR	OPERATION
NOT	Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.
AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values.
XOR	Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values.

Precedence of Operators in Expression Evaluation

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

Parentheses can be used to alter precedence, for example:

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

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

SHL and * are of 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

ACTION: INSTRUCTIONS AND DIRECTIVES

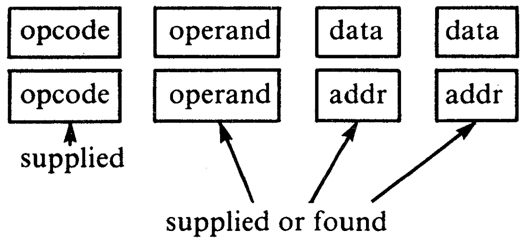
The action field contains either an 8086 instruction mnemonic or a MACRO-86 assembler directive.

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

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

Instructions

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



where *supplied* = part of the instruction and

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

opcode = the binary code for the action of an instruction.

This guide does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this, you will need to consult other texts. Some texts that you can reference are the following:

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

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

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

However, Table 2-7 lists the 8086 instructions alphabetically by their mnemonics. Following Table 2-7, the mnemonics are listed again, first according to the types of instructions they are used in and then by the arguments they take, if any.

Table 2-7 8086 Instructions

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
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

Table 2-7 8086 Instructions (cont'd)

MNEMONIC	FULL NAME
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero
LOOPZ	LOOP while zero

Table 2-7 8086 Instructions (cont'd)

MNEMONIC	FULL NAME
MOV	Move
MOVS	Move byte or word (of string)
MOVBS	Move byte (string)
MOBSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
BIT	BIT
OR	OR
OUT	Output byte or word
POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SCAS	Scan byte or word (of string)
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Subtract

Table 2-7 8086 Instructions (cont'd)

MNEMONIC	FULL NAME
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

In the following sections, the 8086 instruction mnemonics are grouped by instruction type, then by the type of argument(s) they take. In each group showing arguments, the instructions are listed alphabetically in the first column. Then, 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.

These abbreviations are used in the discussion:

OP = opcode; instruction mnemonic

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

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

accum = AX or AL register

immed = immediate

mem = memory operand

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

GENERAL 2 OPERAND INSTRUCTIONS

ADC	OP reg,r/m
ADD	OP r/m,reg
AND	OP accum,immed
CMP	OPr/m,immed
OR	
SBB	
SUB	
TEST	
XOR	

CALL AND JUMP TYPE INSTRUCTIONS

CALL	OP mem NEAR FAR direction
JMP	OP r/m (indirect data -- DWORD, WORD)

RELATIVE JUMPS

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

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

LOOP INSTRUCTIONS: SAME AS RELATIVE JUMPS

LOOP LOOPE LOOPZ LOOPNE LOOPNZ

RETURN INSTRUCTION

Mnemonic	Argument Type
RET	[immed] (optional, number of words to POP)

NO OPERAND INSTRUCTIONS

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

LOAD INSTRUCTIONS

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

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 ret,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)
INDIV	
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-numer,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 = 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

LOCK REP REPE REPZ REPNE REPNZ

Directives

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

In the following discussion, the directives have been divided into groups according to the function they perform. Within each group, the directives are described alphabetically.

- 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.

The following is an alphabetical list of all the directives that MACRO-86 supports.

ASSUME	EVEN	IRPC	.RADIX
	EXITM		RECORD
COMMENT	EXTERN	LABEL	REPT
CREF		.LALL	
	GROUP	.LFCOND	.SALL
DB		.LIST	SEGMENT
DD	IF		.SFCOND

DQ	IFB	MACRO	STRUC
DT	IFDEF		SUBTTL
DW	IFDIF	NAME	
	IFE		.TFCOND
ELSE	IFIDN	ORG	TITLE
END	IFNB	%OUT	
ENDIF	IFNDEF		.XALL
ENDM		PAGE	.XCREF
ENDP	IF1	PROC	.XLIST
ENDS	IF2	PUBLIC	
EQU	IRP	PURGE	

The directives listed above are described in the following sections under the directive type.

MEMORY DIRECTIVES

ASSUME

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

or

ASSUME NOTHING

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register. When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from 1 to 4 arguments to ASSUME.

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

The possible entries for <seg-name> are

- the name of a segment declared with the SEGMENT directive
- the name of a group declared with the GROUP directive
- an expression: either SEG <variable-name> or SEG <label-name> (see the SEG operator in the section **Operands**)
- 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, DSLFOO instead of simply FOO).

Example:

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

COMMENT

```
COMMENT <delim> <text> <delim>
```

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

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

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

Example:

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

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

DEFINE BYTE
DEFINE WORD
DEFINE DOUBLEWORD
DEFINE QUADWORD
DEFINE TENBYTES

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

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

If the optional <varname> is entered, the DEFINE directives define the name as a variable. If <varname> has a colon, it becomes a NEAR label instead of a variable. (See also the sections **Labels** and **Variables**.)

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

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

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

- A constant expression
- The character ? for indeterminate initialization. Usually the ? is used to reserve space without placing any particular value into it. It is the equivalent of the DS pseudo-op in MACRO-80.
- An address expression (for DW and DD only)
- An ASCII string (longer than two characters for DB only)
- <exp> DUP(?)

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

  This expression, like item 5, produces a data block, but initialized with the value of the second <exp>. The first <exp> must be a constant greater than zero and must not be a forward reference.

Example — Define Byte (DB):

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

Example — Define Word (DW):

```
ITEMS           DW   TABLE,TABLE+10,TABLE+20
SEGVAL          DW   OFFFOH
BSIZE           DW   4 * 128
LOCATION        DW   TOTAL + 1
AREA            DW   100 DUP(?)
CLEARED         DW   50 DUP(4),7)
                     ;100 copies of bytes with values 4,4,4,4,4,7
NEW_PAGE        DB   OCH
                     ;form feed character
ARRAY           DB   1,2,3,4,5,6,7
```

Example — Define Doubleword (DD):

```
DBPTR           DD   TABLE
                     ;16-bit OFFSET, then 16-bit
                     ;SEG base value
SEC_PER_DAY     DD   60*60*24
                     ;arithmetic is performed
                     ;by the assembler
```

```
LIST             DD   'XY',2 DUP(?)
HIGH             DD   4294967295
                      ;maximum
FLOAT            DD   6.735E2
                      ;floating point
```

Example — Define Quadword (DQ):

```
LONG_REAL        DQ   3.141597
                      ;decimal makes it real
STRING           DQ   'AB'
                      ;no more than 2 characters
HIGH             DQ   18446744073709661615
                      ;maximum
LOW              DQ   -18446744073709661615
                      ;minimum
SPACER           DQ   2 DUP(?)
                      ;uninitialized data
FILLER           DQ   1 DUP(?,?)
                      ;initialized with
                      ;indeterminate value
HEX_REAL         DQ   0FDCBA9A98765432105R
```

Example — Define Tenbytes (DT):

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

END

```
END  [ <exp> ]
```

The END statement specifies the end of the program.

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

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

Example:

```
END START
```

where START is a label somewhere in the program.

EQU

```
<name>  EQU <exp>
```

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

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

<exp> may be any one of the following:

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

Example:

```
FOO  EQU  BAZ              ;must be defined in this
                           ;module or an error results
B    EQU  [BP+8]           ;index reference (Text)
P8   EQU  DS: [BP+8]       ;segment prefix
                           ;and operand (Text)
CBD  EQU  AAD              ;an instruction name
(Opcode)
All  EQU  DEFREC 2,3,4,    ;DEFREC = record name
                           ;2,3,4, = initial values for fields of record
```

```
EMP  EQU   6                ;constant value
FPV  EQU   6.3E7            ;floating point (text)
```

Equal Sign

```
<name>      =       <exp>
```

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

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

Example:

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

EVEN

```
EVEN
```

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

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

Example:

Before: the PC points to 0019 hex (25 decimal).

```
EVEN
```

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

EXTRN

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

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

<type> may be any one of the following, but must be a valid type for <name>.

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

Unlike the 8080 assembler, placement of the EXTRN directive is significant. If the directive is given with a segment, MACRO-86 assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments, then use either

ASSUME <seg-reg> :SEG <name>

or an explicit segment prefix.

NOTE

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

Example:

In Same Segment:

In Module 1:

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

In Module 2:

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

In Another Segment:

In Module 1:

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

In Module 2:

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

GROUP

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

The GROUP directive collects the segments named after GROUP (<seg-name> s) under one name. The GROUP is used by MS-LINK so that 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 MS-LINK in response to the Object module prompt (where you name the object modules in the order they are to be linked).

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

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

- A segment name, assigned by a SEGMENT directive. The name may be a forward reference.
- An expression: either SEG <var> or SEG <label>

  Both of these entries resolve themselves to a segment name (see the SEG operator in the section **Operands**).

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

- As an immediate value.

  Example:

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

  DGROUP is the paragraph address of the base of DGROUP.
- In an ASSUME statement.

  Example:

  ```
  ASSUME DS:DGROUP
  ```

  The DS register can now be used to reach any symbol in any segment of the group.
- As an operand prefix for segment override.

  Example:

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

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

Example (using GROUP to combine segments):

In Module A:

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

In Module B:

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

INCLUDE

INCLUDE <filename>

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

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

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

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

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

On a MACRO-86 listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See the section FORMATS OF LISTINGS AND SYMBOL TABLES for a description of listing file formats.

Example:

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

LABEL

```
<name>       LABEL       <type>
```

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

The item is assigned a length of 1.

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

- 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 <name>: (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:

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

- For data:

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

  Example:

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

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

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

NAME

```
NAME       <module-name>
```

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

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

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

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

Example:

```
    NAME CURSOR
```

ORG

```
    ORG      <exp>
```

The location counter is set to the value of <exp>, and the MACRO-86 assigns generated code starting with that value.

All names used in <exp> must be known on pass 1. The value of <exp> must either evaluate to an absolute or must be in the same segment as the location counter.

Example:

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

Example — ORG to a boundary (conditional):

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

See the section CONDITIONAL DIRECTIVES for an explanation of conditional assembly.

PROC

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

The default, if no operand is specified, is NEAR. Use FAR if either of these two conditions apply.

- The procedure name is an operating system entry point.
- The procedure will be called from code which has another ASSUME CS value.

Each PROC block should contain a RET statement.

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

The PROC directive, through the NEAR/FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL, and RETs to generate a NEAR or a FAR RET. PROC is used, therefore, for coding simplification so that 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.

Example:

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

                PUBLIC      NEAR_NAME
NEAR_NAME       PROC        NEAR
                .
                .
                .
                RET
NEAR_NAME       ENDP
```

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

```
CALL NEAR_NAME
```

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

```
CALL FAR_NAME
```

PUBLIC

```
PUBLIC      <symbol> [,...]
```

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

<symbol> may be a number, a variable, a label (including PROC labels).

<symbol> may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes.

Example — valid PUBLIC:

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

Example — illegal PUBLIC:

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

.RADIX

```
.RADIX      <exp>
```

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

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

Example:

```
MOV         BX,OFFH
.RADIX      16
MOV         BX,OFF
```

The two MOVs in this example are identical.

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

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

Example:

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

RECORD

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

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

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

<width> is a constant in the range 1 to 16 that specifies the number of bits contained in the field defined by <fieldname>. The WIDTH operator returns this value. If the total width of all declared fields is larger than 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 thè last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

Example:

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

Initially, the bit map would be

```
    ._._._._._._._._._._._._._._._._.
     _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
      <HIGH->  <MID> <LOW>
```

If the total bits are >8, a word is used. If the total bits are <16, they are right shifted with undeclared bits placed at high end of word, thus:

```
 0   0   0   0   0   0   1   1   1   1   0   0   0   0   0   0    <---MASK
•_•_•_•_•_•_•_•_•_•_•_•_•_•_•_•_•

 _   _   _   _   _   _   _   _   _   _   _   _   _   _   _   _
       not            <HIGH->  <MID> <LOW>
     declared         -------  ----------->
                       WIDTH   shift  count
```

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

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

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

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

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

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

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

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

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

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

Example:

```
FOO     RECORD   HIGH:5,MID:3,LOW:3
        .
        .
        .
BAX     FOO      < >; leave indeterminate here
JANE    FOO      10 DUP( <16,8> )   ;HIGH=16,MID=8,LOW=?


        MOV      DX,OFFSET JANE [2]
                 ;get beginning record address
        AND      DX,MASK MID
        MOV      CL,MID
        SHR      DX,CL
        MOV      CL,WIDTH MID
```

SEGMENT

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

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

The <segname> must be an unique, legal name. 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," or private).

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

All three operands are passed to MS-LINK.

The <align> operand

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

PAGE — address is xxx00H (low byte is 0)

PARA — address is xxxx0H (low nibble is 0)
bit map — x x x x 0 0 0 0

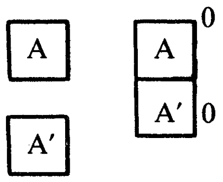
WORD — address is xxxxeH (e=even number; low bit is 0)
bit map — x x x x x x x 0

BYTE — address is xxxxxH (placed anywhere)

The <combine> operand

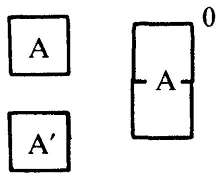
The combine type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or private)



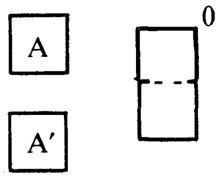
Private segments are loaded separately and remain separate. They may be physically but not logically contiguous, even if the segments have the same name. Each private segment has its own base address.

Public and stack



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

Common



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Memory

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

NOTE

This feature is not supported by MS-LINK. MS-LINK treats memory segments the same as public segments.

At <exp>

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

NOTE

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

The <'class'> operand

<'class'> name must be enclosed in quotation marks. It may be any legal name. Refer to Chapter 4 on MS-LINK for more discussion.

Segment definitions may be nested. When segments are nested, the assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, MACRO-86 takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. Overlapping segments are not permitted.

Example 1:

```
A  SEGMENT          A  SEGMENT
        .                  .
        .                  .
        .                  .
B  SEGMENT          A  ENDS
        .           B  SEGMENT
        .     --->         .
        .                  .
B  ENDS                    .
        .           B  ENDS
        .           A  SEGMENT
        .                  .
A  ENDS                    .
                           .
                    A  ENDS
```

The following arrangement is not allowed.

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

Example 2:

In module A:

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

In module B:

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

STRUC

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

The STRUC directive is very much like RECORD, except 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 (;) are the only statement entries allowed inside a STRUC block.

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

Example:

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

If the <exp> following the DEFINE directive contains a string, it may be overridden by another string. However, if the overriding string is shorter than the initial string, the assembler will pad with spaces. If the overriding string is longer, the assembler will truncate the extra characters.

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

```
<variable> . <field>
```

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

```
FOO     STRUCTURE
        .
        .
        .
FOO     ENDS

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

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

Examples:

Assume you define a structure:

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

The DEFINE directives in this example define the fields of the structure and the field 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, you write:

```
DBAREA   S    <,,7,'ANDY'>     ;overrides 3rd and 4th
                               ;fields only
```

To refer to the structure .FIELD, you append it to the operand as follows:

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

CONDITIONAL DIRECTIVES

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

All conditionals follow the format:

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

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

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

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid phase errors and incorrect evaluation. For IF and IFE, the expression must involve values that were previously defined, and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

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

IF <exp>

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

IFE <exp>

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

IF1 — Pass 1 conditional

If MACRO-86 is in pass 1, the statements in the conditional block are assembled. IF1 takes no expression.

IF2 — Pass 2 conditional

If MACRO-86 is in pass 2, the statements in the conditional block are assembled. IF2 takes no expression.

IFDEF <symbol>

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

IFNDEF <symbol>

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

IFB <arg>

The angle brackets around <arg> are required.

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

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

IFNB <arg>

The angle brackets around <arg> are required.

If <arg> is not blank, the statements in the conditional block are assembled.

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

IFIDN <arg1>,<arg2>

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

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFIDN (and IFDIF) are normally used inside macro blocks. The expression following the IFIDN directive is typically two dummy symbols. When the macro is called, the dummys will be replaced by parameters passed by the macro call. If the macro call specifies two identical parameters to replace the dummys, the block will be assembled. (IFDIF is the opposite case.) Refer to the section MACRO DIRECTIVES for a full explanation.

IFDIF <arg1>,<arg2>

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

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

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

ELSE

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

ENDIF

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

MACRO DIRECTIVES

The macro directives allow you to write blocks of code that can be repeated without recording. 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 used by MACRO-86 include:

Macro definition directive

MACRO

Termination directives

ENDM
EXITM

Directive to create unique symbols within macro blocks

LOCAL

Directive to undefine a macro

PURGE

Repeat directives

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

The macro directives also include some special macro operators:

& ;; ! %

Macro Definition Directive

MACRO

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

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

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

A <dummy> is formed as any other name is formed. It is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. You should include all dummys used inside the macro block on this line. The number of dummys is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. MACRO-86 interprets a series of dummys the same as any list of symbol names.

NOTE

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

One alternative is to list no dummys.

```
<name> MACRO
```

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

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

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instructions: mnemonic, directive, label, variable, or symbol. When MACRO-86 evaluates a statement, it first looks at the macro table it builds during pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: MACRO-86 evaluates macros, then instruction mnemonics/directives.)

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

To use a macro, enter a macro call statement in the format:

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

<name> is the name of the macro block. A <parameter> replaces a dummy on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, MACRO-86 will pass all the items inside the angle brackets as a single parameter.

For example:

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

passes five parameters to the macro, but

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

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummys in the macro definition. If there are more parameters than dummys, the extra dummys will be made null. The assembled code will include the macro block after each macro call statement.

Example:

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

If you then enter a macro call statement:

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

MACRO-86 generates the statements:

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

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

Termination Directives

End Macro

```
    ENDM
```

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

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM directive. Otherwise, the "Unterminated REPT/IRP/IRPC/MACRO" message is generated at the end of each pass. An unmatched ENDM also causes an error.

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

Exit Macro

```
    EXITM
```

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

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

Example:

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

Directive for Unique Symbol Creation within a Macro
LOCAL

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

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

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

Example:

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

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

Macro Undefine Directive

PURGE

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

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

PURGE provides three benefits:

- It frees text space of the macro body.
- It returns any instruction mnemonics or directives that were redefined by macros to their original function.
- It allows you to "edit out" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method allows you to use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you will not use in this program.

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

Example:

```
INCLUDE    MACRO.LIB
PURGE      MAC1
MAC1                         ;tries to invoke purged macro
                             ;returns a syntax error
```

Repeat Directives

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

- MACRO gives the block a name by which to call in the code wherever and whenever needed. The macro block can be used in many different programs by simply entering a macro call statement.
- MACRO allows parameters to be passed to the MACRO block when a macro is called; hence, parameters can be changed.

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

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

Repeat

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

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

Example:

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

assembles as:

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

Indefinite Repeat

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

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

Example:

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

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

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

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

When the macro call statement:

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

is assembled, the macro expansion becomes

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

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

Indefinite Repeat Character

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

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

Example:

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

This example generates the same code (CB 1 - DB 10) as the two previous examples.

Special Macro Operators

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

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

Example:

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

The call ERRGEN A will then generate

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

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

Example:

```
Correct form                   Incorrect form

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

When called, for example, by FOO BAZ, the expansion would be correct in the left column, incorrect in the right. The operation would proceed as follows:

1. MACRO build. Find dummies and change to d1.

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

2. MACRO expansion. Substitute parameter text for d1.

```
        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 d1.

```
BAS&d1  DB      d1          BAZZ    DB      d1
```

4. IRP expansion. Substitute parameter text for d1.

```
BAZ1    DB      1           BAZZ    DB      1
BAZ2    DB      2           BAZZ    DB      2
BAZ3    DB      3           BAZZ    DB      3

                                ;here it's an error,
                                ;multi-defined symbol
```

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

- All text within the angle brackets are seen as a single parameter, even if commas are used.
- Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows.

One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets around parameters as there are levels of nesting.

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

The default listing condition for macros is .XALL (see the section LISTING DIRECTIVES). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

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

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

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

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

Example:

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

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

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

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

```
         %OUT     * SYM1 1m1 SYM2 = 300 *
```

LISTING DIRECTIVES

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

PAGE

```
PAGE [ <length> ][, <width> ]
PAGE [+]
```

PAGE with no arguments or with the optional [+] argument causes MACRO-86 to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

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

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

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

The plus sign increments the major page number and resets the minor page number to 1. Page numbers are in the form major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Example:

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

TITLE

    TITLE <text>

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

Example:

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

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

SUBTITLE

    SUBTTL <text>

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

Any number of SUBTTLs may be given in a program. Each time MACRO-86 encounters SUBTTL, it replaces the <text> from the previous SUBTTL with the <text> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <text>.

Example:

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

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

%OUT

```
%OUT <text>
```

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

%OUT will output on both assembler passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See the section CONDITIONAL DIRECTIVES for descriptions of the IF1 and IF2 directives.

Example:

The assembler will send the following messages to the APC screen for passes 1 and 2 of program assembly when the %OUT statements are encountered:

```
IF1
%OUT *Pass 1 started*
ENDIF

IF2
%OUT *Pass 2 started*
ENDIF
```

.LIST
.XLIST

.LIST lists all lines with their code (the default condition) on the printer.

.XLIST suppresses all listing.

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

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing directives. So, nothing will be listed, even if another listing directive (other than .LIST) is encountered.

Example:

```
    .
    .
    .
    .XLIST        ;listing suspended here
    .
    .
    .
    .LIST         ;listing resumes here
```

.SFCOND

.SFCOND suppresses portions of the listing containing conditional expressions that evaluate as false.

.LFCOND

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

.TFCOND

.TFCOND toggles the current setting. It operates independently from .LFCOND and .SFCOND. It toggles the default setting, which is set by the presence or absence of the /X switch when running the assembler. When /X is used, .TFCOND will cause false conditionals to list. When /X is not used, .TFCOND will suppress false conditionals.

.XALL

.XALL is the default. It lists source code and object code produced by a macro, but source lines that do not generate code are not listed.

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code. Comments preceded by two semicolons (;;) will not be listed.

.SALL

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

.CREF
.XCREF [<variable list>]

.CREF is the default condition. .CREF remains in effect until MACRO-86 encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until MACRO-86 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the source file. Use .CREF to restart the creation of a cross reference file after using the .XCREF directive.

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

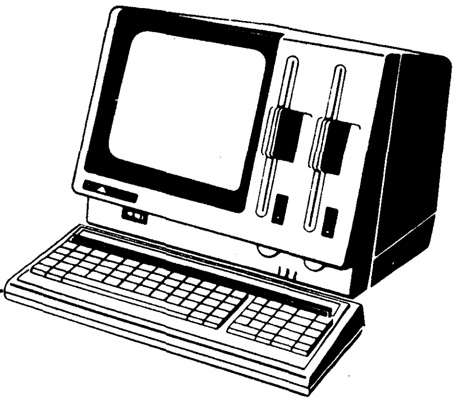
Neither .CREF nor .XCREF without arguments takes effect unless you specify a cross reference file when running the assembler. .XCREF variable list suppresses the variables from the symbol table listing regardless of the creation of a cross reference file.

Example:

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

Chapter 3

Assembling a Macro-86 Source File



Microsoft's MACRO-86 Macro Assembler is a very rich and powerful assembler for 8086 based computers. MACRO-86 incorporates many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from an 8086 or 8088 microprocessor. Even though MACRO-86 is more complex than any other microcomputer assembler, it is easy to use.

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

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

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

This block of code is given a name, called a macro. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, a simple "call" to the macro is placed in the source file. MACRO-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 a program, MACRO-86 first checks the symbol table it created to see if the instruction or directive is a macro name. If it is, MACRO-86 "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-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 non-blank arguments, for defined or not-defined symbols, for equivalence, and 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 the conditionals inside conditionals (nesting). Conditional assembly blocks can be nested up to 255 levels.

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

MACRO-86 provides some relaxed typing. Some 8086 instructions take only one operand type. If a typeless operand is entered for an instruction that accepts only one type of operand (for example, in the instruction PUSH [JBS], [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 insructions, not just NOPs. For example, if you enter

```
    MOV AL, WORDLBL
```

You may have meant one of three instructions:

```
    MOV AL, BYTE PTR WORDLBL
    MOV AL, other
    MOV AX, WORDLBL
```

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

SYSTEM REQUIREMENTS FOR RUNNING MACRO-86

The MACRO-86 Macro Assembler requires 96K bytes of memory minimum to execute: 64K bytes for code and static data, and 32K bytes for run space. For a peripheral device, MACRO-86 needs one disk drive, if and only if output is sent to the same physical diskette from which the input was taken. MACRO-86 does not allow time to swap diskettes during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

OVERVIEW OF MACRO-86 OPERATIONS

This first task is to create a source file. Use EDLIN, the resident editor in MS-DOS operating system (or other 8086 editor compatible with your MS-DOS operating system) to create the MACRO-86 source file. MACRO-86 assumes a default file name extension of .ASM for the source file. Creating the source file involves coding instruction and directive statements that follow the rules and constraints described in Chapter 2 of this guide. When the source file is ready, run MACRO-86.

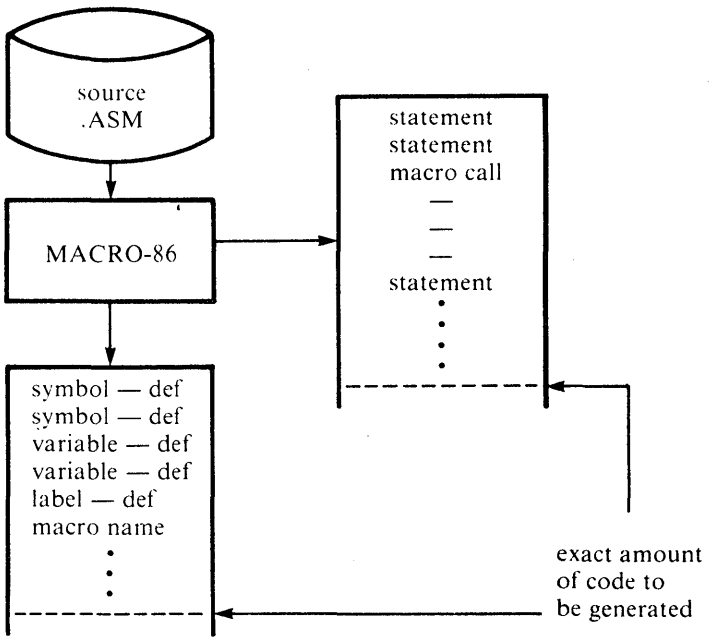
MACRO-86 is a two-pass assembler. This means that the source file is assembled twice. However, slightly different actions occur during each pass.

During the first pass, the assembler evaluates the statements and expands macro call statements; calculates the amount of the code it will generate; and builds a symbol table where all symbols, variables, labels, and macros are assigned values. During the second pass, the assembler fills in the symbol, variable, labels and expression values from the symbol table; expands macro call statements; and emits the relocatable object code into a file that is suitable for processing with MS-LINK. The .OBJ file can be stored as part of a libary of object programs, which later can be linked with one or more .OBJ modules by MS-LINK. The .OBJ modules can also be processed with the MS-LIB Library Manager.

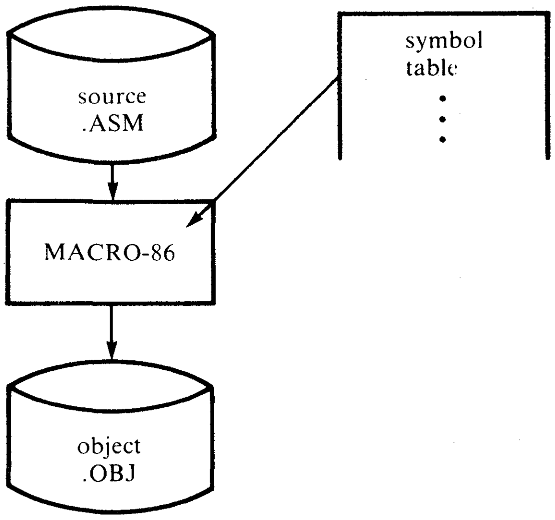
The source file can also be assembled without creating an .OBJ file. All the other assembly steps are performed, but the object code is not sent to disk. Only erroneous source statements are displayed on the console screen. This practice is useful for checking the source code for errors. It is faster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

The following illustrates the operations of MACRO-86 during its two passes:

PASS 1


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

PASS 2


source
.ASM
symbol
table
MACRO-86
object
.OBJ

MACRO-86 will create, on command, a listing file and a cross-reference file.

The listing file contains the beginning relative addresses (offsets from the segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. The listing that is generated contains a symbol table that shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file is given the default filename extension .LST.

The cross reference file contains a compact representation of variables, labels, and symbols. This file is given the default filename extension .CRF. When the cross reference file is processed by MS-CREF, it is converted into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order, followed by the line number in the source program where each is defined, followed by the line numbers where each is used in the program. The final cross reference listing receives the filename extension .REF. (Refer to Chapter 5, THE CROSS REFERENCE UTILITY for further explanation and instructions.)

HOW TO ASSEMBLE A MACRO-86 SOURCE FILE

Assembling with MACRO-86 requires invoking MACRO-86 and answering command prompts. In addition, four switches control alternate MACRO-86 features. Usually, you will enter all the commands to MACRO-86 at the keyboard. As an option, answers to the command prompts and any switches may be contained in a batch file (see the *MS-DOS System User's Guide* for batch file processing instructions). Some command characters are provided to assist you in entering assembler commands.

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 command line used to invoke MACRO-86.

Method 1: MASM

Enter

```
MASM
```

MACRO-86 is loaded into memory, then returns a series of four text prompts. You must answer the prompts as commands to MACRO-86.

At the end of each line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, MACRO-86 defaults to not performing the function described for the switches.

MACRO-86 COMMAND PROMPTS

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 a filename extension,
a device designation and filename,
or a device designation, filename, and filename extension.

You may not enter only a filename extension.

Table 3-1 summarizes the MACRO-86 command prompts.

Table 3-1 MACRO-86 Command Prompts

PROMPT	RESPONSE
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 name of the file you want to receive the generated object code. If you simply press RETURN when this prompt appears, the object file will be given the same name as the source file, but with the filename extension .OBJ. If you want your object file to have a different name or a different filename extension, you must enter your choice(s) in response to this prompt. If you want to change only the filename but keep the .OBJ extension, enter the filename only. To change the extension only, you must enter both the filename and the extension.
Source listing [NUL.LST]:	Enter the name of the file, if any, you want to receive the source listing. If you press RETURN, MACRO-86 does not produce this listing file. If you enter a filename only, the listing is created and placed in a file with the name you enter plus the filename extension .LST. You may also enter your own extension. The source listing file will contain a list of all the statements in your source program and will show the code and offsets generated for each statement. The listing will also show any error messages generated during the session.

Table 3-1 MACRO-86 Command Prompts (cont'd)

PROMPT	RESPONSE
Cross reference [NUL.CRF]:	Enter the name of the file, if any, you want to receive the cross reference file. If you press only RETURN, MACRO-86 does not produce this cross reference file. If you enter a filename only, the cross reference file is created and placed in a file with the name you enter plus the filename extension .CRF. You may also enter your own extension. The cross reference file is used as the source file for the MS-CREF Cross Reference Utility. MS-CREF converts this cross reference file into a cross reference listing, which you can use to aid you during program debugging. The cross reference file contains a series of control symbols that identify records in the file. MS-CREF uses these control symbols to create a listing that shows all occurrences of every symbol in your program. The occurrence that defines the symbol is also identified.

MACRO-86 COMMAND SWITCHES

The three MACRO-86 command switches, /D, /O, and /X, control various assembler functions. These switches must be entered at the end of a prompt response, regardless of which method is used to invoke MACRO-86. They 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.

Table 3-2 contains the descriptions of the MACRO-86 command switches.

Table 3-2 MACRO-86 Command Switches

SWITCH	FUNCTION
/D	Produces a source listing on both assembler passes. The listings will, when compared, show where in the program phase errors occur and possibly, give 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	Prints the listing file in octal radix. The generated code and the offsets shown on the listing will all be given in octal. The actual code in the object file will be the same as if the /O switch were not given. The /O switch affects only the listing file.
/X	Suppresses the listing of false conditionals. If your program contains conditional blocks, the listing file will show the source statements but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not generate code, use the /X switch to suppress the blocks that evaluate false conditionals 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.

Table 3-2 MACRO-86 Command Switches (cont'd)

SWITCH	FUNCTION
	If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes the listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When MACRO-86 encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing. Of course, the /X switch has no effect if no listing is created. See additional discussion under the .TFCOND directive in Chapter 2.

The following information represents the effects of the conditional listing directives in combination with the /X switch.

PSEUDO-OP	NO /X	/X
(none)	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

COMMAND CHARACTERS

MACRO-86 provides two command characters.

; Use a single semicolon (;), followed immediately by RETURN, any time after responding to the first prompt (from Source filename on) to select default responses to the remaining prompts. This feature saves time.

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 prompts. For this, use the RETURN key.

Example:

```
Source filename [.ASM]:   FUN  RETURN
Object filename [FUN.OBJ]:   ;  RETURN
```

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

To achieve exactly the same result, you could alternatively enter

```
Source filename [.ASM]:   FUN;  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 2: MASM filenames [switches]

Enter

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

MACRO-86 is loaded into memory, then immediately begins assembly. The entries following MASM are responses to the command prompts. The entry fields for the different prompts must be separated by commas,

where: <source> is the source filename

<object> is the name of the file to receive the relocatable output

<listing> is the name of the file to receive the listing

<cross-ref> is the name of the file to receive the cross reference output

/switch are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <cross-ref>, as shown).

To select the default for a field, simply enter a second comma without space in between.

Example:

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

This example causes MACRO-86 to be loaded, then causes the source file FUN.ASM to be assembled. MACRO-86 then outputs the relocatable object code to a file named FUN.OBJ (default caused by two commas in a row), creates a listing file named FUN.LST for both assembly passes but with false conditionals suppressed, and creates a cross reference file named FUN.CRF. If filenames are not entered for listings and a cross reference, these files are not to be created. If listing file switches are given but no filename, the switches are ignored.

FORMATS OF LISTINGS AND SYMBOLS TABLES

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

The first part of the listing shows:

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

The second part of the listing shows:

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

Program Listing

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

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

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.

Table 3-3 provides a summary of listing symbols.

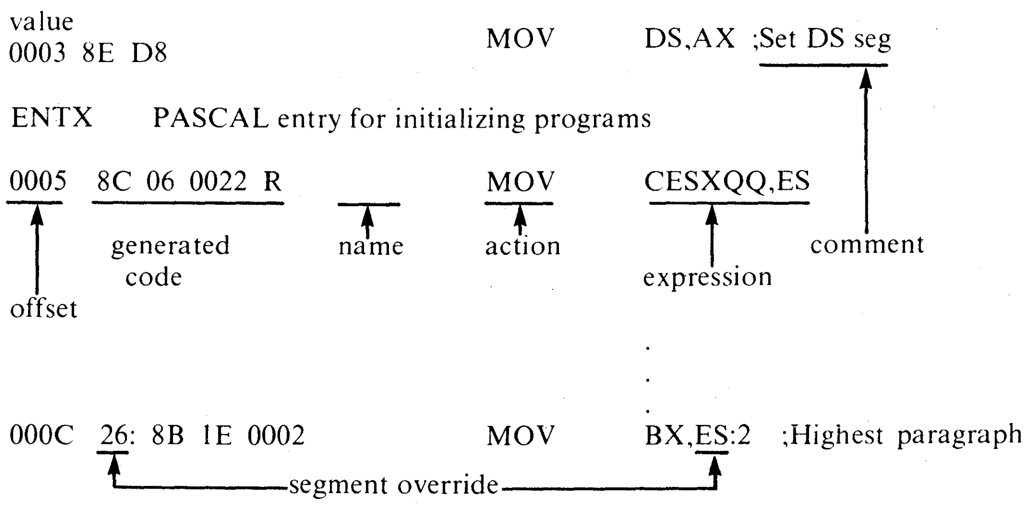
Table 3-3 MACRO-86 Source Program Listing Symbols

LISTING SYMBOL	ACTION
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. For example: 003C F3/ A5 REP MOVSW ;move DS:SI to ES:DI until CX=0
[xx]	DUP expression; xx is the value in parentheses following DUP. For example: DUP(?) places "??" where "xx" is shown here.
+	Line comes from a macro expansion.
C	Line comes from file named in INCLUDE directive statement.

A sample of a source listing for an assembled program follows. Several items have been underlined and notated to improve readability.

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

ENTX     PASCAL entry for initializing programs

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

                                           ;Base of heap before init
done
0000      14 [                    DB       20 DUP(?)
              ??<-- shows value in parentheses
                ]
              Indicates DUP expression
= 0014               SKTOP        EQU      THIS BYTE
0014              STACK           ENDS

0000 MAINSTARTUP SEGMENT 'MEMORY'
                     DGROUP       GROUP    DATA,STACK,CONST,
                                  ASSUME   HEAP,MEMORY
                                           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
```

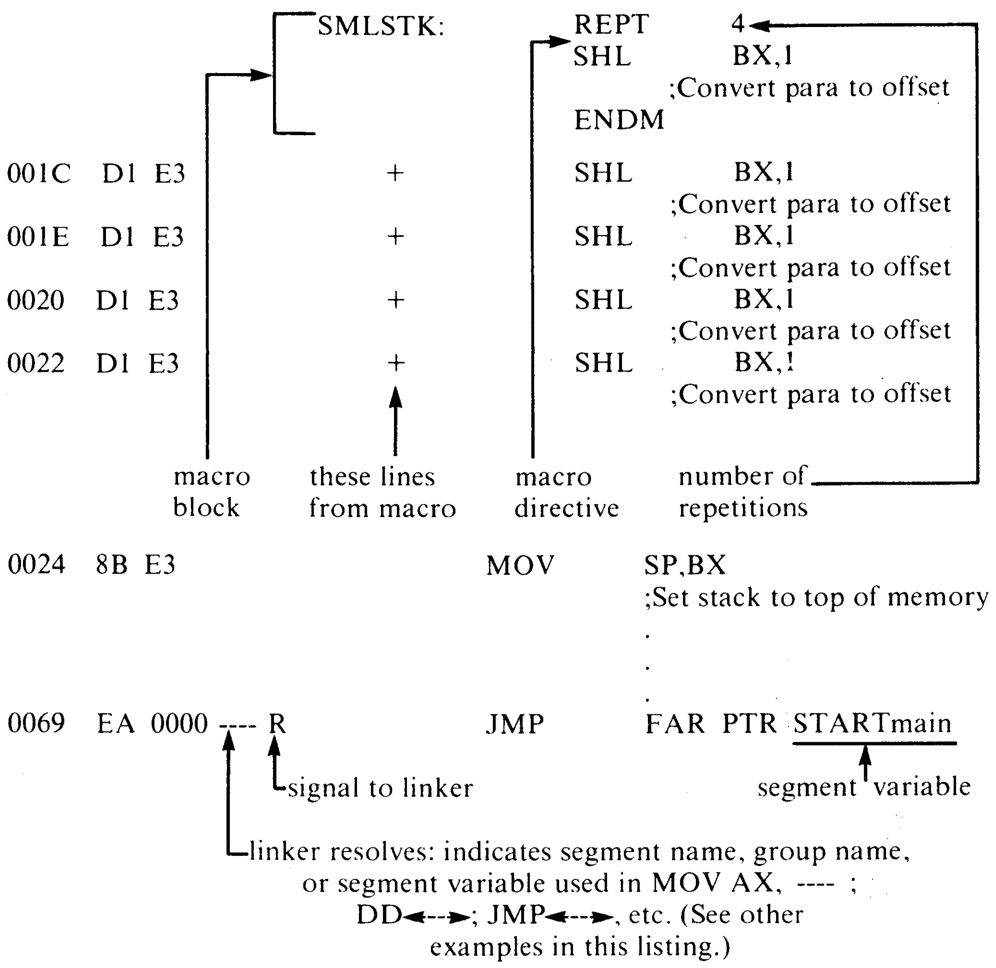
```
Microsoft MACRO-86 MACRO Assembler    1-Dec-81

ENTX      PASCAL entry for initializing programs
```



```
value
0003 8E D8                        MOV      DS,AX  ;Set DS seg

ENTX      PASCAL entry for initializing programs

0005  8C 06 0022 R                MOV      CESXQQ,ES
                 .
                 .
                 .
000C  26: 8B 1E 0002              MOV      BX,ES:2   ;Highest paragraph

0011  2B D8                       SUB      BX,AX  ;Get # paras for DS
0013  81 FB 1000                  CMP      BX,4096 ;More than 64K?
0017  7E 03                       JLE      SMLSTK  ;No, use what we have
0019  BB 1000                     MOV      BX,4096 ;Can only address 64K
64K
```

Microsoft MACRO-86 MACRO Assembler   1-Dec-81

ENTX   PASCAL entry for initializing programs

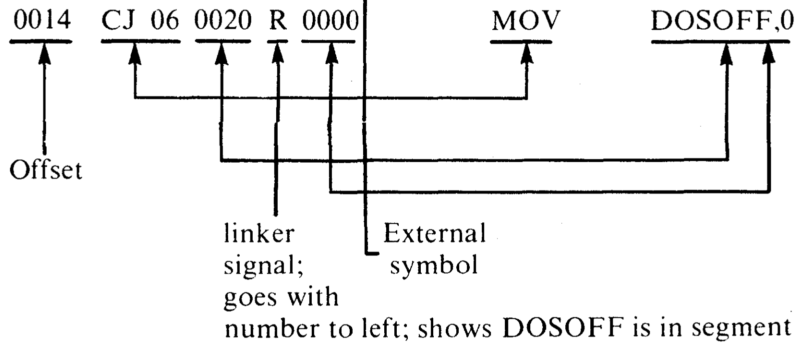


```
                       SMLSTK:         REPT     4
                                       SHL         BX,1
                                                ;Convert para to offset
                                       ENDM

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,!
                                                ;Convert para to offset

0024   8B  E3                      MOV       SP,BX
                                             ;Set stack to top of memory
                                             .
                                             .
                                             .
0069   EA  0000 ---- R             JMP       FAR  PTR  STARTmain
```

macro block
these lines from macro
macro directive
number of repetitions
signal to linker
segment variable
linker resolves: indicates segment name, group name, or segment variable used in MOV AX, ---- ; DD◄--►; JMP◄--►, etc. (See other examples in this listing.)

```
006E               BEGXQQ     ENDP
                                             .
                                             .
                                             .
007E            MAINSTARTUP                ENDS
```

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

ENTX      PASCAL entry for initializing programs

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

0000                    STARTmain PROC     FAR ;This code remains
0000  9A 0000 ---- E              CALL     ENTGQQ
                                             ;call main program
                        ;
0005                    ENDXQQ    LABEL    FAR
                                             ;termination entry point
0005  9A 0000 ---- E              CALL     ENDOQQ
                                             ;user system termination
000A  9A 0000 ---- E              CALL     ENDYQQ
                                             ;close all open files
000F  9A 0000 ---- E              CALL     ENDUQQ
                                             ;file system termination
0014  CJ 06 0020 R 0000           MOV      DOSOFF,0
```


Offset
linker signal; goes with number to left; shows DOSOFF is in segment
External symbol

```
00 2E 0020 R                      JMP      DWORD PTR DOSOFF
                                             ;return to DOS
001E                    STARTmain ENDP
                                       .
                                       .
                                       .
0037                    ENTXCM    ENDS
                                  END      BEGXQQ
```

There are differences between the pass 1 and pass 2 listings.

If you give the /D switch when you run MACRO-86 to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that assembled without reporting any errors. When the source file was reassembled using the /D switch, an error was produced on pass 1, but not on pass 2 (which is when errors are usually reported).

Example:

During pass 1 a jump with a forward reference produces:

```
0017  7E 00                  JLE     SMLSTK ;No, use what we have
Error  ---       9:Symbol not defined
0019  BB 1000                MOV     BX,4096 :Can only address 64K
001C             SMLSTK:     REPT    4
```

During pass 2 this same instruction is fixed and does not return an error.

```
0017  7E 00                  JLE     SMLSTK ;No, use what we have
0019  BB 1000                MOV     BX,4096 :Can only address 64K
001C             SMLSTK:     REPT    4
```

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

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

Symbol Table Format

The symbol table portion of a listing separates all "symbols" into their respective categories, showing appropriate descriptive data. This data gives you an idea how your program is using various symbolic values. Use this information to help you debug.

Also, you can use a cross reference listing, produced by MS-CREF, to help you locate uses of the various symbols in your program.

On the next few pages is a complete symbol table listing. Following this complete listing, sections from different symbol tables are shown with explanatory notes.

For all sections of symbol tables, this rule applies: if there are no symbolic values in your program for a particular category, the heading for the category will be omitted from the symbol table listing. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

```
Microsoft MACRO-86 MACRO
Assembler     date     PAGE     Symbols-1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXMP1M.ASM)

Macros:

                  Name      Length

BIOSCALL ............      0002
DISPLAY ..............     0005
DOSCALL .............      0002
KEYBOARD ...........       0003
LOCATE ..............      0003
SCROLL ...............     0004

Structures and records:

                  Name      Width       # fields
                            Shift       Width     Mask            Initial

PARMLIST ............      001C        0004
  BUFSIZE ...........      0000
  NAMESIZE ..........      0001
  NAMETEXT ..........      0002
  TERMINATOR .......       001B

Segments and groups:

                  Name      Size        align     combine         class

CSEG .................     0044        PARA      PUBLIC          'CODE'
STACK ................     0200        PARA      STACK           'STACK'
WORKAREA ..........        0031        PARA      PUBLIC          'DATA'
```

```
Symbols:

                    Name        type       Value    Attr

CLS ....................    N PROC     0036     CSEG         Length =000E
MAXCHAR ............        Number     0019
MESSG .................     L BYTE     001C     WORKAREA
PARMS .................     L 001C     0000     WORKAREA
RECEIVR ..............      L FAR      0000                  External
START .................     F PROC     0000     CSEG         Length =0036

Warning  Severe
Errors   Errors
0        0
```

NAMES OF MACROS

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

Example:

```
Macros:

                    Name        Length←—number of 32 byte blocks
                                        macro occupies
BIOSCALL ............       0002        in memory
DISPLAY ..............      0005
DOSCALL .............       0002
KEYBOARD ...........        0003
LOCATE ..............       0003
SCROLL ...............      0004
```

STRUCTURES AND RECORDS

This section of a symbol table 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 your record occupies.

# fields shows how many fields comprise your record.

For fields of structures:

Shift shows the number of bytes the fields is offset into the structure. The other columns are not used for fields of structures.

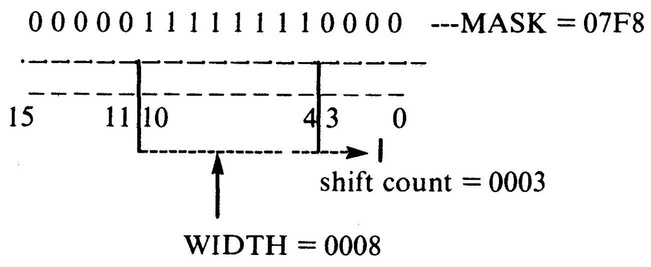
For fields of records:

Shift is the shift count to the right.

Width (lower line) shows the number of bits this field occupies.

Mask shows the maximum value of record, expressed in hexadecimal, if one field is masked and ANDed (field is set to all 1's and all other fields are set to all 0's).

Using field BZ1 of the record BAZ1 above to illustrate:



Initial shows the value specified as the initial value for the field, if any.

When naming the field, you specified:

fieldname:# = value

where: 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:

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

Example for structures:

```
                                  This line applies to structure names
                                             / (being in column 1)
                Name      Width      # fields<-/
                          Shift      Width     Mask     Initial<--This line
                                                                  for fields

PARMLIST ............     001C   |\   0004
of records                       | \    \
┌BUFSIZE ............     0000   |  \    \      (indented).
│ NAMESIZE ..........     0001   |\  \    \
│ NAMETEXT .........      0002   | \  \    \
└TERMINATOR .......       001B   |  \  \   Number of fields in
                                     \  \   structure
└ field names of          Offset of field \
  PARMLIST Structure       into structure  The number of bytes
                                            wide of structure
```

Example for records:

```
                         Name     Width     # fields
                                  Shift     Width      Mask     Initial<-This line
                                                                          for fields
                                                                          of records

BAZ . . . . . . . . . . . . .   --0008      0003 ------------------------number of
                                                                         fields in record
  FLD1 . . . . . . . . . . . .    0006      0002       00C0     0040
  FLD2 . . . . . . . . . . . .    0004      0003       0038     0000     initial value
  FLD3 . . . . . . . . . . . .    0000      0003       0007     0003
BAZ1 . . . . . . . . . . . . .  --000B      0002            \____________MASK of
                                                                         field
  BZ1  . . . . . . . . . . . .    0003      0008       07F8     0400     (maximum
                                                                         value)
  BZ2  . . . . . . . . . . . .    0000      0003       0007     0002

                        number of        shift      number of
                      bits in record     count     bits in field
                                        to right
```

SEGMENTS AND GROUPS

This section of a symbol table lists group and segment names and their attributes.

For groups:

The name of the group will appear under the name column, beginning in column 1 with the applicable segment names indented 2 spaces. The word group will appear under the size column.

For segments:

The segment names may appear in column 1 (as here) if you do not declare them part of a group. If you declare a group, the segment names will appear indented under their group name.

For all segments, whether a part of a group or not:

Size is the number of bytes the segment occupies.

Align is the type of boundary where the segment begins:

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

```
PARA   = paragraph - address is xxxx0H
         (low nibble = 0); default
WORD   = word - address is xxxxeH
         (e even number;
         low bit of low byte = 0)
       bit map -|x|x|x|x|x|x|x|0|
BYTE   = byte = address is xxxxxxH (anywhere)
```

Combine describes how MS-LINK will combine the various segments. (See Chapter 4 on MS-LINK for a full description.)

Class is the class name under which MS-LINK will combine segments in memory. (See Chapter 4 on MS-LINK for a full description.)

Segments and groups:

```
                     Name     Size     Align  Combine Class
                                                 ┌────called private
                                                ↙ for MS-LINK
AAAXQQ ..............       0000     WORD  NONE    'CODE'<── segment
DGROUP ..............       GROUP<────────────────────────── group
  DATA ..............       0024     WORD  PUBLIC  'DATA'    |
  STACK .............       0014     WORD  STACK   'STACK'   |segments
  CONST .............       0000     WORD  PUBLIC  'CONST'   ├of
  HEAP ..............       0000     WORD  PUBLIC  'MEMORY'  |DGROUP
  MEMORY ...........        0000     WORD  PUBLIC  'MEMORY'  |
ENTXCM ..............       0037     WORD  NONE    'CODE'
MAIN_STARTUP .......        007E     PARA  NONE    'MEMORY'
                            ────     ──────────────────────
                            length   statement line entries
                              of
                            segment
```

SYMBOLS

This section of a symbol table 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 ├◄ all defined by EQU or = directive
Opcode ┘

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

For all procedures, the length of the procedure is given after its attribute (segment).

You may also see an entry under type like:

```
L 0031
```

This entry results from code such as the following:

```
BAZ LABEL FOO
```

where FOO is a STRUC that is 31 bytes long.

BAZ will be shown in the symbol table with the L 0031 entry. Basically, Number (and some other similar entries) indicates that the symbol was defined by an EQU or = directive.

Value (usually) shows the numeric value the symbol represents. (In some cases, the Value column will show some text -- when the symbol was defined by EQU or = directive.)

Attr always shows the segment of the symbol, if known. Otherwise, the Attr column is blank. Following the segment name, the table will show either External, Global, or a blank (which means not declared with either the EXTRN or PUBLIC directive). The last entry applies to PROC types only. This is a length = entry, which is the length of the procedure.

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

Each of the four symbol 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, for example:

  FOO EQU ADD
- Alias shows a symbol name that the named symbol equals, for example:

  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, for example, the directive statements:

  GOO EQU 'WOW'
  BAZ EQU DS:8[BX]
  ZOO EQU 1.234

Examples:

```
Symbols:

                 Name  Type           Value        Attr

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

Symbols:

                 Name  Type           Value        Attr

BEGHQQ .............  L WORD         0012         DATA           Global
BEGOQQ .............  L FAR          0000           External
BEGXQQ .............  F PROC         0000         MAIN_STARTUP Global Length =0006E
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                Global
ENTGQQ ............. L FAR     0000                External
FREXQQ ............. F PROC    006E   MAIN_STARTUP Global Length =0010
HDRFQQ ............. L WORD    0006   DATA         Global
HDRVQQ ............. L WORD    0008   DATA         Global
HEAPBEG ............ BYTE      0000   STACK                 EQU stateme
HEAPLOW ............ BYTE      0000   HEAP                  showing segn
INIUQQ ............. L FAR     0000                External
PNUXQQ ............. L WORD    0004   DATA         Global
RECEQQ ............. L WORD    0010   DATA         Global
REFEQQ ............. L WORD    000C   DATA         Global
REPEQQ ............. L WORD    000E   DATA         Global
RESEQQ ............. L WORD    000A   DATA         Global
SKTOP .............. BYTE      0014   STACK
SMLSTK ............. L NEAR    001C   MAIN_STARTUP
STARTMAIN .......... F PROC    0000   ENTXCM       Length =001E
STKBQQ ............. L WORD    0018   DATA         Global
STKHQQ ............. L WORD    001A   DATA         Global
```

If MACRO-86 knows this length as one of the type lengths (BYTE, WORD, DWORD, QWORD, TBYTE), it shows that type name here.

MACRO-86 MESSAGES

Most of the messages generated by MACRO-86 are error messages. The nonerror messages from MACRO-86 are the banner MACRO-86 displays when first invoked, the command prompt messages, and the successful end of assembly message. These nonerror messages are classified here as operating messages. The error messages are categorized as either assembler errors, I/O handler errors, or runtime errors.

Operating Messages

The MACRO-86 banner message and command prompts appear on the APC screen as follows:

```
MACRO-86 v1.0 Copyright (C) Microsoft, Inc.
Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing [NUL.LST]:
Cross reference [NUL.CRF]:
```

The end of assembly message looks like this:

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

(The MS-DOS system prompt)

Error Messages

If MACRO-86 encounters errors, error messages are output, along with the numbers of warning and fatal errors, and control is returned to MS-DOS. The message is output either to your APC screen or to the listing file if you command one be created.

Error messages are divided into three categories: assembler error, I/O handler error, and runtime error messages. The assembler and I/O handler messages are listed in tables in numerical order.

Table 3-4 MACRO-86 Error Messages

ERROR MESSAGE	ERROR CODE	MEANING
Block nesting error	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.

Table 3-4 MACRO-86 Error Messages (Cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Extra characters on line	1	This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.
Register already defined	2	This will only occur if the assembler has internal logic errors.
Unknown symbol type	3	Symbol statement has something in the type field that is unrecognizable.
Redefinition of symbol	4	This error occurs on pass 2 and succeeding definitions of a symbol.
Symbol is multi-defined	5	This error occurs on a symbol that is later redefined.
Phase error between passes	6	The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes (see the section on switches in this chapter).
Already had ELSE clause	7	Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF... ENDIF).

Table 3-4 MACRO-86 Error Messages (Cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Not in conditional block	8	An ENDIF or ELSE is specified without a previous conditional assembly directive action.
Symbol not defined	9	A symbol is used that has no definition.
Syntax error	10	The syntax of the statement does not match any recognizable syntax.
Type illegal in context	11	The type specified in of an unacceptable size.
Must be declared in pass 1	13	Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.
Symbol type usage illegal	14	Illegal use of a PUBLIC symbol.
Symbol already different kind	15	Attempt to define a symbol differently from a previous definition.
Symbol is reserved word	16	Attempt to use an assembler reserved word illegally. (For example, to declare MOV as a variable.)
Forward reference is illegal	17	Attempt to forward reference something that must be defined in pass 1.
Must be register	18	Register unexpected as operand but user furnished symbol -- was not a register.

Table 3-4 MACRO-86 Error Messages (Cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Wrong type of register	19	Directive or instruction expected one type of register, but another was specified. For example, INC CS.
Must be segment or group	20	Expecting segment or group and something else was specified.
Symbol has no segment	21	Trying to use a variable with SEG, and the variable has no known segment.
Already defined locally	23	Tried to define a symbol as EXTERNAL that had already been defined locally.
Segment parameters are changed	24	List of arguments to SEGMENT were not identical to the first time this segment was used.
Not proper align/combine type	25	SEGMENT parameters are incorrect.
Reference to multi-defined	26	The instruction references something that has been multi-defined.
Operand was expected	27	Assembler is expecting an operand but an operator was received.
Operator was expected	28	Assembler was expecting an operator but an operand was received.
Division by 0 or overflow	29	An expression is given that results in a divide by 0.
Shift count is negative	30	A shift expression is generated that results in a negative shift count.

Table 3-4 MACRO-86 Error Messages (Cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Operand types must match	31	Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.
Illegal use of External	32	Use of an external in some illegal manner. For example, BD M DUP(?) where M is declared external.
Must be record field name	33	Expecting a record field name but got something else.
Must be record or field name	34	Expecting a record or field name and received something else.
Operand must have size	35	Expected an operand to have a size, but it did not.
Left operand must have segment	38	Used something in the right operand that required a segment in the left operand. (For example, ":.")
One operand must be const	39	This is an illegal use of the addition operator.
Operands must be same or 1 abs	40	Illegal use of the subtraction operator.
Normal type operand expected	41	Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.
Constant was expected	42	Expecting a constant and received something else.
Operand must have segment	43	Illegal use of SEG directive.

Table 3-4 MACRO-86 Error Messages (Cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Must be associated with data	44	Use of code related item where a data-related item was expected. For example, MOV AX, code-label.
Must be associated with code	45	Use of data related item code item was expected.
Already have base register	46	Trying to double base register.
Already have index register	47	Trying to double index address.
Must be index or base register	48	Instruction requires a base or index register and some other register was specified in square brackets, [ ].
Illegal use of register	49	Use of a register with an instruction where there is no 8086 or 8088 instruction possible.
Value is out of range	50	Value is too large for expected use. For example, MOV AL,5000.
Operand not in IP segment	51	Access of an operand is impossible because it is not in the current IP segment.
Improper operand type	52	Use of an operand such that the opcode cannot be generated.
Relative jump out of range	53	Relative jumps must be within the range -128 to +127 of the current instruction, and the specific jump is beyond this range.
Index displ. must be constant	54	

Table 3-4 MACRO-86 Error Messages (cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Illegal register value	55	The register value specified does not fit into the "reg" field (the reg field is greater than 7).
Illegal size for item	57	Size of referenced item is illegal. For example, shift of a double word.
Byte register is illegal	58	Use of one of the byte registers in context where it is illegal. For example, PUSH AL.
CS register illegal usage	59	Trying to use the CS register illegally. For example, XCHG CS, AX.
Must be AX or AL	60	Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.
Improper use of segment reg	61	Specification of a segment register where this is illegal. For example, an immediate move to a segment register.
No or unreachable CS	62	Trying to jump to a label that is unreachable.
Operand combination illegal	63	Specification of a two-operand instruction where the combination specified is illegal.
Label can't have seg. override	65	Illegal use of segment override.
Can't override ES segment	67	Trying to override the ES segment is an instruction where this override is not legal. For example, store string.

Table 3-4 MACRO-86 Error Messages (cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Can't reach with segment reg	68	There is no assume value that makes the variable reachable.
Must be in segment block	69	Attempt to generate code when not in a segment.
Can't use EVEN on BYTE segment	70	Segment was declared to be byte segment and attempt to use EVEN was made.
Forward needs override	71	This message not currently used.
Illegal value for DUP count	72	DUP counts must be a constant that is not 0 or negative
Symbol already external	73	Attempt to define a symbol as local that is already external.
DUP is too large for linker	74	Nesting of DUP's was such that too large a record was created for MS-LINK.
Usage of ? (indeterminate) bad	75	Improper use of the "?". For example, ?+5.
More values than defined with	76	Too many fields given in the REC or STRUC allocation.
Only initialize list legal	77	Attempt to use STRUC name without angle brackets.
Directive illegal in STRUC	78	All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the DEFINE directives.

Table 3-4 MACRO-86 Error Messages (cont'd)

ERROR MESSAGE	ERROR CODE	MEANING
Override with DUP is illegal	79	In a STRUC initialization statement, you tried to use DUP in an override.
Field cannot be overridden	80	In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.
Override is of wrong type	81	In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.
Register can't be forward ref	82	
Circular chain of EQU aliases	83	An alias EQU eventually points to itself.
Should have been group name		Expecting a group name but something other than this was given.

I/O Handler Errors

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

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

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

The error-message-text is one of the messages in Table 3-5. This table lists the I/O handler error messages in code number order.

Table 3-5 I/O Handler Error Messages

ERROR CODE	ERROR MESSAGE
114	Data format
108	Device full
102	Device name
105	Device offline
112	File in use
107	File name
110	File not found
113	File not open
104	File system
101	Hard data
115	Line too long
106	Lost file
103	Operation
111	Protected file
109	Unknown device

Runtime Errors

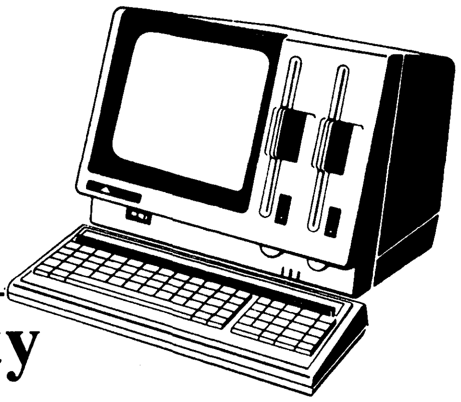
These messages may be displayed as your assembled program is being executed.

Internal Error — Usually caused by an arithmetic check. If it occurs, notify your APC service representative.

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

Chapter 4

The MS-Link Linker Utility



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

MS-LINK is user-friendly. It prompts you for all the necessary and optional commands. Your answers to the prompts are the commands for MS-LINK.

The output file from MS-LINK (run file) is not bound to specific memory addresses and, therefore, can be loaded and executed at any convenient address by your MS-DOS operating system.

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

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

SYSTEM REQUIREMENTS FOR RUNNING MS-LINK

MS-LINK requires 49K bytes of memory minimum: 40K bytes for code and data, and 10K bytes for run space.

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

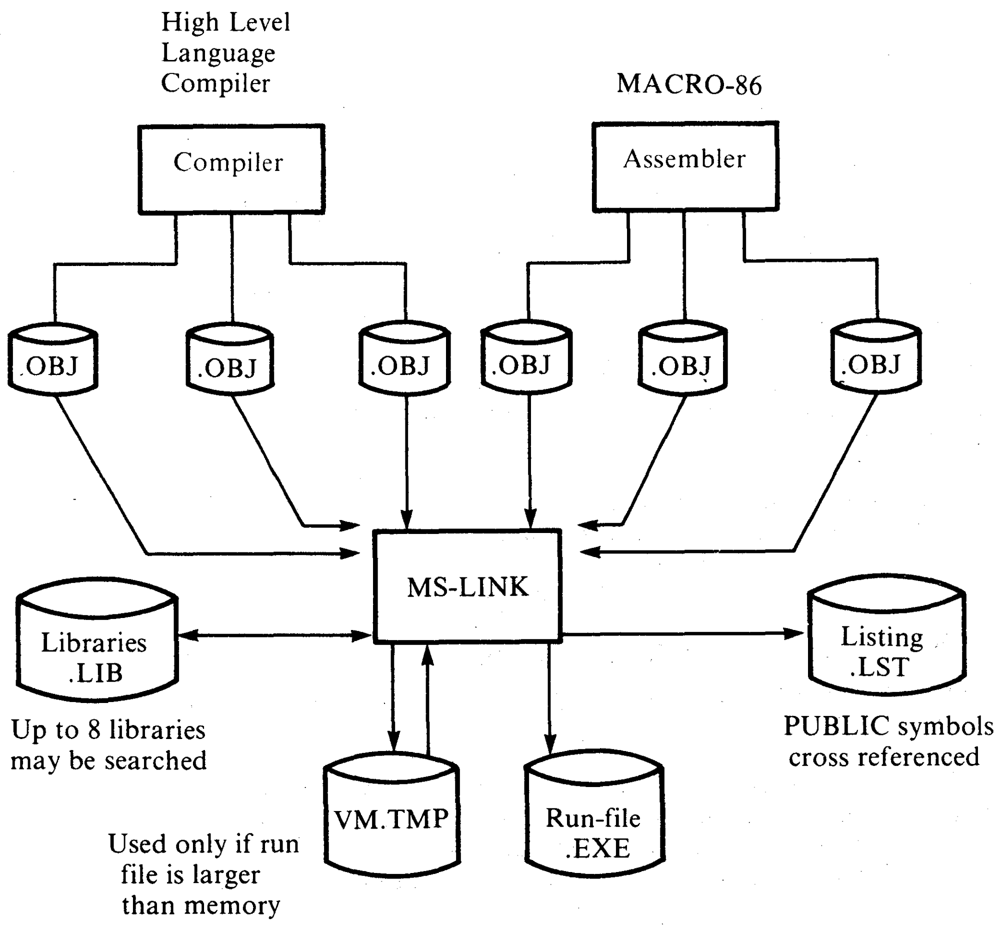
OVERVIEW OF MS-LINK OPERATIONS

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

As it combines modules, MS-LINK resolves external references between object modules. It also searches multiple library files for definitions of any external references left unresolved.

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

MS-LINK uses available memory as much as possible.When available memory is exhausted, MS-LINK then creates a disk file and becomes a virtual linker. The following illustrates the MS-LINK's operations.


High Level Language Compiler
MACRO-86
Compiler
Assembler
.OBJ
.OBJ
.OBJ
.OBJ
.OBJ
.OBJ
MS-LINK
Libraries .LIB
Listing .LST
Up to 8 libraries may be searched
PUBLIC symbols cross referenced
VM.TMP
Run-file .EXE
Used only if run file is larger than memory

How MS-DOS Divides Programs into Executable Portions

When programs are executed, MS-DOS places portions of the linked object code in memory according to their size and designated order of execution. Object code is divided into segments and groups of segments. Segments are assigned to classes for placement in memory at execution time.

SEGMENT

A segment is a contiguous area of memory up to 64K bytes in length. A segment may be located anywhere in 8086 memory on a "paragraph" (16 byte) boundary. The contents of a segment are addressed by a segment-register/offset pair.

GROUP

A group is a collection of segments that fit within 64K bytes of memory. The segments are named to the group by the MACRO-86, 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 a segment register plus an offset. MS-LINK checks to see that the object modules of a group meet the 64K byte constraint.

CLASS

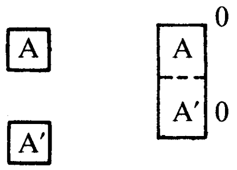
A class is a collection of segments. The naming of segments to a class controls the order and relative placement of segments in memory. The class name is given by you in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler. The segments are named to a class at compile or assembly time. The segments of a class are loaded into memory contiguously. The segments are ordered within a class in the order MS-LINK encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to MS-LINK. Classes may be loaded across 64K byte boundaries. The classes will be divided into groups for addressing.

How MS-LINK Combines and Arranges Program Segments

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

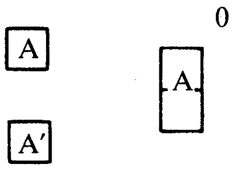
MS-LINK combines segments for these combine types as follows:

Private



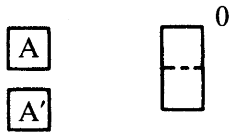
Private segments are loaded separately and remain separate. They may be physically but not logically, contiguous even if the segments have the same name. Each private segment has its own base address.

Public



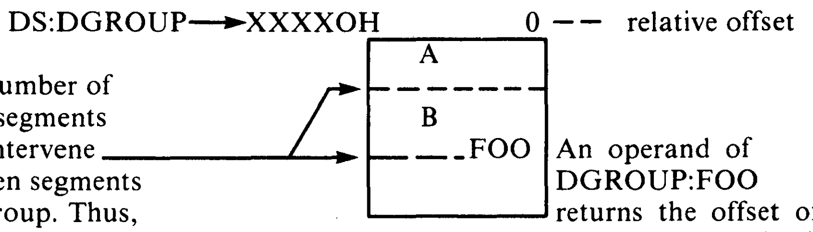
Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine types stack and memory are treated the same as public.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

Common



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Placing segments in a group for MACRO-86 provides offset addressing of items from a single base address for all segments in that group.



Any number of other segments may intervene between segments of a group. Thus, the offset of FOO may be greater than the size of the segments in a group combined, but no larger than 64K.

An operand of DGROUP:FOO returns the offset of FOO from the beginning of the first segment DGROUP (segment A here)

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

If your program contains:	They will be loaded as:
A SEGMENT 'FOO'	'FOO'
B SEGMENT 'BAZ'	A
C SEGMENT 'BAS'	E
D SEGMENT 'ZOO'	'BAS'
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 MS-LINK Object Modules prompt. The dummy module declares segments into classes in the order you want the classes loaded.

CAUTION

Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs.Allow the compiler and the linker to perform their tasks in the normal way.

Example

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

You should be careful to declare all classes to be used in your program in this module. If you do not, you lose absolute control over the ordering of classes.

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

Files That MS-LINK Uses

MS-LINK works with one or more input files, produces two output files, may create a virtual memory file, and may be directed to search one to eight library files. For each type of file, you may give a three part file specification. The format for MS-LINK file specifications is the same as that of a disk file:

```
[d:] <filename> [ <.ext> ]
```

where: d: is the drive designation. Permissible drive designations for MS-LINK are A: through O: The colon is always required as part of the drive designation.

<filename> is any legal filename of one to eight characters.

<.ext> is a one- to three-character extension to the filename. The period is always required as part of the extension.

INPUT FILE EXTENSIONS

If no extensions are given in the input (object) file specifications, MS-LINK recognizes by default:

File	Default Extension
Object	.OBJ
Library	.LIB

OUTPUT FILE EXTENSIONS

MS-LINK appends to the output (run and list) files the following default extensions:

File	Default Extension	
Run	.EXE	(may not be overridden)
List	.MAP	(may be overridden)

VM.TMP FILE

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

```
VM.TMP has been created.
Do not change disk 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 MS-LINK is unpredictable, and MS-LINK might return the error message

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

MS-LINK uses VM.TMP as a virtual memory. The contents of VM.TMP are subsequently written to the file named for the Run File prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

CAUTION

Do not use VM.TMP as a filename for any file. If you have a file named VM.TMP on the default drive and MS-LINK requires the VM.TMP file, MS-LINK will delete the VM.TMP on disk and create a new VM.TMP. Thus, the contents of the previous VM.TMP file will be lost.

RUNNING MS-LINK

Running MS-LINK requires a command to invoke MS-LINK and answers to command prompts. In addition, six switches control alternative MS-LINK features. Usually, you will enter all the commands to MS-LINK at the APC keyboard. As an option, answers to the command prompt and any switches can be contained in a "response file." Some special command characters are provided to assist you in entering linker commands.

MS-LINK can be involved in three ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the line used to invoke MS-LINK. By the third method, you create a response file that contains all the necessary commands.

Method 1: LINK

Enter

```
    LINK
```

MS-LINK is to be loaded into memory. Then, it returns a series of four text prompts for your commands to MS-LINK. At the end of each command line, you may enter one or more switches, each of which must be preceded by a slash mark. If a switch is not included, MS-LINK defaults to not performing the function for the switch.

MS-LINK COMMAND PROMPTS

MS-LINK prompts you for the names of object, run, and list files, and for libraries that you want to search.

Table 4-1 lists the MS-LINK command prompts.

Table 4-1 MS-LINK Command Prompts

PROMPT	RESPONSE
Object Modules [.OBJ]:	Enter a list of the object modules to be linked. MS-LINK assumes by default that the filename extension is .OBJ. If an object module has any other filename extension, the extension must be given here. Otherwise, the extension may be omitted. Modules must be separated by plus signs (+). Remember that MS-LINK loads segments into classes in the order in which the object modules are encountered. Use this information for setting the order in which the object modules are entered.
Run File [First-Object-filename.EXE]:	The file name entered here will be created to store the run (executable) file that results from the link session. All run files receive the filename extension .EXE, even if you specify another extension. The user-specified extension is ignored. If no response is entered to the Run File prompt, MS-LINK uses the first filename entered in response to the Object Modules prompt as the RUN file name. *Example:* Run File [FUN.EXE]: B:PAYROLL/P This response directs MS-LINK to create the run file PAYROLL.EXE on drive B:. Also, MS-LINK will pause, which allows you to insert a new disk to receive the run file.

Table 4-1 MS-LINK Command Prompts (cont'd)

PROMPT	RESPONSE
List File [Run-filename.MAP]:	The list file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the run file. The default response is the run filename with the default file name extension .MAP.
Libraries [ ]:	The valid responses are one to eight library filenames or simply a RETURN. A carriage return means no library search. Library files must have been created using a library utility. MS-LINK assumes by default that the filename extension is .LIB for library files. Library filenames must be separated by blank spaces or plus signs (+). MS-LINK searches the library files in the order listed to resolve external references. When it finds the module that defines the external symbol, MS-LINK processes the module as another object module. If MS-LINK cannot find a library file on the 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).

Table 4-1 MS-LINK Command Prompts (cont'd)

PROMPT	RESPONSE
	MS-LINK does not search within each library file sequentially. MS-LINK uses a method called the dictionary-indexed library search. This means that MS-LINK finds definitions for external references by index access rather than searching from the beginning of the file to the end for each reference. This indexed search reduces substantially the link time for any sessions involving library searches.

MS-LINK COMMAND SWITCHES

Six switches control alternate linker functions: /DSALLOCATE, /HIGH, /LINENUMBERS, /MAP, /PAUSE, and /STACK. These switches must be entered at the end of a prompt response, regardless of which method is used to invoke MS-LINK. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is entered at the end of one response, each switch must be preceded by the slash mark (/).

All switches may be abbreviated, from a single letter through the whole switch name. The only restriction is that an abbreviation must be a sequential sub-string from the first letter through the last entered. No gaps or transpositions are allowed, for example:

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

The MS-LINK command switches are summarized in Table 4-2.

Table 4-2 MS-LINK Command Switches

SWITCH	FUNCTION
/DSALLOCATE	When the /DSALLOCATE switch is used, MS-LINK loads all data (DGroup) at the high end of the data segment. Otherwise, MS-LINK loads all data at the low end of the data segment. At runtime, the DS pointer is set to the lowest possible address and allows the entire DS segment to be used. Use of the /DSALLOCATE switch in combination with the default load low (that is, the /HIGH switch is not used), permits the user application to allocate dynamically any variable 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.

Table 4-2 MS-LINK Command Switches (cont'd)

SWITCH	FUNTION
	NOTE The application program may dynamically allocate up to 64K bytes or the actual amount available less the amount allocated within DGroup.
/HIGH	When the /HIGH switch is used, MS-LINK places the Run image as high as possible in memory. Otherwise, MS-LINK places the run file as low as possible. CAUTION Do not use the /HIGH switch with Pascal or FORTRAN programs.
/LINENUMBERS	When the /LINENUMBERS switch is used, MS-LINK includes in the list file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the list file. NOTE Not all compilers produce object modules that contain line number information. In these cases, of course, MS-LINK cannot include line numbers.
/MAP	The /MAP switch directs MS-LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, MS-LINK will list only errors (which includes undefined globals). The symbols are listed alphabetically. For each symbol, MS-LINK lists its value and its segment: the offset location in the run file. The symbols are listed at the end of the list file.

Table 4-2 MS-LINK Command Switches (cont'd)

SWITCH	FUNTION
/PAUSE	The /PAUSE switch directs MS-LINK to pause in the link session when the switch is encountered. Normally, MS-LINK performs the linking session without stop from beginning to end. This allows you to swap the disks before MS-LINK outputs the run (.EXE) file. When MS-LINK encounters the /PAUSE switch, it displays the message: About to generate .EXE file Change disks hit any key MS-LINK resumes processing when you press any key. CAUTION Do not swap the disk that will receive the List file, or the disk used for the VM.TMP file, if created.
/STACK: number	The number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If the /STACK switch is not specified for a link session, MS-LINK calculates the necessary stack size automatically. If a value from 1 to 511 is entered, MS-LINK uses 512. All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size. At least one object (input) module must contain a stack allocation statement. If not, MS-LINK will return a WARNING: NO STACK STATEMENT error message.

MS-LINK COMMAND CHARACTERS

MS-LINK provides three command characters.

+ Use the plus sign (+) to separate entries and to extend the current physical line following the Object Modules and Libraries prompts. (A blank space may be used to separate object modules.) To enter a large number of responses (each which may also be very long), enter a plus sign/carriage return at the end of the physical line (to extend the logical line). If the plus sign/carriage return is the last entry following these two prompts, MS-LINK will prompt the user for more modules names. When the Object Modules or Libraries prompt appears again, continue to enter responses. When all the modules to be linked have been listed, be sure the response line ends with a module name and a carriage return and not a plus sign/carriage return.

Example:

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

Use a single semicolon (;) followed immediately by RETURN at any time after the first prompt (from Run File on) to select default responses to the remaining prompts. This feature saves time and overrides the need to enter a series of carriage returns.

NOTE

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

Example:

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

The remaining prompts will not appear, and MS-LINK will use the default values (including FUN.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 MS-LINK then reinvoke MS-LINK and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

Method 2: LINK <filenames>[/switches)

Enter

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

The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

where: <object list> is a list of object modules, separated by plus signs

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

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

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

/switch are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <lib-list>, as shown).

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

Example:

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

This example first causes MS-LINK to be loaded, followed by the object module FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. MS-LINK then pauses (caused by the /P switch). When you press any key, MS-LINK links the object modules, produces a global symbol map (the /M switch), defaults to the 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 MS-LINK prompts (shown under method 1 for invoking the linker), and may also contain any of the switches. Method 3 may also contain any of the switches. It permits you to conduct the MS-LINK session without interactive (direct) user responses to the MS-LINK prompts.

NOTE

Before using method 3 to invoke MS-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 on the screen.

Use switches and command characters in the response file the same way as they are used for responses entered at the APC keyboard.

When the MS-LINK session begins, each prompt will be displayed in turn with the responses from the response file. If the response file does not contain answers for all the prompts, either in the form of filenames or the semicolon special character or carriage returns, MS-LINK will, after displaying the prompt which does not have a response, wait for you to enter a legal response. When a legal response has been entered, MS-LINK continues the link session.

Example:

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

This response file will cause MS-LINK to load the four object modules named. It will pause before creating and producing a public symbol map to permit you to swap diskettes. When you press any key, the output files will be named FUN.EXE and FUNLIST.MAP. MS-LINK will search the library file COBLIB.LIB and will use the default settings for the flags.

EXAMPLE OF A MS-LINK SESSION

This example shows you the type of information that is displayed during an MS-LINK session.

In response to the MS-DOS prompt, enter

```
LINK
```

The system displays the MS-LINK banner and then the command prompts. Then you make your responses to the prompts.

```
Microsoft Object Linker V.2.00
(C) Copyright 1982 by Microsoft Inc.

Object Modules [.OBJ]: IBMBIO SYSINIT
Run File [IBMBIO.EXE]:
List File [NUL.MAP]: IBMBIO -MAP
Libraries [.LIB]: ;
```

Some options you have in entering responses are the following.

- By specifying -MAP for the List File prompt, you get both an alphabetic listing and a chronological listing of public symbols.
- By responding PRN to the List File prompt, you can redirect your output to the printer.
- By specifying the -LINE switch, MS-LINK gives you a listing of all line numbers for all modules. (Note that the -LINE switch can generate a large volume of output.)
- By pressing RETURN in response to the Libraries prompt, an automatic library search is performed.

Once MS-LINK locates all libraries, the linker map displays a list of segments in the order of their appearance within the load module. The list might look like this:

```
Start      Stop       Length     Name
00000H     009ECH     09EDH      CODE
009F0H     01166H     0777H      SYSINITSEG
```

The information in the Start and Stop columns shows the 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module.

The addresses displayed are not the absolute addresses where these segments are loaded. Consult the section EXECUTABLE FILE STORING AND LOADING for information on how to determine where relative zero is actually located and also, on how to determine the absolute address of a segment.

When the -MAP switch is used, MS-LINK displays the public symbols by name and value. For example:

```
ADDRESS        PUBLICS_BY_NAME
009F:0012      BUFFERS
009F:0005      CURRENT_DOS_LOCATION
009F:0011      DEFAULT_DRIVE
009F:000B      DEVICE_LIST
009F:0013      FILES
009F:0009      FINAL_DOS_LOCATION
009F:000F      MEMORY_SIZE
009F:0000      SYSINIT

ADDRESS        PUBLICS BY VALUE
009F:0000      SYSINIT
009F:0005      CURRENT_DOS_LOCATION
009F:0009      FINAL_DOS_LOCATION
009F:000B      DEVICE_LIST
009F:000F      MEMORY_SIZE
009F:0011      DEFAULT_DRIVE
009F:0012      BUFFERS
009F:0013      FILES
```

EXECUTABLE FILE STRUCTURE AND LOADING

MS-LINK procedures produce executable modules in the form of .EXE files. These .EXE files consist of two parts: control and relocation information, and the load module.

The control and relocation information, which is described below, is at the beginning of the file in an area known as the header. The load module immediately follows the header. The load module begins on a sector boundary and is the memory image of the module constructed by the linker.

The header is formatted as follows:

Hex Offset	Contents
00-01	4DH, 5AH - This is the MS-LINK program's signature to mark the file as a valid .EXE file.
02-03	Length of image mod 512 (remainder after dividing the load module image size by 512).
04-05	Size of the file in 512-byte increments (pages), including the header.
06-07	Number of relocation table items that follow the formatted position of the header.
08-09	Size of the header in 16-byte increments (paragraphs). This is used to locate the beginning of the load module in the file.
0A-0B	Minimum number of 16-byte paragraphs required above the end of the loaded program.
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program.
0E-0F	Offset of stack segment in load module (in segment form).
10-11	Value to be given in the SP register when the module is given control.
12-13	Word checksum — negative sum of all the words in the file, ignoring overflow.
14-15	Value to be given in the IP register when the module is given control.
16-17	Offset of code segment within load module (in segment form).
18-19	Offset of the first relocation item within the file.
1A-1B	Overlay number (0 for resident part of the program).

The relocation table follows the formatted area just described. The relocation table is made up of a variable number of relocation items. The number of items is contained at offset 06-07. The relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word that requires modification before the module is given control. This process is called relocation and is accomplished in the following manner:

1. A Program Segment Prefix is built following the resident portion of the program that is performing the load operation.
2. The formatted part of the header is read into memory (its size is at offset 08-09).
3. The load module size is determined by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Note that all files created by pre-release 1.10 MS-LINK programs always placed a value of 4 at that location, regardless of actual program size. Therefore, this field should be ignored if it contains a value of 4. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.
4. The load module is read into memory beginning with the start segment.
5. The relocation table items are read into a work area (one or more at a time).
6. Each relocation table item segment value is added to the start segment value. This calculated segment, in conjunction with the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.
7. Once all relocation items have been processed, the SS and SP registers are set from the values in the header and the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is used to give the module control.

MS-DOS PROGRAM SEGMENTATION

When you enter an external command or invoke a program through the EXEC function call, MS-DOS determines the lowest available address to use as the start of available memory for the program being invoked. This area is called the Program Segment. It must not be moved.

At offset 0 within the Program Segment, MS-DOS builds the Program Segment Prefix control block (see Figure 4-2). EXEC loads the program at offset 100H and gives it control.

The program returns from EXEC by one of four methods:

- a long jump to offset 0 in the Program Segment Prefix
- issuing an INT 20H
- issuing an INT 21H with register AH=0 with CS:0 pointing at the PSP, or 4CH and no restrictions on CS
- calling location 50H in the Program Segment Prefix with AH=0 or 4CH

NOTE

You must make sure that the CS register contains the segment address of the Program Segment Prefix for a program when terminating via any of these methods, except for a call to Function 4CH. For this reason, a call to function 4CH is preferred to using AH=0.

All four methods result in transferring control to the program that issued the EXEC. During this returning process, interrupt vectors 22H, 23H, and 24H (Terminate Address, CTRL-C Exit Address, and Fatal Error Abort Address) addresses are restored from the values and saved in the Program Segment Prefix of the terminating program. Control is then given to Terminate Address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued. Otherwise, COMMAND.COM performs a checksum on its transient part, reloads that if necessary, then issues the system prompt and waits for the next command to be entered from the keyboard.

When a program receives control, certain conditions are in effect.

Environment Information for .EXE and .COM Programs

The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.

The environment is a series of ASCII strings (totaling less than 32K) in the form:

```
NAME=parameter
```

Each string is terminated by a byte of zeros, and the entire set of strings is terminated by another byte of zeros. The environment built by the Command Processor (and passed to all programs it invokes) will contain a COMSPEC= string at a minimum (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings entered through the MS-DOS SET command.

The environment that you are passed is actually a copy of the invoking process environment. If your application uses a "terminate and stay resident" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent SET, PATH, or PROMPT commands are issued.

Offset 50H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. Thus, by placing the desired function number in AH, a program can issue a long call to PSP +50 to invoke an MS-DOS function, rather than issuing an interrupt type 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.

Disk transfer address (DTA) is set to 80H (default DTA in the Program Segment Prefix).

File control blocks at 5CH and 6CH are formatted from the first two parameters entered when the command was invoked. If either parameter contained a pathname, then the corresponding FCB will contain only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters entered after the command name (including leading and imbedded delimiters), with 80H set to the number of characters. If the >, or parameters were entered on the command line, they (and the filenames associated with them) will not appear in this area, because redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX reflects the validity of drive specifiers entered with the first two parameters as follows:

- AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)
- AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

Environment Information for .EXE Programs Only

DS and ES registers are set to point to the Program Segment Prefix.

CS,IP,SS, and SP registers are set to the values passed by MS-LINK.

Environment Information for .COM Programs Only

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

All of user memory is allocated to the program. If the program invokes another program through the EXEC function call, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being invoked.

The Instruction Pointer (IP) is set to 100H.

The SP register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

A word of zeros is placed on top of the stack.

The Program Segment Prefix (with offsets in hexadecimal) is formatted as shown in Figure 4-1.

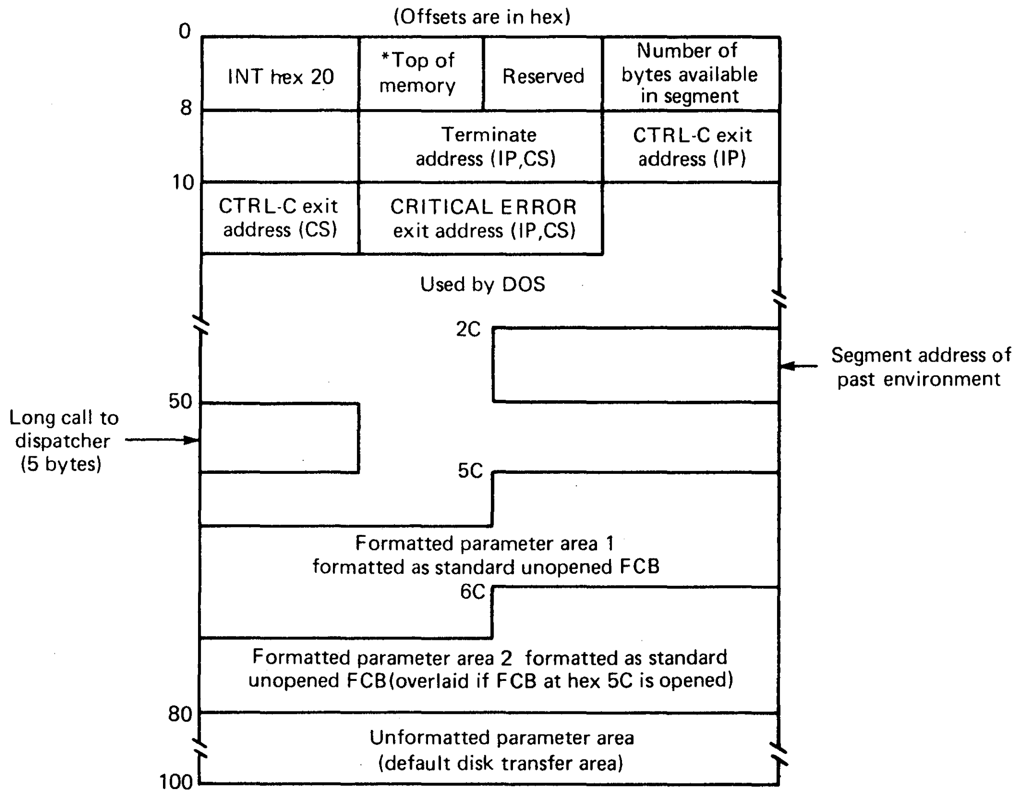

(Offsets are in hex)
0
INT hex 20
*Top of memory
Reserved
Number of bytes available in segment
8
Terminate address (IP,CS)
CTRL-C exit address (IP)
10
CTRL-C exit address (CS)
CRITICAL ERROR exit address (IP,CS)
Used by DOS
2C
Segment address of past environment
50
Long call to dispatcher (5 bytes)
5C
Formatted parameter area 1 formatted as standard unopened FCB
6C
Formatted parameter area 2 formatted as standard unopened FCB(overlaid if FCB at hex 5C is opened)
80
Unformatted parameter area (default disk transfer area)
100

Figure 4-1 Program Segment Prefix

Notes:
*First segment of available memory is in segment (paragraph) form (for example, 100H would represent 64K).
**The word at offset 6 contains the number of bytes available in the segment.
***Offset 2CH contains the segment address of the environment.

CAUTION

Programs must not alter any part of the PSP below offset 5CH.

MS-LINK MESSAGES

Most of the messages generated by MS-LINK are error messages. These messages are described in Table 4-3. Note that all errors cause the link session to abort. Therefore, after the cause is found and corrected, MS-LINK must be rerun.

MS-LINK also produces I/O handler errors and runtime errors.

Table 4-3 MS-LINK Error Messages

ERROR MESSAGE	MEANING
Attempt to access data outside of segment bonds, possibly bad object module	Probably a bad object file.
Bad numeric parameter	Numeric value not in digits.
Cannot open temporary file	MS-LINK is unable to create the file VM.TMP because the disk directory is full. Designate a new disk. Do not change the disk that will receive the "list.MAP" file.
Error: Dup record too complex	DUP record in the assembly language module is too complex. Simplify the DUP record.
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 a bad object file.
Invalid object module	Object module(s) incorrectly formed or incomplete (as when assembly was stopped).
Symbol defined more than once	MS-LINK found two or more modules that define a single symbol name.

Table 4-3 MS-LINK Error Messages (cont'd)

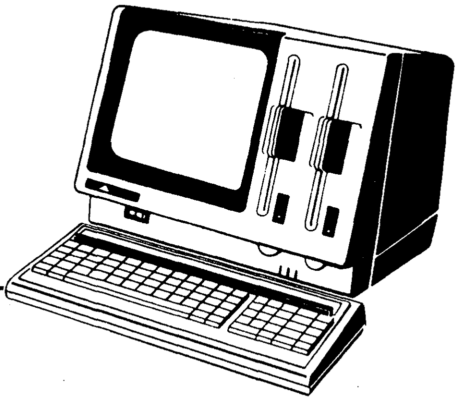
ERROR MESSAGE	MEANING
Program size or number of segments exceeds capacity of linker	The total size may not exceed 384K bytes and the number of segments may not exceed 255.
Request stacked size exceed 64K	Specify a size of 64K bytes with the /STACK switch.
Segment size exceeds 64K	64K bytes is the addressing system limit.
Symbol table capacity exceeded	Very many, very long names entered exceeding approximately 25K bytes.
Too many external symbols in one module	The limit is 256 external symbols per module.
Too many groups	The limit is 10 groups.
Too many libraries specified	The limit is 8 libraries.
Too many public symbols	The limit is 1024 public symbols.
Too many segments or classes	The limit is 256 segments and classes taken together.
Unresolved externals: list	The external symbols listed have no defining module among the modules or library files specified.
VM read error:	A disk problem; not MS-LINK caused.
Warning: No stack segment	None of the object modules specified contains a statement allocating stack space, but you entered the /STACK switch.
Warning: Segment of absolute or unknown type	A bad object module or an attempt to link modules MS-LINK cannot handle (for example, an absolute object module).

Table 4-3 MS-LINK Error Messages (cont'd)

ERROR MESSAGE	MEANING
Write error in tmp file	No more disk space remaining to expand the VM.TMP file.
Write error on run file	Usually, not enough disk space for the run file.

Chapter 5

The MS-LIB Library Manager



The MS-LIB Library Manager creates and modifies library files that are used with MS-LINK. MS-LIB can add object files to a library and delete modules from a library. It can extract modules from a library and place the extracted modules into separate object files.

With MS-LIB you can create a library for several object files or for one program only. Placing just one program module in a library can make for very fast linking and possibly more efficient execution.

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

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

SYSTEM REQUIREMENTS FOR RUNNING MS-LIB

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

For disk storage space, it requires 1 disk drive if and only if output is sent to the same physical disk from which the input was taken. There is no time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration to use MS-LIB on.

OVERVIEW OF MS-LIB OPERATIONS

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

- deleting a module
- extracting a module to place it in a separate object file
- appending an object file as a module of a library
- replacing a module in the library file with a new version of that module
- creating a library file.

During a library session, MS-LIB first deletes or extracts modules, then appends new ones. In a single operation, MS-LIB reads each module into memory, checks it for consistency of format, and writes it back to the file. If you delete a module, MS-LIB reads in that module but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places the module at the end of the last module written. This procedure effectively "closes up" the disk space to keep the library file from growing larger than necessary. When MS-LIB has read through the whole library file, it appends any new modules to the end of the file. Finally, MS-LIB creates the index, which MS-LINK uses to find modules and symbols in the library file, 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.

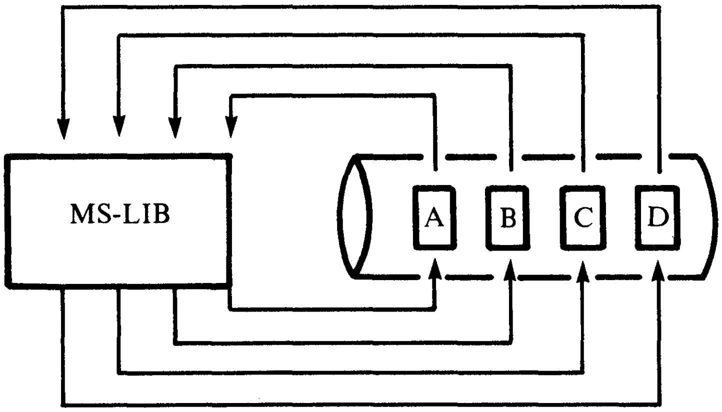
Example:

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

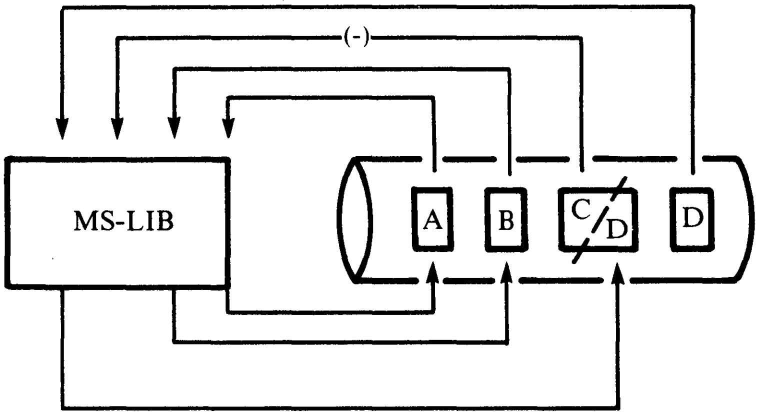
first deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. This order of execution prevents confusion in MS-LIB when a new version of a module replaces an older version in the library file. Note that the replace function is simply the delete-append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

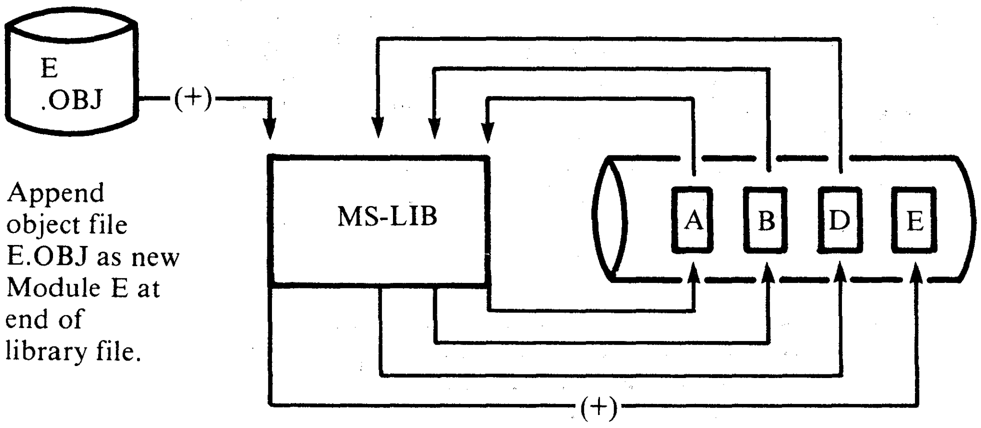
The following illustrates the manipulation of member modules under the several functions of MS-LIB:

Consistency
check only.

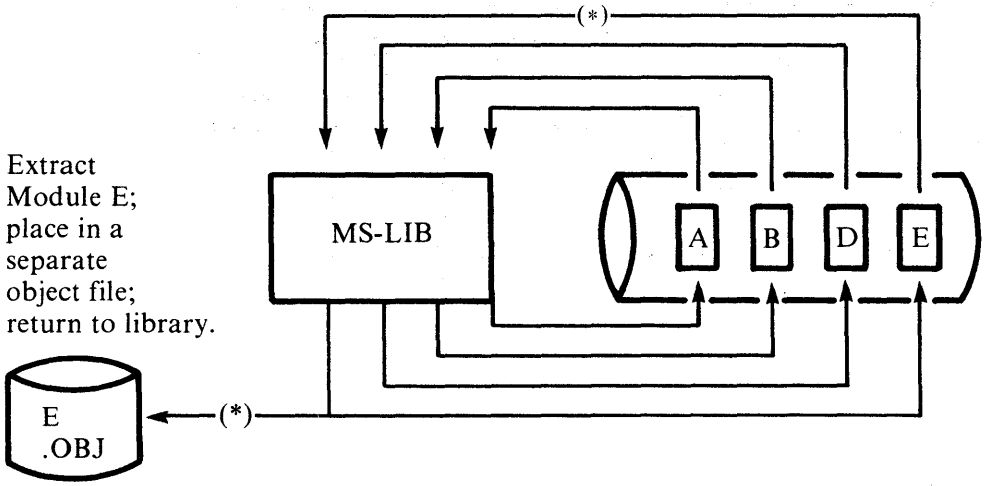


Delete
Module C;
Module D
written to
space of
Module C.



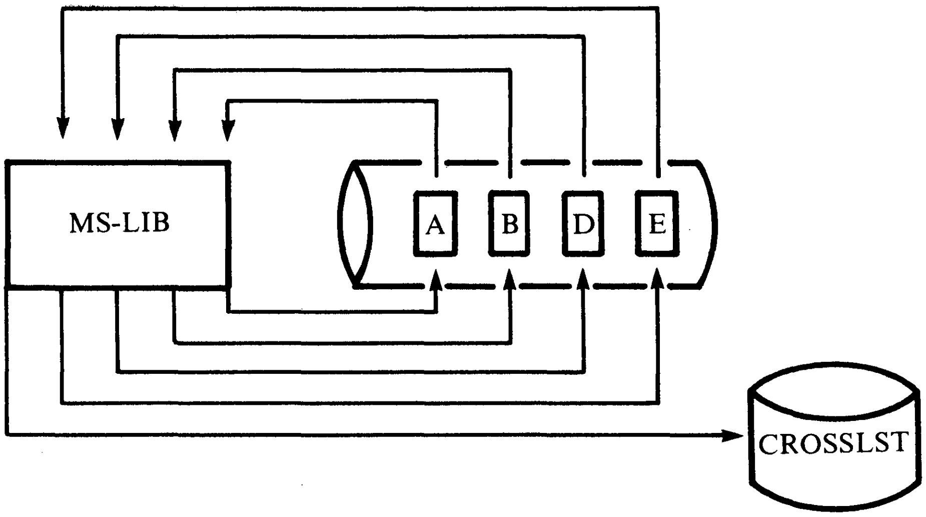


Append object file E.OBJ as new Module E at end of library file.



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

Do a consistency Check, then output a cross reference listing of PUBLIC symbols.



RUNNING MS-LIB

Running MS-LIB requires invoking MS-LIB and answering command prompts. Usually you will enter all the commands to MS-LIB at the APC keyboard. Optionally, answers to the command prompts may be contained in a "response file." Some special command characters exist. Some are used as a required part of MS-LIB commands. Others assist you while entering MS-LIB commands.

MS-LIB may be invoked three ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the command line used to invoke MS-LIB. By the third method, you create a response file that contains all the necessary commands.

Method 1: LIB

Enter

```
LIB
```

MS-LIB is loaded into memory, then, returns a series of three text prompts.

MS-LIB COMMAND PROMPTS

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

Table 5-1 describes the MS-LIB command prompts.

Table 5-1 MS-LIB Command Prompts

PROMPT	RESPONSE
Library file:	Enter the name of the library file that you want to manipulate. MS-LIB assumes that the filename extension is .LIB. You can override this assumption by giving a filename extension when you enter the library filename. Because MS-LIB can manage only one library file at a time, only one filename is allowed in response to this prompt. Additional reponses, except the semicolon command character, are ignored. If you enter a library filename and follow it immediately with a semicolon command character, MS-LIB will perform a consistency check only, then return to the operating system. Any errors in the file will be reported. If the filename you enter does not exist, MS-LIB returns the prompt: Library file does not exist. Create? You must enter either Yes or No, in either upper or lower (or mixed) case. Actually, MS-LIB checks the response for the letter Y as the first character. If any other character is entered first, MS-LIB terminates and control returns to the operating system.
Operation:	Enter one of the three command characters for manipulating modules (+, -, *), followed immediately (no space) by the module name or the object filename. The plus sign appends an object file as the last module in the library file (see further discussion under the description of the plus sign below). The minus sign deletes a module from the library file. The asterisk extracts a module from the library and places it in a separate object file with the filename taken from the module name and a filename extension .OBJ.

Table 5-1 MS-LIB Command Prompts (cont'd)

PROMPT	RESPONSE
	When you have a large number of modules to manipulate (more than can be typed on one line), enter an ampersand (&) as the last character on the line. MS-LIB will repeat the Operation prompt, which permits you to enter additional module names and object filenames. MS-LIB allows you to enter operations on modules and object files in any order you want. More information about order of execution and what MS-LIB does with each module is given in the descriptions of each command character.
List file:	If you want a cross reference first of the PUBLIC symbols in the modules in the library file after your manipulations, enter the filename of the file in which you want MS-LIB to place the cross reference listing. If you do not enter a filename, no cross reference listing is generated (a NUL file). The response to the List file prompt is a file specification. Therefore, you can specify, along with a 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.

MS-LIB COMMAND CHARACTERS

MS-LIB provides six command characters:

- Three of the command characters are required in responses to the Operation prompt
- The other three command characters provide additional commands.

Table 5-2 describes the MS-LIB command characters.

Table 5-2 MS-LIB Command Characters

COMMAND CHARACTER	DESCRIPTION
+	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 MS-LIB sees the plus sign, it assumes that the filename extension is .OBJ. You may override this assumption by specifying a different filename extension. MS-LIB strips the drive designation and the extension from the object file specification, leaving only the filename. For example, if the object file to be appended as a module to a library is B:CURSOR.OBJ a response to the Operation prompt of +B:CURSOR.OBJ causes MS-LIB to strip off the B: and the .OBJ, leaving only CURSOR, which becomes a module named CURSOR in the library. NOTE The distinction between an object file and a module (or object module) is that the file possesses a drive designation (even if it is the default drive) and a filename extension. Object modules possess neither of these.

Table 5-2 MS-LIB Command Characters (cont'd)

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

*	The asterisk followed by a module name extracts that module from the library file and places it into a separate object file. The module will still exist in the library (extract means, essentially, copy the module to a separate object file). The module name is used as the filename. MS-LIB adds the default drive designation and the filename extension .OBJ. For example, if the module to be extracted is
	
	`CURSOR`
	
	and the current default disk drive is A:, a response to the Operation prompt of
	
	`*CURSOR`
	
	causes MS-LIB to extract the module named CURSOR from the library file and to set it up as an object file with the file specification of
	
	`default drive:CURSOR.OBJ`
	
	(The drive designation and filename extension cannot be overridden. You can, however, rename the file, giving a new filename extension, and/or copy the file to a new disk drive, giving a new filename and/or filename extension.)

;	Use a single semicolon (;) followed immediately by RETURN any time after responding to the first prompt (from Library file on) to select default repsonses to the remaining prompts. This feature saves time and eliminates answering additional prompts.

Table 5-2 MS-LIB Command Characters (cont'd)

COMMAND CHARACTER	DESCRIPTION
	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, press RETURN. *Example:* Library file: FUN RETURN Operation: +CURSOR; RETURN The remaining prompts will not appear, and MS-LIB will use the default value (no cross reference file).
&	Use the ampersand to extend the current physical line. This command character will only be needed for the Operation prompt. MS-LIB can perform many functions during a single library session. The number of modules you can append is limited only by disk space. The number of modules you can replace or extract is also limited only by disk space. The number of modules you can delete is limited only by the number of modules in the library file. However, the line length for a response to any prompt is limited to the command line length. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. MS-LIB will display the Operation prompt again, then enter more responses. You may use the ampersand character as many times as you need. *Example:* Library file: FUN RETURN Operation: +CURSOR-HEAP+HEAP*FOIBLES& Operation: *INIT+ASSUME+RIDE; RETURN

Table 5-2 MS-LIB Command Characters (cont'd)

COMMAND CHARACTER	DESCRIPTION
	MS-LIB will delete the module HEAP, extract the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then append the object files CURSOR, HEAP, ASSUME, and RIDE. Note, however, that MS-LIB allows you to enter 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 a module name, or an incorrectly spelled filename or module name), you must press CRTL-C to exit MS-LIB then reinvoke MS-LIB and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

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

Enter

```
LIB <library><operations>,<listing>
```

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

The entry fields are described as follows:

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

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

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

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

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

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

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

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

Example:

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

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

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

Example:

```
LIB PASCAL RETURN
```

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

Example:

```
LIB PASCAL,PASCROSS.PUB
```

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

Method 3: LIB @ filespec

Enter

```
LIB @ <filespec>
```

where <filespec> is the name of a response file. A response file contains answers to the MS-LIB prompts (summarized under method 1 for invoking). Method 3 permits you to conduct the MS-LIB session without interactive (direct) response to the MS-LIB command prompts.

NOTE

Before using method 3 to invoke MS-LIB, you must first create the required response file.

A response file has text lines, one for each prompt. Responses must appear in the same order as the command prompts appear on the screen.

Use command characters in the response file the same way as they are used for responses entered from the APC keyboard.

When the library session begins, each prompt will be displayed in turn with the responses from the response file. If the response file does not contain answers for all the prompts, MS-LIB will use the default responses (no changes to the modules currently in the library file for operation, and no cross reference listing file created).

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

If you enter a library filename only press RETURN for operations, then type a comma and a list filename, MS-LIB will perform its consistency check of the library file, and produce the cross reference listing file.

Example:

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

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

MS-LIB ERROR MESSAGES

MS-LIB generates the error messages listed in Table 5-3.

Table 5-3 MS-LIB Error Messages

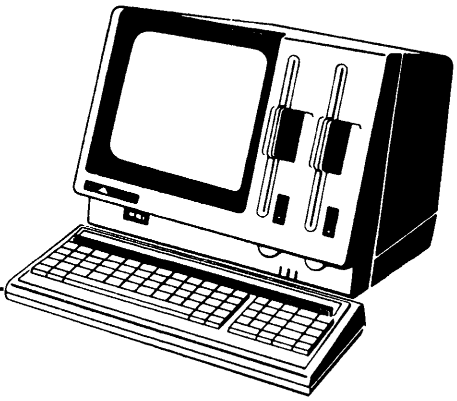
ERROR MESSAGE	MEANING
Symbol is a multiply-defined public. Proceed?	Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol. A No response leaves the library in an undetermined state. Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.
Allocate error on VM.TMP	Out of space.
Cannot create extract file	No room in directory for the extract file.
Cannot create list file	"@filespec" in response (or indirect) file
Cannot open VM.TMP	No room for VM.TMP in disk directory.
Cannot write library file	Out of space.
Close error on extract file	Out of space.
Error: An internal error has occurred	Contact your APC service representative

Table 5-3 MS-LIB Error Messages (cont'd)

ERROR MESSAGE	MEANING
Fatal error: Cannot open input file	Mistyped object file name.
Fatal error: Module is not in the library	Trying to delete a module that is not in the library.
Input file read error	Bad object module or faulty disk.
Invalid object module/library	Bad object and/or library.
Library disk is full	Out of space.
No library file specified	No response to Library file prompt.
Read error on VM.TMP	Disk not ready for read.
Symbol table capacity exceeded	Too many public symbols (about 30K characters in symbols).
Too many object modules	More than 500 object modules.
Too many public symbols	1024 public symbols maximum.
Write error on library/extract file	Out of space.
Write error on VM.TMP	Out of space.

Chapter 6

The MS-CREF Cross Reference Utility



The MS-CREF Cross Reference Utility can aid you in debugging your assembly language programs. MS-CREF produces an alphabetical listing of all the symbols in a special file produced by MACRO-86. With this listing, you can quickly locate all occurrences of any symbol in your source program by line number.

The MS-CREF produced listing is meant to be used with the symbol table produced by MACRO-86.

The MACRO-86 symbol table listing shows the value of each symbol, its type, and its length. This information is needed to correct erroneous symbol definitions or uses.

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

SYSTEM REQUIREMENTS FOR RUNNING MS-CREF

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

The peripheral devices needed are

- One disk drive if and only if output is sent to the same physical disk from which the input was taken. There is no time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration for using MS-CREF.
- A printer for printed output.

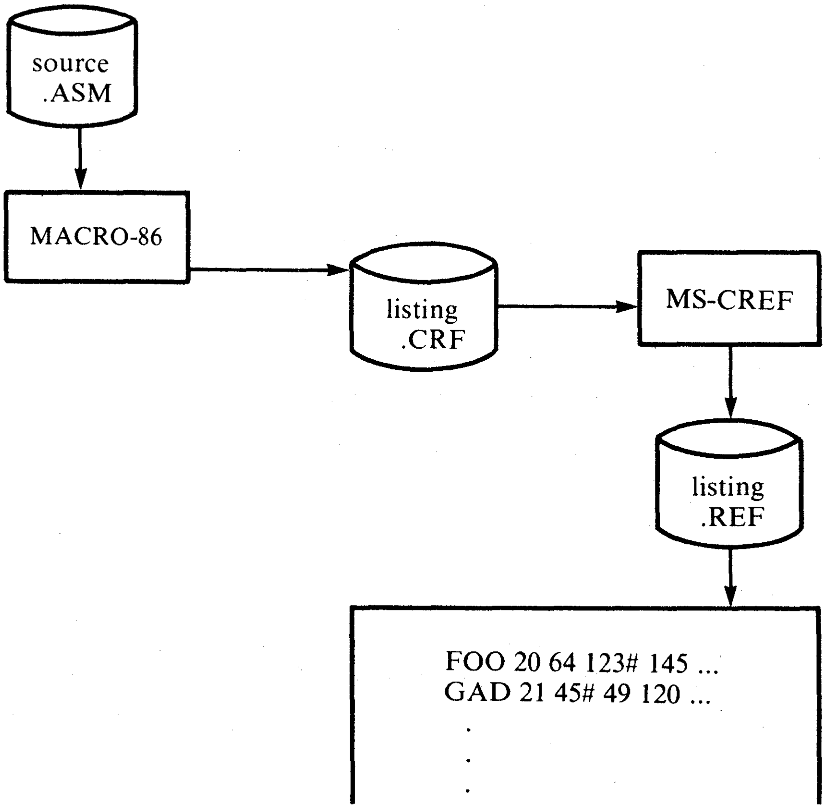
OVERVIEW OF MS-CREF OPERATIONS

MS-CREF produces a file with cross references for the symbolic names in your program.

First, you must create a cross reference file with MACRO-86. Then, MS-CREF takes this cross reference file, which has the filename extension .CRF, and turns it into an alphabetical listing of the symbols in the file. The cross reference listing file is given the default filename extension .REF.

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

The following illustrates MS-CREF's operations:



RUNNING MS-CREF

Running MS-CREF requires invoking MS-CREF and answering command prompts. You will enter all the commands to MS-CREF at the APC keyboard. Some special command characters exist to assist you in entering MS-CREF commands.

Before you can use MS-CREF to create the cross reference listing, you must first have created a cross reference file using MACRO-86.

Creating a Cross Reference File

A cross reference file is created during an assembly session. To create this file, you answer the fourth MACRO-86 command prompt with the name of the file you want to receive the cross reference file.

The fourth assembler prompt is

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

If you do not enter a filename in response to this prompt, or if you in any other way use the default response to this prompt, the assembler will not create a cross reference file. Therefore, you must enter a filename. You may also specify which drive you want to receive the file and what filename extension you want the file to have, if different from .CRF. If you change the filename extension from .CRF to anything else, you must remember to specify the filename extension when naming the file in response to the first MS-CREF command prompt.

When you have given a filename in response to the fourth MACRO-86 prompt, the cross reference file will be generated during the assembly session.

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

Invoking MS-CREF

MS-CREF may be invoked two ways. By the first method, you enter the commands as answers to individual prompts. By the second method, you enter all commands on the command line used to invoke MS-CREF.

METHOD 1: CREF

Enter

```
CREF
```

MS-CREF is loaded into memory, then displays two prompts.

MS-CREF Command Prompts

MS-CREF prompts you for the names of the cross reference file for conversion and the cross reference listing file to be produced.

Table 6-1 describes the MS-CREF command prompts.

Table 6-1 MS-CREF Command Prompts

PROMPT	RESPONSE
Cross reference [.CRF]:	Enter the name of the cross reference file you want MS-CREF to convert into a cross reference listing. The name of the file is the name you gave to MACRO-86 when you directed it to produce the cross reference file. MS-CREF assumes that the filename extension is .CRF. If you do not specify a filename extension when you enter the cross reference filename, MS-CREF will look for a file with the name you specify and the filename extension .CRF. If your cross reference file has a different extension, specify that extension when entering the filename. See the section FORMAT OF MS-CREF COMPATIBLE FILES for a description of what MS-CREF expects to see in the cross reference file. You will need this information only if your cross reference file was not produced by a Microsoft assembler.

Table 6-1 MS-CREF Command Prompts (cont'd)

PROMPT	RESPONSE
Listing [crffile.REF]:	Enter the name you want the cross reference listing file to have. MS-CREF will automatically give the cross reference listing the filename extension .REF. If you want your cross reference listing to have the same filename as the cross reference file but with the filename extension .REF, simply press RETURN when the Listing prompt appears. if you want your cross reference listing file to be named anything else and/or to have any other filename extension, you must enter a response following the Listing prompt. If you want the listing file placed on a drive or device other than the default drive, specify the drive or device when entering your response to the Listing prompt.

MS-CREF Command Characters

Two command characters can be used in entering MS-CREF commands. An override command character allows you to select default reponses to the MS-CREF command prompts. CTRL-C aborts MS-CREF execution.

Table 6-2 describes the MS-CREF command characters.

Table 6-2 MS-CREF Command Characters

COMMAND CHARACTER	DESCRIPTION
;	Use a single semicolon (;) followed immediately by a RETURN at any time after responding to the Cross reference prompt to select the default response for the Listing prompt. This feature saves time and overrides the need to answer the Listing prompt. If you use the semicolon, MS-CREF gives the listing file the filename of the cross reference file and the default filename extension .REF. *Example:* Cross reference.[.CRF]: FUN; MS-CREF will process the cross reference file named FUN.CRF and output a listing file named FUN.REF.
CTRL-C	Use CTRL-C at any time to abort the MS-CREF session. If you enter an erroneous response (the wrong filename, or an incorrectly spelled filename), you must press CTRL-C to exit MS-CREF, then reinvoke MS-CREF and start over. If the error has been typed but not entered, you may delete the erroneous characters, but for that line only.

METHOD 2: CREF <CRFFILE>, <LISTING>

Enter

```
CREF <crffile>,<listing>
```

MS-CREF will be loaded into memory. Then, MS-CREF immediately procedes to convert your cross reference file into a cross reference listing.

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

The entry fields are described as follows:

<crffile> is the name of a cross reference file produced by MACRO-86. MS-CREF assumes that the filename extension is .CRF, which you may override by specifying a different extension. If the file named for the crffile does not exist, MS-CREF will display the message:

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

Control then returns to MS-DOS.

<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 semi-colon after you enter the crffile name.

Example:

```
CREF FUN; RETURN
```

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

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

```
CREF FUN,B:WORK.ARG
```

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

FORMAT OF CROSS REFERENCE LISTINGS

An example of a cross reference listing produced by MS-CREF is provided in the next few pages.

Each page is headed with the title of the program or program module.

Then comes the alphabetical list of symbols. Following each symbol name is a list of the line numbers where the symbols occur in the program. The line number for the definition has a pound sign (#) appended to it.

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

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

  Symbol Cross Reference         (# is definition)

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

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

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

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

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

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

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

INIUQQ ..............    31   161

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

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

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

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

MS-CREF ERROR MESSAGES

All errors cause MS-CREF to abort. Control is returned to MS-DOS.

Error messages are displayed in the format:

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

where: *filename* is the name of the file where the error occurs

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

Table 6-3 lists the error messages for MS-CREF in the numerical order of their code numbers.

Table 6-3 MS-CREF Error Messages

ERROR CODE	MEANING
101	Hard data error. Unrecoverable disk I/O error.
101	Device name error. Illegal device specification (for example, X:F00.CRF).
103	Internal error. Report to your APC service representative.
104	Internal error. Report to your APC service representative.
105	Device offline. Disk drive door open, no printer attached, and so on.
106	Internal error. Report to your APC service representative.
108	Disk full.
110	Fatal I/O error. File not found.
111	Disk is write-protected.
112	Internal error. Report to your APC service representative.
113	Internal error. Report to your APC service representative.
114	Internal error. Report to your APC service representative.
115	Internal error. Report to your APC service representative.

FORMAT OF MS-CREF COMPATIBLE FILES

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

General Description of MS-CREF File Processing

In essence, MS-CREF reads a stream of bytes from the cross reference input file, sorts them, then emits them as a printable listing file (the .REF file). The symbols are held in memory as a sorted tree. References to the symbols are held in a linked list.

MS-CREF keeps track of line numbers in the source file by the number of end-of-line characters it encounters. Therefore, every line in the source file must contain at least an end-of-line character (see Table 6-4).

MS-CREF attempts to place a heading at the top of every page passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol (see Table 6-4). If MS-CREF encounters more than one title symbol in the source file, it uses the last title read for all page headings. If MS-CREF does not encounter a title symbol in the file, the title line on the listing is left blank.

Format of Source Files

MS-CREF uses the first three bytes of the source file as format specification data. The rest of the file is processed as a series of records that either begin or end with a byte identifing the type of record.

FIRST THREE BYTES

The first byte of the source file contains the number of lines to be printed per page (page length range is from 1 to 255 characters).

The second byte indicates the number of characters per line (line length range is from 1 to 132 characters).

The third byte is the page symbol (07) that tells MS-CREF that the two preceding bytes define listing page size.

The PAGE directive in your assembler, that takes arguments for page length and line length, will pass the above information to the cross reference file.

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

RECORD CONTROL SYMBOLS

The information given below shows the types of records that MS-CREF recognizes and the byte values and placement it uses to recognize record types.

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

Records that begin with a control symbol:

Byte Value	Control Symbol	Subsequent Bytes
01	Reference symbol	Record is a reference to a symbol name (1 to 80 characters)
02	Define symbol	Record is a definition of a symbol name (1 to 80 characters)
04	End of line	(none)
05	End of file	1AH

Records that end with a control symbol:

Byte Value	Control Symbol	Preceding Bytes
06	Title defined	Record is title text (1 to 80 characters)
07	Page length/ line length	One byte for page length followed by one byte for line length

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

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

The record control symbols are defined in Table 6-4.

Table 6-4 Record Control Symbols

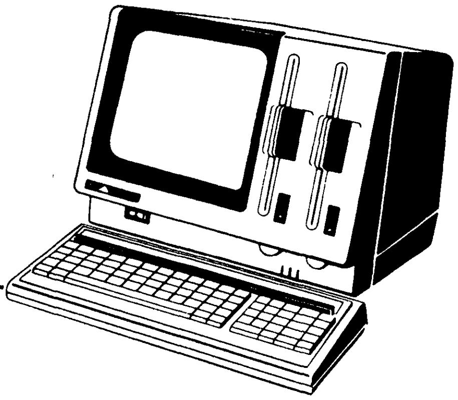
CONTROL SYMBOL	DEFINITION
Reference symbol	The record contains the name of a symbol that is referenced. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.
Define symbol	The record contains the name of a symbol that is defined. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.
End of line	The record is an end of line symbol character only (04H or Control-D).
End of file	The record is the end of file character (1AH).
Title defined	ASCII characters of the title to be printed at the top of each listing page. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used for the title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the listing is left blank.
Page length/line length	The first byte of the record contains the number of lines to be printed per page (range is from 1 to 255 lines). The second byte contains the number of characters to be printed per page (range is from 1 to 132 characters). The default page length is 58 lines. The default line length is 80 characters.

The .CRF file record contents are the following:

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

Chapter 7

The DEBUG Program



The Debugging Program (DEBUG) is used to examine and make corrections to binary and executable object files. As the Line Editor (EDLIN) is used to alter source files, DEBUG modifies binary files.

With DEBUG, you do not need to reassemble or recompile a program to see if a problem has been fixed by a minor change. You can alter the contents of a file or a CPU register, and then immediately reexecute a program to check the validity of the changes.

The DEBUG commands may be aborted at any time by pressing CTRL-C. CTRL-S suspends the display, so that you can read it before the output scrolls away. Entering any key other than CTRL-C or CTRL-S restarts the display. All of these commands are consistent with the control character functions available at the MS-DOS command level.

INVOKING DEBUG

To invoke DEBUG, enter

```
    DEBUG [<filespec> [<arglist>] ]
```

For example, if a <filespec> is specified, then the following is a typical invocation:

```
    DEBUG FILE.EXE
```

DEBUG then loads FILE.EXE into memory starting at 100 hexadecimal in the lowest available segment. The BX:CX registers are loaded with the number of bytes placed into memory. Then, DEBUG returns its command prompt. The DEBUG prompt is a right angle bracket (>).

An <arglist> may be specified if <filespec>is present. These are filename parameters and switches that are to be passed to the program <filespec>. Thus, when <filespec> is loaded into memory, it is loaded as if it had been invoked with the command:

```
    <filespec>    <arglist>
```

Here, <filespec> is the file to be debugged, and <arglist> is the rest of the command line that is used when <filespec> is invoked and loaded into memory.

If no <filespec> is specified, then DEBUG is invoked as follows:

```
    DEBUG
```

DEBUG then returns with the prompt, signaling that it is ready to accept your commands. Since no filename has been specified, current memory, disk sectors, or disk files can be worked on by invoking later commands.

The DEBUG Commands

Each DEBUG command consists of a single letter followed by one or more parameters. Additionally, the control character and the dual-mode (PF) key functions that work on the command line all apply inside DEBUG. (See the *MS-DOS System User's Guide* for a description of the functions mentioned above.)

If a syntax error occurs in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow (^) and the word error.

For example:

```
    dcs: 100 cs:110
         ^ error
```

All commands and parameters may be entered in either upper or lower case. Any combination of upper and lower case may be used in commands.

The DEBUG commands are summarized in Table 7-1 and are described in detail with examples in the section **DEBUG COMMAND DESCRIPTIONS.**

Table 7-1 DEBUG Commands

COMMAND	FUNCTION
A [<address>]	Assemble
C<range> <address>	Compare
D[<range>]	Dump
E<address> [<list>]	Enter
F<range> [<list>]	Fill
G[=<address> [<address> ...]]	Go
H<address> <address>	Hex
I<value>	Input
L[<address> [<drive> <record> <record>]]	Load
M<range> <address>	Move
N<filespec>	Name
O<value> <byte>	Output
Q	Quit
R[<register-name>]	Register
S<range> <list>	Search
T[=<address>] [<value>]	Trace
U[<range>]	Unassemble
W[<address> [<drive> <record> <record>]]	Write

DEBUG Command Parameters

As Table 7-1 shows, all DEBUG commands accept parameters, except the Quit command. Parameters may be separated by delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

Also, entries may be made in any combination of upper or lower case letters.

Table 7-2 lists the DEBUG command parameters, describing their legal values.

Table 7-2 DEBUG Command Parameters

PARAMETER	DEFINITION
<drive>	A one-digit hexadecimal value to indicate which drive a file will be loaded from or written to. The legal values are 0-7. These values designate the drives as follows: 0=A (FD1), 1=A (FD2D), 2=B (FD1).
<byte>	A two-digit hexadecimal value to be placed in or read from an address or register.
<record>	A one- to three-digit hexadecimal value used to indicate the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors. However, their numbering differs, since they represent the entire disk space.
<value>	A hexadecimal value of up to four digits used to specify a port number or the number of times a command should repeat its functions.

Table 7-2 DEBUG Command Parameters (cont'd)

PARAMETER	DEFINITION
<address>	A two-part designation consisting of either an alphabetic segment register designation or a four-digit segment address plus an offset value. The segment designation or segment address may be omitted, in which case the default segment is used. DS is the default segment for all commands except G. L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal. For example: CS:0100 04BA:0100 The colon is required between a segment designation (whether numeric or alphabetic) and an offset.
<range>	<range> can be expressed as: ● two addresses; <address> <address> ● one address, an L, and a value; <address>L<value> (where <value> is the number of lines the command should operate on) ● simply <address>, where L80 is assumed. The last form cannot be used if another hex value follows the <range>, since the hex value would be interpreted as the second address of the <range>. Examples: C5:100 110 CS:100 L 10 CS:100 The following is illegal: CS/100 CS:110 ^ Error The limit for <range> is 10000 hex. To specify a <value> of 10000 hex within four digits, enter 0000 (or 0).

Table 7-2 DEBUG Command Parameters (cont'd)

PARAMETER	DEFINITION
<list>	A series of <byte> values or of <string>s.line. <list> must be the last parameter on the command line. Example: [fcs:100 42 45 52 54 41
<string>	Any number of characters enclosed in quote marks. Quote marks may be either single (') or double ("). Within <string>s, the opposite set of quote marks may be used freely as literals. If the delimiter quote marks must appear within a <string>, the quote marks must be doubled. For example, the following strings are legal. 'This is a "string" is okay.' 'This is a ' 'string' ' is okay.' However, this string is illegal. 'This is a 'string' is not.' Similarly, these strings are legal. "This is a 'string' is okay." "This is a " "string" " is okay." However, this string is illegal. "This is a "string" is not."
	Note that the double quotations are not necessary in the following strings: "This is a ' 'string' ' is not necessary." 'This is a " "string" " is not necessary.' The ASCII values of the characters in the string are used as a <list> of byte values.

DEBUG COMMAND DESCRIPTIONS

Descriptions of the DEBUG commands listed in Table 7-2 are given in the following pages.

A(ssemble) Command

Format:

```
A[<address>]
```

Function:

Assembles 8086/8087/8088 mnemonics directly into memory.

Remarks:

All numeric input to the Assemble command is in hexadecimal. The assembly statements you enter are assembled into memory at successive locations, starting with the address specified in <address>. If no <address> is specified, the statements are assembled into the area at CS:0100 if no previous Assemble command was used, or into the location following the last instruction assembled by a previous Assemble command. When all desired statements have been entered, press RETURN at the prompt for the next statement, to return to the DEBUG prompt.

If a syntax error is encountered, DEBUG responds with

```
^ Error
```

and redisplays the current assembly address.

Observe the following rules when using the Assemble command.

- All numeric values are hexadecimal and may be entered as one to four characters.
- Prefix mnemonics must be entered in front of the opcode to which they refer. They may also be entered on a separate line.
- The segment override mnemonics are CS:, DS:, ES:, and SS:.
- String manipulation mnemonics must explicitly state the string size. For example, the MOVSW must be used to move word strings and MOVSB must be used to move byte strings.
- The mnemonic for the far return is RETF.

- The assembler will automatically assemble short, near, or far jumps and calls depending on byte displacement to the destination address. These may be overridden with the NEAR or FAR prefix. For example:

  ```
  0100:0550  JMP    502        ; a 2 byte short jump
  0100:0502  JMP    NEAR 505; a 3 byte near jump
  0100:0505  JMP    FAR 50A  ; a 5 byte far jump
  ```

- The NEAR prefix can be abbreviated to NE, but the FAR prefix cannot be abbreviated.

DEBUG cannot tell whether some operands refer to a word memory location or a byte memory location. In this case, the data type must be explicitly stated with the prefix "WORD PTR" or "BYTE PTR." DEBUG will also accept the abbreviations "WO" or "BY." For example:

```
NEG                         BYTE PTR [128]
DEC                         WO[SI]
```

DEBUG also cannot tell whether an operand refers to a memory location or to an immediate operand. DEBUG uses the common convention that operands enclosed in square brackets refer to memory. For example:

```
MOV   AX,21                 ;Load AX with 21H
MOV   AX,[21]               ;Load AX with the contents
                            ; of memory location 21H
```

Two popular pseudo-instructions have also been included. The DB opcode will assemble byte values directly into memory. The DW opcode will assemble word values directly into memory. For example:

```
DB    1,2,3,4, "THIS IS AN EXAMPLE"
DB    'THIS IS A QUOTE:" '
DB    "THIS IS A QUOTE:' "

DW     1000,2000,3000,"BACH"
```

All forms of the register indirect commands are supported. For example:

```
ADD                         BX,34[BP+2].[SI-1]
POP                         [BP+DI]
PUSH                        [SI]
```

All opcode synonyms are supported. For example:

```
LOOPZ          100
LOOPE          100

JA             200
JNBE           200
```

For 8087 opcodes, the WAIT or FWAIT prefix must be explicitly specified. For example:

```
FWAIT FADD ST,ST(3)      ;This line will assemble
                         ;a FWAIT prefix

FLD TBYTE PTR [BX]       ;This line will not
```

Example:

```
C>debug
—a200
08B4:0200 xor ax,ax
08B4:0202 mov [bx],ax
08B4:0204 ret
08B4:0205
```

C(ompare) Command

Format:

```
C<range> <address>
```

Function:

Compares the portion of memory specified by <range> to a portion of the same size beginning at <address>.

Remarks:

If the two areas of memory are identical, there is no display and DEBUG returns with the DEBUG prompt. If there are differences, they are displayed as

```
<address1> <byte1> <byte2> <address2>
```

Example:

The followmg commands have the same effect:

```
C100,1FF 300
```

or

```
C100L100 300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

D(ump) Command

Format:

```
D[<range>]
```

Function:

Displays the contents of the specified region of memory.

Remarks:

If a range of addresses is specified, the contents of the range are displayed. If the D command is entered without parameters, 128 bytes are displayed at the first address (DS:100) after the one displayed by the previous Dump command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line shows sixteen bytes with a hyphen between the eighth and ninth bytes. Each displayed line, except possibly the first, begins on a 16-byte boundary.

Examples:

If you enter the command:

```
dcs:100 110
```

DEBUG displays

```
04BA:0100 42 45 52 54 41 ... 4E 44 BERTA T. BORLAND
```

If the following command is entered:

```
D
```

the display is formatted as described above. Each line of the display begins with an address incremented by 16 from the address on the previous line. Each subsequent D (entered without parameters) displays the bytes immediately following those last displayed.

If you enter the command:

```
DCS:100 L 20
```

the display is formatted as described above, but 20H bytes are displayed.

If you enter the command:

```
DCS:100 115
```

the display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed.

E(nter) Command

Format:

```
E<address> [<list>]
```

Function:

Enters byte values into memory at the specified <address>.

Remarks:

If the optional <list> of values is entered, the replacement of byte values occurs automatically. (If an error occurs, no byte values are changed.) If the <address> is entered without the optional <list>, DEBUG displays the address and its contents, then repeats the address on the next line and waits for your input. At this point, the Enter command waits for you to perform one of the following actions:

- Replace a byte value with a value you type in. You simply type the value after the current value. If the value typed in is not a legal hexadecimal value or if more than two digits are typed, the illegal or extra character is not echoed.
- Press the space bar to advance to the next byte. To change the value, simply enter the new value as described above. If you space beyond an eight-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, typing the hyphen returns the current position to the previous byte. When the hyphen is typed, a new line is started with the address and its byte value displayed.
- Press RETURN to terminate the Enter command. The RETURN key may be pressed at any byte position.

Example:

Assume the following command is entered:

```
ESC:100
```

DEBUG displays

```
04BA:0100 EB._
```

To change this value to 41, enter "41" as shown.

```
04BA:0100 EB.41_
```

To step through the subsequent bytes, press the space bar to see

```
04BA:0100 EB.41     10.     00.     BC._
```

To change BC to 42:

```
04BA:0100 EB.41      10.      00.      BC.42_
```

Now, realizing that 10 should be 6F, enter the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F.

```
04BA:0100 EB.41      10.      00.      BC.42-
04BA:0102 00.-_
04BA:0101 10.6F_
```

Pressing RETURN ends the Enter command and returns to the DEBUG command level.

F(ill) Command

Format:

F<range> <list>

Function:

Fills the addresses in the <range> with the values in the <list>.

Remarks:

If the <range> contains more bytes than the number of values in the <list>, the <list> will be used repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range>, the extra values in the <list> will be ignored. If any of the memory in the <range> is not valid (bad or nonexistent), the error will be propagated into all succeeding locations.

Example:

Assume the following command is entered:

```
F04BA:100 L 100 42 45 52 54 41
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values are repeated until all 100H bytes are filled.

G(o) Command

Format:

```
G[=<address>][<address>...]]
```

Function:

Executes the program currently in memory.

Remarks:

If the Go command is entered alone, the program executes as if the program had run outside DEBUG.

If =<address> is set, execution begins at the address specified. The equal sign (=) is required, so that DEBUG can distinguish the start =<address> from the breakpoint <address>es.

With the other optional addresses set, execution stops at the first <address> encountered, regardless of that address' position in the list of addresses to halt execution. This happens no matter which branch the program takes. When program execution reaches a breakpoint, the registers, flags, and the decoded instruction are displayed for the last instruction executed. (The result is the same as if you had entered the Register command for the breakpoint address.)

Up to ten breakpoints may be set. Breakpoints may be set only at addresses containing the first byte of an 8086 opcode. If more than 10 breakpoints are set, DEBUG returns the BP Error message.

The user stack pointer must be valid and have six bytes available for this command. The G command uses an IRET instruction to cause a jump to the program under test. The user stack pointer is set, and the user Flags, Code Segment register, and Instruction Pointer are pushed on the user stack. (Thus, if the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es). When an instruction with the breakpoint code is encountered, all breakpoint addresses are restored to their original instructions. If execution is not halted at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Example:

Assume the following command is entered.

```
GCS:7550
```

The program currently in memory executes up to the address 7550 in the CS segment. Then DEBUG displays registers and flags, after which the Go command is terminated.

After a breakpoint has been encountered, if you enter the Go command again, then the program executes just as if you had entered the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

H(ex) Command

Format:

```
H<value> <value>
```

Function:

Performs hexadecimal arithmetic on two specified parameters.

Remarks:

First, DEBUG adds the two parameters. Then, it subtracts the second parameter from the first. The results of the arithmetic is displayed on one line; first the sum, then the difference.

Example:

Assume the following command is entered.

```
B19F 10A
```

DEBUG performs the calculations and then returns the results

```
02A9 0095
```

I(nput) Command

Format:

```
I<value>
```

Function:

Inputs and displays one byte from the port specified by <value>.

Remarks:

A 16-bit port address is allowed.

Example:

Assume the following command is entered.

```
I2F8
```

Assume also that the byte at the port is 42H. DEBUG inputs the byte and displays the value:

```
42
```

L(oad) Command

Format:

```
L[<address> [<drive> <record> <record>]]
```

Function:

Loads a file into memory.

Remarks:

Set BX:CX to the number of bytes read. The file must have been named either with the DEBUG invocation command or with the N command (see the Name command). Both the invocation and the N commands format a filename in the normal format of a file control block at CS:5C.

If the L command is given without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If the L command is given with an address parameter, loading begins at the memory <address> specified. If L is entered with all parameters, absolute disk sectors are loaded, not a file. The <record>s are taken from the <drive> specified (the drive designation is numeric here - 0=A:(FD1), 1=A:(FD2D), 2=B:(FD1), etc.). DEBUG begins loading with the first <record> specified, and continues until the number of sectors specified in the second <record> have been loaded.

Example:

Assume the following commands are entered:

```
A:DEBUG
NFILE.COM
```

Now, to load FILE.COM, enter

```
L
```

DEBUG loads the file and returns the DEBUG prompt. Assume you want to load only portions of a file or certain records from a disk. To do this, enter

```
L04ba:100 2 0F 6D
```

DEBUG then loads 109 (6D hex) records beginning with logical record number 15 into memory beginning at address 04BA:0100. When the records have been loaded, DEBUG simply returns its prompt.

If the file has an .EXE extension, then it is relocated to the load address specified in the header of the .EXE file; the <address> parameter is always ignored for .EXE files. Note that the header itself is stripped off the .EXE file before it is loaded into memory. Thus, the size of a .EXE file on disk will differ from its size in memory.

If the file named by the Name command or specified on invocation is a .HEX file, then entering the L command with no parameters loads the file beginning at the address specified in the .HEX file. If the L command includes the option <address>, DEBUG adds the specified address to the address found in the .HEX file in order to determine the start address for loading the file.

M(ove) Command

Format:

```
M<range> <address>
```

Function:

Moves the block of memory specified by <range> to the location beginning at the <address> specified.

Remarks:

Overlapping moves (moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that could be overwritten are moved first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the block's lowest address, working towards the highest. The sequence for moves from lower addresses to higher addresses is to move the data beginning at the block's highest address, working towards the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data there before the move will remain; that is, the M command really copies the data from one area into another, in the sequence described, and writes over the new addresses. This is why the sequence of the move is important.

Example:

Assume you enter

```
MCS:100 110 C5:500
```

DEBUG first moves address CS:110 to address CS:510, then CS:10F to CS:50F, and so on, until CS:100 is moved to CS:500. You should enter the D command, using the <address> entered for the M command, to review the results of the move.

N(ame) Command

Format:

```
N<filename> [<filename>...]
```

Function:

Sets filenames.

Remarks:

The Name command performs two distinct functions, both having to do with filenames. First, Name is used to assign a filename for a later Load or Write command. Thus, if you invoke DEBUG without naming any file to be debugged, then the N<filename> command must be given before a file can be loaded. Second, Name is used to assign filename parameters to the file being debugged. In this case, Name accepts a list of parameters that are used by the file being debugged.

These functions overlap. Consider the following set of DEBUG commands.

```
>NFILE1.EXE
>L
>G
```

Because of the two-pronged effect of the Name command, the following happens.

1. N(ame) assigns the filename FILE1.EXE to the filename to be used in any later Load or Write commands.
2. N(ame) also assigns the filename FILE.EXE to the first filename parameter to be used by any program that is later debugged.
3. L(oad) loads FILE.EXE into memory.
4. G(o) causes FILE.EXE to be executed with FILE.EXE as the single filename parameter. (That is, FILE.EXE is executed as if FILE.EXE had been typed at the command level.)

A more useful chain of commands might go like this:

```
>NFILE1.EXE
>L
>NFILE2.DAT FILE3.DAT
>G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory, and then the Name command is used again, this time to specify the parameters to be used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE is executed as if FILE1 FILE2-.DAT FILE3.DAT had been typed at the MS-DOS command level. Note that if a Write command were executed at this point, then FILE1.EXE — the file being debugged — would be saved with the name FILE2.DAT! To avoid such undesirable results, you should always execute a Name command before either a Load or a Write.

There are four distinct regions of memory that can be affected by the Name command:

CS:5C	FCB for file 1
CS:6C	FCB for file 2
CS:80	Count of characters
CS:81	All characters entered

A File Control Block (FCB) for the first filename parameter given to the Name command is set up at CS:5C. If a second filename parameter is given, then an FC8 is set up for it beginning at CS:6C. The number of characters typed in the Name command (exclusive of the first character, N) is given at location CS:80. The actual stream of characters given by the Name command (again, exclusive of the letter N) begins at CS:81. Note that this stream of characters may contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

Example:

A typical use of the Name command would be

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the Go command executes the file in memory as if the following command line had been entered:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal runtime environment for PROG.COM.

O(utput) Command

Format:

```
O<value> <byte>
```

Function:

Sends the <byte> specified to the output port specified by <value>.

Remarks:

A 16-bit port address is allowed.

Example:

Enter

```
02F8 4F
```

DEBUG outputs the byte value 4F to output port 2F8.

Q(uit) Command

Format:

```
Q
```

Function:

Terminates the debugger..

Remarks:

The Q command takes no parameters and exits DEBUG without saving the file currently being operated on. You return to the MS-DOS command level.

Example:

To end the debugging session, enter

```
Q
```

DEBUG is terminated, and control returns to the MS-DOS command level.

R(egister) Command

Format:

R[<register-name>]

Function:

Displays the contents of one or more CPU registers.

Remarks:

If no <register-name> is entered, the R command dumps the register save area and displays the contents of all registers and flags.

If a register name is entered, the 16-bit value of that register is displayed in hexadecimal, and then a colon appears as a prompt. You then either enter a <value> to change the register, or simply press RETURN if no change is wanted.

The only valid <register-name>s are

AX	BP	SS	
BX	SI	CS	
CX	DI	IP	(IP and PC both refer to the
DX	DS	PC	instruction pointer.)
SP	ES	F	

Any other entry for <register-name> results in a BR Error message.

If F is entered as the <register-name>, DEBUG displays each flag with a two-character alphabetic code. To alter any flag, enter the opposite two-letter code. The flags are either set or clear.

The flags with their codes for set and clear are listed in Table 7-3.

Table 7-3 Register Command Flags

FLAG NAME	SET CODE	CLEAR CODE
Overflow	OV	NV
Direction	DN Decrement	UP Increment
Interrupt	EI Enabled	DI Disabled
Sign	NG Negative	PL Plus
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE Even	PO Odd
Carry	CY	NC

Whenever you enter the command RF, the flags are displayed in the order shown above in a row at the beginning of a line. At the end of the list of flags, DEBUG displays a hyphen (-). You may enter new flag values as alphabetic pairs. The new flag values can be entered in any order. You are not required to leave spaces between the flag entries. To exit the R command, press RETURN. Flags for which new values were not entered remain unchanged.

If more than one value is entered for a flag, DEBUG returns a DF error message. If you enter a flag code other than those shown above, DEBUG returns a BF error message. In both cases, the flags up to the error in the list are changed; flags at and after the error are not.

At start up, the segment registers are set to the bottom of free memory, the instruction pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

Example:

Enter

```
R
```

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, then DEBUG might display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A  CD21          INT     21
```

If you enter

```
RF
```

DEBUG displays the flags:

```
NV UP DI NG NZ AC PE NC - _
```

Now enter any valid flag designation, in any order, with or without spaces.

For example, you enter

```
NV UP DI NG NZ AC PE NC - PLEICY
```

DEBUG responds only with the DEBUG prompt. To see the changes, entei either the R or RF command.

```
NV UP EI PL NZ AC PE CY - _
```

Press RETURN to leave the flags this way or to enter different flag values.

S(earch) Command

Format:

```
S<range> <list>
```

Function:

Searches the range specified for the list of bytes specified.

Remarks:

The <list> may contain one or more bytes, each separated by a space or comma. If the <list> contains more than one byte, only the first address of the byte string is returned. If the <list> contains only one byte, all addresses of the byte in the <range> are displayed.

Example:

If you enter

```
SCS:100 110 41
```

DEBUG might return the response:

```
04BA:0104
04BA:010D
_
```

T(race) Command

Format:

```
T[=<address>][<value>]
```

Function:

Executes one instruction and displays the contents of all registers, flags, and the decoded instruction.

Remarks:

If the optional =<address> is entered, tracing occurs at the address specified. The optional value causes DEBUG to execute and trace the number of steps specified by <value>.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you may also trace instructions stored in ROM.

Example:

Enter

```
T
```

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. Assume that the current position is 04BA:011A; then DEBUG might return the display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A  CD21          INT    21
```

Now enter

```
T=011A 10
```

DEBUG executes sixteen (10 hex) instructions beginning at 011A in the current segment and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that CTRL-S suspends the display at any point, so that you can study the registers and flags for any instruction.

U(nassemble) Command

Format:

```
U[<range>]
```

Function:

Disassembles bytes and displays the source statements that correspond to them, along with addresses and byte values.

Remarks:

The display of disassembled code looks like a listing for an assembled file. If you enter the U command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous Unassemble command. If you enter the U command with the <range> parameter, then DEBUG disassembles all bytes in the range. If the <range> is given as an address only, then 20H bytes are disassembled, not 80H.

Example:

Enter

```
U04BA:100 L10
```

DEBUG disassembles 16 bytes beginning at address 04BA:0100.

```
04BA:0100   206472   AND   [SI+72],AH
04BA:0103   69       DB    69
04BA:0104   7665     JQE   016B
04BA:0106   207370   AND   [BP+DI+70],DH
04BA:0109   65       DB    65
04BA:010A   63       DB    63
04BA:010B   69       DB    69
04BA:010C   66       DB    66
04BA:010D   69       DB    69
04BA:010E   63       DB    63
04BA:010F   61       DB    61
```

If you enter

```
u04ba:0100 0108
```

The display shows

```
04BA:0100   206472   AND   [SI+72],AH
04BA:0103   69       DB    69
04BA:0104   7665     JBE   016B
04BA:0106   207370   AND   [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. The U command can be entered for the changed locations, the new instructions viewed and the disassembled code used to edit the source file.

W(rite) Command

Format:

```
W[<address> [<drive> <record> <record>]]
```

Function:

Writes the file being debugged to a disk file.

Remarks:

If only the W appears, BX:CX must already be set to the number of bytes to be written. The file is written beginning from CS:100. If the W command is given with just an address, then the file is written beginning at that address. If a G or T command was used, BX:CX must be reset before using the Write command without parameters. (Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file as long as the length has not changed.)

The file must have been named either with the DEBUG invocation command or with the N command (see the Name command). Both the invocation and the N commands format a file name in the normal format of a File Control Block at CS:5C.

If the W command is given with parameters, the write begins from the memory address specified. The file is written to the <drive> specified (the drive designation is numeric here — D=A:(FD1), 1=A:(FD2D), 2=B:(FD1), etc.). DEBUG writes the file beginning at the logical record number specified by the first record and continues until the number of sectors specified in the second record have been written.

CAUTION

Writing to absolute sectors is extremely dangerous, because the process bypasses the file handler.

Examples:

Enter

```
W
```

DEBUG writes out the file to disk then displays the DEBUG prompt.

```
W
>_
```

Another example is as follows:

```
WCS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100 to the disk in drive B. The data written out starts in disk logical record number 37H and consists of 2BH records. When the write is complete, DEBUG displays the prompt:

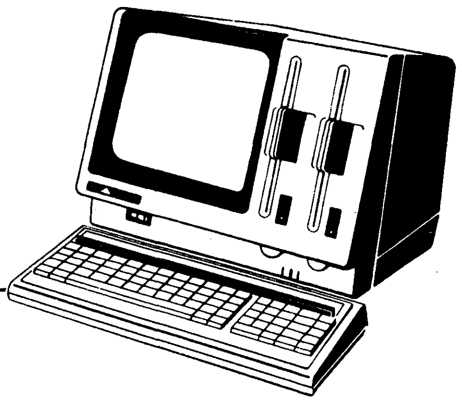
```
WCS:100 1 37 2B
>_
```

DEBUG ERROR MESSAGES

During the DEBUG session, you may receive any of the error messages shown in Table 7-4. Each error terminates the DEBUG command with which it is associated, but does not terminate DEBUG itself.

Table 7-4 DEBUG Error Messages

ERROR CODE	DEFINITION
BF	Bad flag. You 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. You specified more than ten breakpoints as parameters to the G command. Reenter the Go command with ten or fewer breakpoints.
BR	Bad register. You entered the R command with an invalid register name. See the Register command for the list of valid register names.
DF	Double flag. You entered a two values for one flag. The user may specify a flag value only once per RF command.



Chapter 8

The FC File Comparison Utility

It is sometimes useful to compare files on your disks. If you have copied a file and later want to compare copies to see which one is current, you can use the File Comparison Utility (FC).

FC compares the contents of two files. The differences between the two files can be output to the console or to a third file. The files being compared may be either text files (source files containing source statements of a programming language or data files) or binary files (files output by the MACRO-86 Macro Assembler, the MS-LINK Linker Utility, or by a high-level language compiler).

The comparisons are made in one of two ways: on a line-by-line or a byte-by-byte basis. The line-by-line comparison isolates blocks of lines that are different between the two files and prints those blocks of lines. The byte-by-byte comparison displays the bytes that are different between the two files.

LIMITATIONS ON SOURCE COMPARISONS

FC uses a large amount of memory as buffer (storage) space to hold the source files. If source files are larger than available memory, FC will compare what can be loaded into the buffer space. If no lines in the portions of these files match, FC will only display the message:

```
FILES ARE DIFFERENT
```

For binary files larger than available memory, FC compares both files completely, overlaying the portion of the file in memory with its next portion from disk. All differences are output in the same manner as for files that fit completely in memory.

FILE SPECIFICATIONS

All file specifications centered for FC use the following syntax:

```
[d:] <filename> [<.ext>]
```

where *d:* is the letter designating a disk drive. If the drive designation is omitted, FC defaults to the current default drive.

<filename> is the one- to eight-character name of a file.

<.ext> is a one- to three-character extension to the filename.

INVOKING FC

The syntax of the command FC is as follows:

```
FC [/# /B /W /C] <filename1> <filename2>
```

FC matches the first file (filename1) against the second (filename2) and reports any differences between them. Both filenames can be pathnames. For example:

```
FC B: \FOO\BAR\FILE1.TXT \BAR\FILE2.TXT
```

FC takes FILE1.TXT in the \FOO\BAR directory of disk B and compares it with FILE2.TXT in the \BAR directory. Since no drive is specified for filename2, FC assumes that the \BAR directory is on the disk in the default drive.

You can use four switches with FC. These switches are summarized in Table 8-1.

Table 8-1 FC Command Switches

SWITCH	DESCRIPTION

/B	Forces a binary comparison of both files. The two files are compared byte-to-byte, with no attempt to re-synchronize after a mismatch. The mismatches are printed as follows:
	
	`--ADDRS----F1----F2-`
	`xxxxxxxx    yy    zz`
	
	where: xxxxxxxx is the relative address of the pair of bytes from the beginning of the file.

Table 8-1 FC Command Switches (cont'd)

SWITCH	DESCRIPTION
	Addresses start at 00000000; yy and zz are mismatched bytes from filename1 (F1) and filename2 (F2), respectively. If one of the files contains less data than the other, then a message is printed out. For example, if filename1 ends before filename2, then FC displays ***Data left in F2***
/#	# stands for a number from 1 to 9. This switch specifies the number of lines required to match for the files to be considered as matching again after a difference has been found. If this switch is not specified, it defaults to 3. This switch is used only in source comparisons.
/W	Causes FC to compress whites (tabs and spaces) during the comparison. Thus, multiple contiguous whites in any line will be considered as a single white space. Note that although FC compresses whites, it does not ignore them. The two exceptions are beginning and ending whites in a line, which are ignored. For example (note that an underscore represents a white), ____More___data_to_be_found____ will match with More_data_to_be_found and with ______More______data_to_be_____found______ but will not match with ____Moredata_to_be_found This switch is used only in source comparisons.

Table 8-1 FC Command Switches (cont'd)

SWITCH	DESCRIPTION

/C	Causes the matching process to ignore the case of letters. All letters in the files are considered uppercase letters. For example,
	
	`Much_MORE_data_IS_NOT_FOUND`
	
	will match
	
	`much_more_data_is_not_found`
	
	If both the /W and /C options are specified, then FC will compress whites and ignore case. For example,
	
	`___DATA_was_found____`
	
	will match
	
	`data_was_found`
	
	This switch is used only in source comparisons.

DIFFERENCE REPORTING

FC reports the differences between the two files you specify by displaying the first filename, followed by the lines that differ between the files, followed by the first line to match in both files. (The matches serve as delimiters to resynchronize the files for the continuing comparison.) FC then displays the name of the second file followed by the lines that are different, followed by the first line that matches. The default for the number of lines to match between the files is 3. If you want to change this default, specify the number of lines with the /# switch. For example:

```
    . . .
    . . .
----------<filename1>
<difference>
<1st line to match filename2 in filename1>
```

```
----------<filename2>
<difference>
<1st line to match filename1 in filename2>
--------------------------------------------

          . . .
          . . .
```

FC will continue to list each difference.

If there are too many differences (involving too many lines), the program will simply report that the files are different and stop.

If no matches are found after the first difference is found, FC will display

```
*** Files are different ***
```

and will return to the system prompt.

REDIRECTING FC OUTPUT TO A FILE

The differences and matches between the two files you specify will be displayed on your screen unless you redirect the output to a file. This is accomplished in the same way as MS-DOS command redirection. (Refer to the *MS-DOS System User's Guide* for more information on the redirection feature.)

For example, to compare File1 and File2 and then send the FC output to DIFFER.TXT, enter

```
FC File1 File2 > DIFFER.TXT
```

The differences and matches between File1 and File2 will be put into DIFFER.TXT on the default drive.

FILE COMPARISON EXAMPLES

Three examples of file comparison are given in the next few pages.

Example 1:

Assume the following two ASCII files are on disk.

ALPHA.ASM	BETA.ASM
FILE A	FILE B
A	A
B	B
C	C
D	G
E	H
F	I
G	J
H	1
I	2
M	P
N	Q
O	R
P	S
O	T
R	U
S	V
T	4
U	5
V	W
W	X
X	Y
Y	Z
Z	

To compare the two files and display the differences on the APC screen, enter

```
FC ALPHA.ASM BETA.ASM
```

FC compares ALPHA.ASM with BETA.ASM and displays the differences on the APC screen. All other defaults remain intact. (The defaults are: "do not use tabs, spaces, or comments for matches," and "do a source comparison on the two files.")

The output will appear as follows on the APC screen (not including the notes):

```
----------ALPHA.ASM
D                         NOTE: ALPHA file
E                         contains DEFG,
F                         BETA contains G.
G
----------BETA.ASM
G
------------------------------

----------ALPHA.ASM
M                         NOTE: ALPHA file
N                         contains MNO where
O                         BETA contains J12.
P
----------BETA.ASM
J
1
2
P
------------------------------

----------ALPHA.ASM
W                         NOTE: ALPHA file
                          contains W where
----------BETA.ASM        BETA contains 45W.
4
5
W
```

Example 2:

You can print the differences between the same two source files on the line printer. In this example, four successive lines must be the same to constitute a match.

Enter

```
FC -4 ALPHA.ASM BETA.ASM > PRN
```

The following output will appear on the line printer.

```
----------ALPHA.ASM
D
E
F
G
H
I
M
N                                   NOTE: P is the first of
O                                   a string of 4 matches.
P
----------BETA.ASM
G
H
I
J
1
2
P

-----------------------------

----------ALPHA.ASM
W
                                    NOTE: W is the first of
----------BETA.ASM                  a string of 4 matches.
4
5
W
```

Example 3:

This example forces a binary comparison and then displays the differences on the APC screen using the same two source files as were used in the previous examples.

Enter

```
FC /B ALPHA.ASM BETA.ASM
```

The /B switch in this example forces a binary comparison. This switch and any others must be typed before the filenames in the FC command line. The following display should appear.

```
--ADDRS----F1---F2--
 00000009   44  47
 0000000C   45  48
 0000000F   46  49
 00000012   47  4A
 00000015   48  31
 00000018   49  32
 0000001B   4D  50
 0000001E   4E  51
 00000021   4F  52
 00000024   50  53
 00000027   51  54
 0000002A   52  55
 0000002D   53  56
 00000030   54  34
 00000033   55  35
 00000036   56  57
 00000039   57  58
 0000003C   58  59
 0000003F   59  5A
 00000042   5A  1A
```

FC ERROR MESSAGES

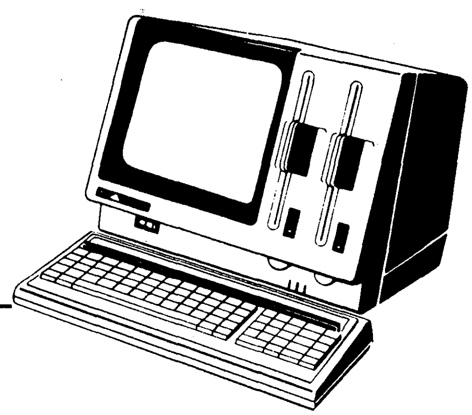
When the File Comparison Utility detects an error, one or more of the error messages will be displayed in Table 8-2.

Table 8-2 FC Error Messages

ERROR MESSAGE	MEANING
Incorrect DOS version	You are running FC under a version of MS-DOS that is not 2.0 or higher.
Invalid parameter: <option>	One of the switches that you have specified is invalid.
File not found: <filename>	FC could not find the filename you specified.
Read error in: <filename>	FC could not read the entire file.
Invalid number of parameters	You have specified the wrong number of options on the FC command line.

Chapter 9

The Auxiliary Character Generator Program



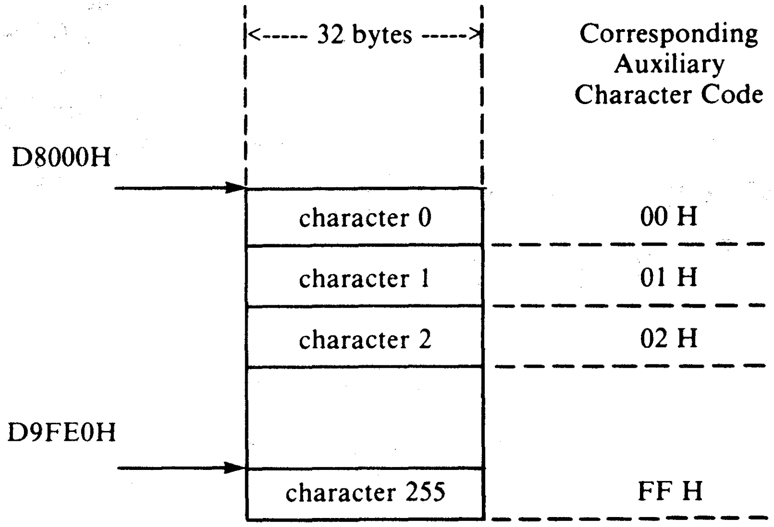
It is possible under MS-DOS to write assembly language programs that display specially created 8 by 16 matrix characters on the APC screen instead of the characters from the computer's default set. Before a special character set can be displayed, it must be created using the Auxiliary Character Generator Program and loaded by this same program to a reserved portion of random access memory, called the auxiliary character generator (CG) RAM.

This chapter reviews the Auxiliary Character Generator Program, originally presented in the *MS-DOS System User's Guide.*

It also describes the auxiliary character generator (CG) RAM address and format, and explains how you direct the loading of a character set.

AUXILIARY CHARACTER GENERATOR RAM ADDRESS AND FORMAT

The following illustration represents the format of the auxiliary CG RAM.



The starting address of the auxiliary CG RAM is D8000H. Each character is made up of a 32-byte bit pattern. You must transfer a character set to the auxiliary CG RAM in words, not bytes. This is a hardware restriction.

Figure 9-1 is an example of the bit pattern for a graphic character.

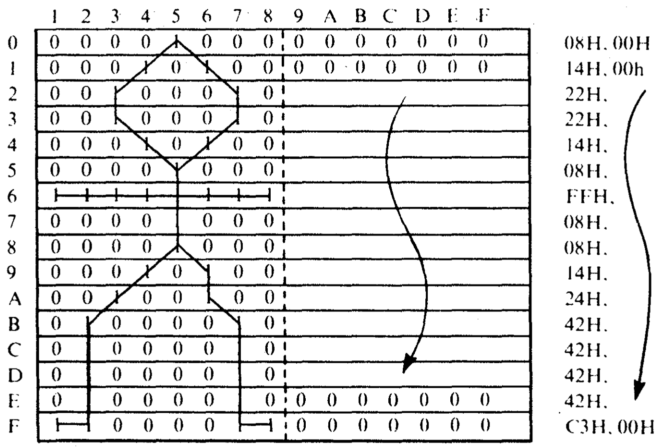
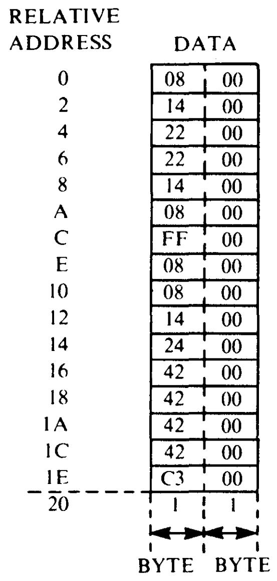


Figure 9-1 Example of a Bit Pattern for a Graphic Character

The following illustration demonstrates how the bit pattern in Figure 9-1 would be stored in the auxiliary CG RAM.



RELATIVE ADDRESS	DATA	
0	08	00
2	14	00
4	22	00
6	22	00
8	14	00
A	08	00
C	FF	00
E	08	00
10	08	00
12	14	00
14	24	00
16	42	00
18	42	00
1A	42	00
1C	42	00
1E	C3	00
20		

Figure 9-2 Sample Data in Auxiliary CG RAM

THE AUXILIARY CHARACTER GENERATOR PROGRAM

The Auxiliary Character Generator Program (CHR) is invoked by entering the CHR external command to MS-DOS. This program both generates the character set and loads it into the auxiliary CG RAM.

Creating the Auxiliary Character Set

CHR creates (and updates) an auxiliary character set stored in a data file with the file extension CHR. Any number of character sets can be created, each containing up to 256 character patterns identified by the hexadecimal values 00 to FF. Each auxiliary character is within a set is constructed in an 8 by 16 matrix.

To invoke the Auxiliary Character Generator Program, enter CHR at the system prompt.

The following prompt appears.

```
UPDATE OR LOAD (U,L)?
```

If you enter L, you must identify the input filename in one of two formats: <filename> or <d:filename>.

If you only press RETURN, CHR automatically searches the default advice for the filename AUXCG.CHR, which is the default auxiliary character file. When the input file is located, the auxiliary character set is loaded into memory starting at hexadecimal address 0D8000.

If you enter U to update an auxiliary character file, the filename you enter for the input file determines whether you will be maintaining an existing file or creating a new one. If the specified filename does not exist, CHR creates a file on drive d: (the default) using the name you entered.

Next, you must enter an output file specification. The output file will store the changed version of the character set. The entry format is the same as for the input file specification. If you only press RETURN, the filename of the output file is the same as that for the input file.

When you press certain keys during CHR's operations you issue sub-commands. Table 9-1 lists the keys you can use and the functions they perform.

Table 9-1 CHR Sub-Commands

SUB-COMMAND	KEY USED	FUNCTION
Cursor Up	↑	Moves the cursor upward one row at a time within the matrix.
Cursor Down	↓	Moves the cursor downward one row at a time within the matrix.
Cursor Left	+	Moves the cursor to the left one column at a time within the matrix.
Cursor Right	-	Moves the cursor to the right one column at a time within the matrix.

Table 9-1 CHR Sub-Commands (cont'd)

SUB-COMMAND	KEY USED	FUNCTION
Bit Off	Space	Turns off the bit or graphic block at the current cursor position by overtyping with a space. The cursor then moves forward one position.
Bit On	*	Turns on the bit at the current cursor position. The graphic block at the position appears highlighted, and the cursor moves forward one position.
Cursor Home	CLEAR HOME	Moves the cursor to the home position, the upper left corner of the matrix (0,0).
Display	D	Displays or redisplays the current character following modifications.
Next	RETURN	Displays the next character in the auxiliary character set according to the next hexadecimal code in sequence. Pressing RETURN at the first character in the file (code=FF) redisplays the first character (code=00).
Back	B	Displays the preceding character in the auxiliary character set according to the preceding hexadecimal code. Pressing B at the first character in the file (code=00) redisplays the last character (code=FF).
Load	L	Loads all the characters of the auxiliary character set being updated or created into memory.

Table 9-1 HCR Sub-Commands (cont'd)

SUB-COMMAND	KEY USED	FUNCTION
Search Code	C	Prompts for input of a hexadecimal code, then displays the character corresponding to that code. Pressing RETURN instead of entering a code redisplays the most recently displayed character on the screen.
End	E	Displays a prompt confirming the end of auxiliary character set updating. Enter Y to end the program. When you are creating a new file, a prompt asking "CREATE filename?" appears. When you are updating an existing file, a prompt asking "UPDATE filename?" appears.

Loading the Auxiliary Character File

Use the following procedure to load the auxiliary CG RAM with data from the CHR file.

1. Open the .CHR file.
2. Read the .CHR file into the user buffer. (See the file format below.)
3. Transfer the data (8K bytes) in the user buffer to the auxiliary CG RAM, starting at D8000H, by word.

The following illustrates the .CHR file format.

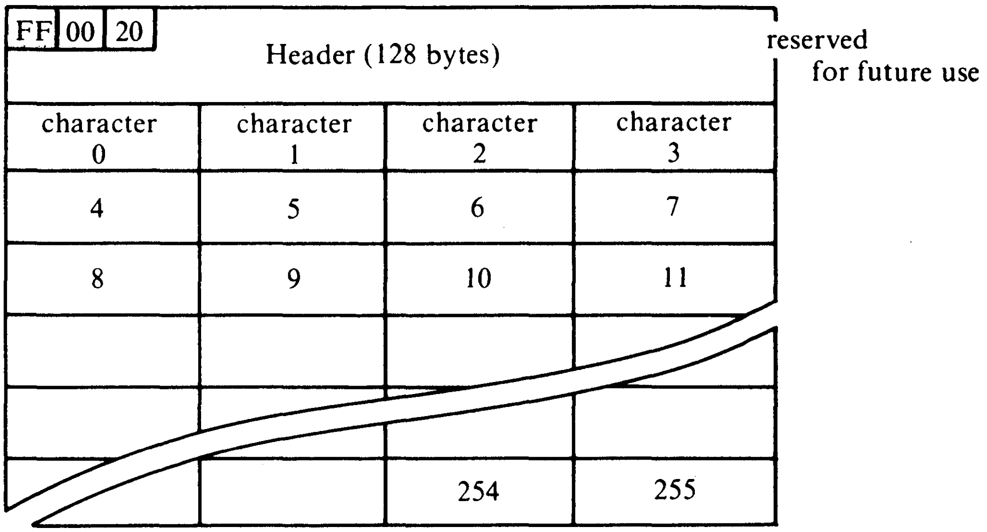


Figure 9-3 .CHR File Format

DISPLAYING THE AUXILIARY CHARACTER SET

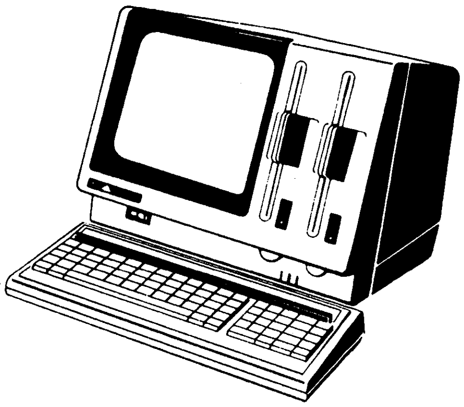
Once an auxiliary character set has been loaded in the auxiliary CG RAM, it can be displayed using the external command DISP.

At the system prompt, enter

```
DISP
```

When you invoke DISP, the characters for all 256 hexadecimal codes in the currently loaded set are displayed in a matrix on the APC screen. Codes without assigned characters appear as blanks or miscellaneous images. If no auxiliary character set is currently loaded, all positions hold miscellaneous images.

Note that the default APC character set can also be displayed. The DISP1 external command does this.



Chapter 10

The Soft Key Definition Program

Assembly language programs can access a soft key table in the I/O System containing data strings or control characters assigned to the dual-mode function keys PF1 through P16. This chapter describes the Soft Key Definition Program that generates the function key definitions and loads them into the Soft Key Table. It also includes a description of the Soft Key Table address and format, as well as an explanation of how to load key definitions to it.

SOFT KEY TABLE FORMAT AND ADDRESS

The Soft Key Table address in memory and its format can be illustrated as follows.

THE SOFT KEY DEFINITION PROGRAM

The Soft Key Definition Program allows you to create, display, change, and delete the dual-mode function key definitions. The external command KEY invokes this program.

KEY requires a data file with the file extension KEY for input.

A total of 32 functions can be defined in each .KEY file considering the shifted position for each key. The shifted function of PF1 through PF16 is accessed by simultaneously pressing the function key and FNC.

Creating the Soft Key Table

To invoke KEY, you enter the KEY external command at the system prompt. The following prompt appears:

```
SOFT KEY DEFINITION PROGRAM V1.1
    UPDATE OR LOAD (U,L)?
```

If you enter L, you must then identify the input filename in one of two formats: <filename> or <d:filename>.

If you type U to update a function key file, the filename you enter for the input file determines whether you will be maintaining an existing file or creating a new one. If the filename entered does not exist, KEY creates a file by that on d:, the default drive.

An output file specification must then be entered. The output file stores the new or changed function key set. The entry format is the same as for the input file. If you press only RETURN, the output file is the same as the input file.

KEY has the sub-commands listed in Table 10-1.

Table 10-1 KEY Sub-Commands

SUB-COMMAND	ACTION
F##[XXXXXXXXXXXXXXX]	Assigns a function to a particular key. The ## identifies the number of the key (1-16) you are assigning the function to. The optional F before the number indicates that there is also a function assigned to the key in its shifted moved. The Xs represent the actual function assigned, expressed in no more than 15 characters.
D	Displays all function key assignments currently in memory.
E	Ends function key assignment.

In addition to data strings, you can also include standard ASCII and MS-DOS control codes within function key assignments. To invoke a carriage return as a part of the data string, use the ^ character.

The following represent valid function key assignments that you can make after you enter input and output file specifications.

```
1,COPY B:*.*^
```

The ^ performs a carriage return after the command COPY B:*.* is entered by pressing PF1.

If you enter the following assignment:

```
F16,KEY^
```

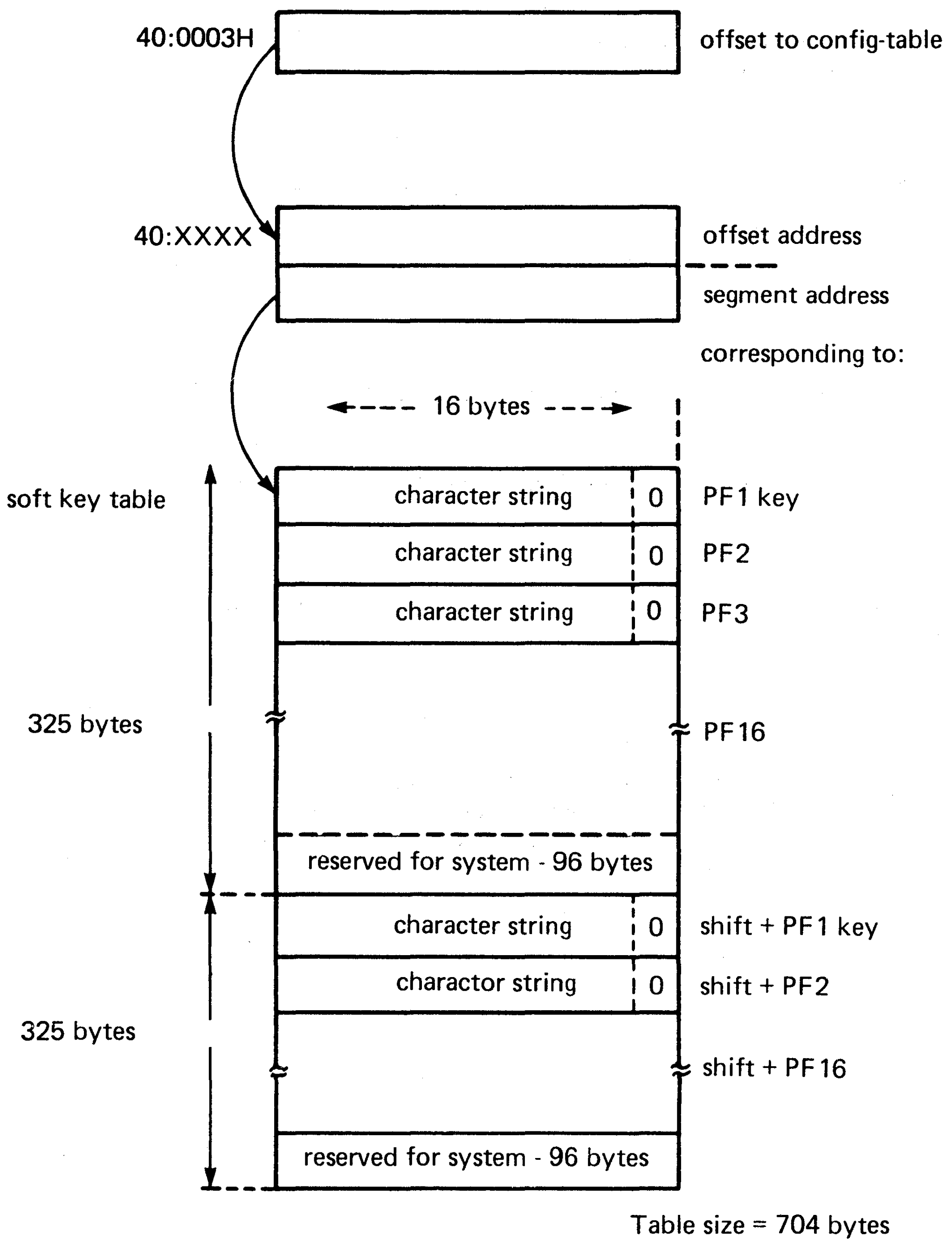
The ^ performs a carriage return after the KEY command is entered when FNC and PF16 are pressed simultaneously.

The soft key routine in the I/O System recognizes the end of the character string by the code 00H.

Loading the Soft Key Table

Use the following procedure to load the Soft Key Table file generated by the Soft Key Definition Program.

1. Open the KEY file.
2. Read the KEY file. (See file format below.)
3. Transfer the data to the Soft Key Table in two moves. Load the first 256 bytes starting at the address of the table. This address can be found in the Configuration Table loaded by CONFIG.SYS. Load the next 256 bytes starting at 352 plus the start address of the Soft Key Table.


40:0003H
offset to config-table
40:XXXX
offset address
segment address
corresponding to:
16 bytes
soft key table
character string
0
PF1 key
character string
0
PF2
character string
0
PF3
325 bytes
PF16
reserved for system - 96 bytes
character string
0
shift + PF1 key
charactor string
0
shift + PF2
325 bytes
shift + PF16
reserved for system - 96 bytes
Table size = 704 bytes

KEY FILE FORMAT

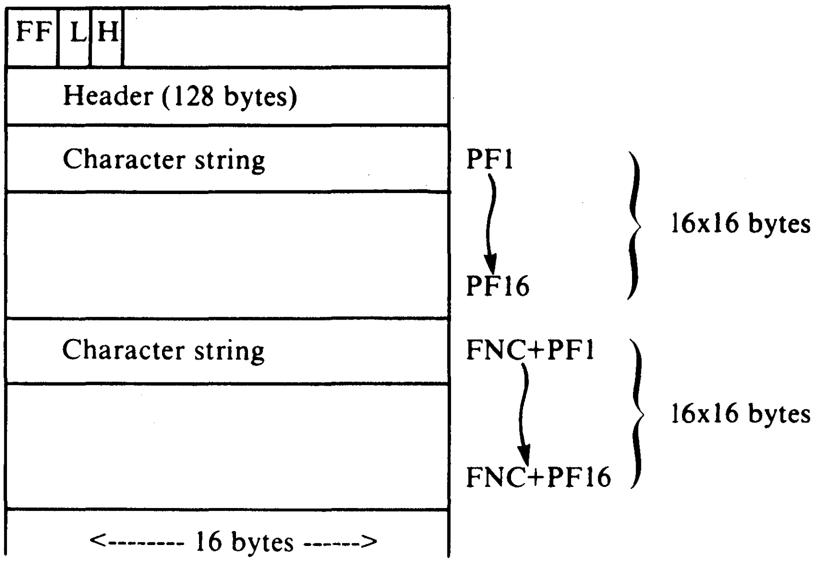


Figure 10-1 KEY File Format

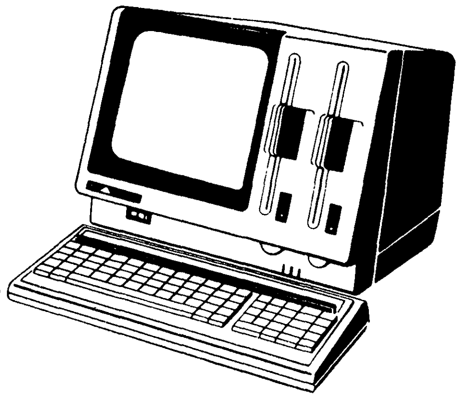
The last six function keys on the keyboard, unshifted and shifted, produce the following predefined escape sequences.

ESCOO	ESCOU
ESCOP	ESCOV
ESCOQ	ESCOW
ESCOR	ESCOX
ESCOS	ESCOY
ESCOT	ESCOZ

These sequences do not actually perform any function. They must be implemented by an application program, which establishes an action or series of operations for each escape sequence.

The last six dual-mode function keys on the keyboard, unshifted and shifted, produce the following predefined escape sequences.

ESCOO	ESCOU
ESCOP	ESCOV
ESCOQ	ESCOW
ESCOR	ESCOX
ESCOS	ESCOY
ESCOT	ESCOZ



Chapter 11
Creating Device Drivers

MS-DOS allows you to create the device drivers for controlling input and output to the serial I/O devices attached to the APC.

WHAT IS A DEVICE DRIVER?

A device driver under MS-DOS is a binary file of code that manipulates the peripheral devices while providing a consistent interface to the operating system. This file has a special header at the beginning that identifies it as a device, defines the strategy and interrupt entry points, and describes various attributes of the device.

There are two kinds of device drivers: character device drivers and block device drivers.

Character devices are designed to perform serial character I/O. These devices are given names (for example, CON, AUX, PRN, and CLOCK), so user programs open channels (FCBs) to do I/O to them.

Block devices are the "disk drives" on the system, which perform random I/O in blocks (usually the physical sector size). These devices are not named as the character devices are, and therefore cannot be opened directly. Instead they are identified by the drive letters: A,B,C, and so on.

Block devices also have units. A single driver may be responsible for one or more disk drives. For example, block device driver ALPHA may be responsible for drives A:,B:,C:, and D:. This means that it has four units (0-3) defined and, therefore, takes up four drive letters. The position of the driver in the list of all drivers determines which units correspond to which drive letters. If driver ALPHA is the first block driver in the device list and it defines 4 units (0-3), its devices will be A:,B:,C:, and D:. If BETA is the second block driver and defines three units (0-2), the devices will be E:,F:, G:, and so on. MS-DOS 2.0 is not limited to 16 block device units, as previous versions were. The theoretical limit is 63 (26 - 1), but note that after 26 the drive letters are unconventional.

NOTE

Character devices cannot define multiple units because they have only one name.

DEVICE HEADERS

A device header is required at the beginning of a device driver. A device header is illustrated in Figure 11-1.

DWORD pointer to next device defined by this driver (Must be set to -1 for last device defined in the driver)
WORD attributes Bit 15 = 1 if char device 0 is blk if bit 15 is 1 Bit 0 = 1 if current sti device   Bit 1 = 1 if current sto output   Bit 2 = 1 if current NUL device   Bit 3 = 1 if current CLOCK dev   Bit 4 = 1 if special   Bits 5-12 Reserved; must be set to 0 Bit 14 is the IOCTL bit Bit 13 is the NON IBM FORMAT bit
WORD pointer to device strategy entry point
WORD pointer to device interrupt entry point
8-BYTE character device name field Character devices set a device name. For block devices the first byte is the number of units

Figure 11-1 Device Header Format

Note that the device entry points are words. They must be offsets from the same segment number used to point to this table. For example, if XXX:YYY points to the start of the table, then XXX:strategy and XXX:interrupt are the entry points.

Pointer to Next Device Field

The pointer to the next device header field is a double word field (offset followed by the segment) that is set by MS-DOS to point at the next driver in the system list at the time the device driver is loaded. It is important that this field be set to -1 prior to load (when it is on the disk as a file) unless there is more than one device driver in the file. If there is more than one driver in the file, the first word of the double word pointer should be the offset of the next driver's device header.

NOTE

The last driver in the .COM file must have the pointer to the next device header field set to -1.

Attribute Field

The attribute field is used to tell the system whether a device is a block or character device (bit 15). Other bits of the field are used to give selected character devices certain special treatment. (Note that these bits mean nothing on a block device.) For example, assume that you have a new device driver that you want to be the standard input and output device. Besides installing the driver, you must tell MS-DOS that you want the new driver to override the current standard input and standard output (the CON device). This is accomplished by setting the attributes to the desired characteristics (bits 0 and 1 to 1). Note that they are separate!

Similarly, a new CLOCK device could be installed by setting that attribute. (Refer to the section THE CLOCK DEVICE in this chapter for more information.)

Although there is a NUL device attribute, the NUL device cannot be reassigned. This attribute exists so that MS-DOS can determine if the NUL device is being used.

The NON IBM FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (Bios I/O System Parameter Block) device call. (Refer to the section MEDIA CHECK AND BUILD BPD for further information on this call.)

The IOCTL bit has meaning on character and block devices. This bit tells MS-DOS whether the device can handle control strings (via the IOCTL system call, Function 44H. If a driver cannot process control strings, it should initially set this bit to 0. This tells MS-DOS to return an error if an attempt is made (via Function 44H) to send or receive control strings to this device. A device that can process control strings should initialize the IOCTL bit to 1. For drivers of this type, MS-DOS will make calls to the IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data to be sent and received by the device for its own use (for example, to set baud rate, stop bits, and form length), instead of passing data over the device channel as for a normal read or write. The interpretation of the passed information is up to the device, but it must not be treated as a normal I/O request.

Strategy and Interrupt Fields

These two fields are the pointers to the entry points of the strategy and interrupt routines. They are word values, so they must be in the same segment as the device header.

Name Field

This is an 8-byte field that contains the name of a character device or the number of units of a block device. If it names a block device, the number of units can be put in the first byte. This is optional, because MS-DOS will fill this location with the value returned by the driver's INIT code. Refer to the section INSTALLATION OF DEVICE Drivers in this chapter for more information.

HOW TO CREATE A DEVICE DRIVER

In order to create a device driver that MS-DOS can install, you must write a binary file with a device header at the beginning of the file. Note that for device drivers, the code should not originate at 100H, but rather at 0 (ORG 0 or no ORG statement in the source code). This is because files are not loaded using the Program Segment Prefix. The link field (pointer to next device header) should be -1, unless there is more than one device driver in the file. The attribute field and entry points must be set correctly.

If the device is a character device, the name field should contain the name of that character device. The name can be any legal 8-character filename.

MS-DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on a new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

NOTE

Because MS-DOS can install the driver anywhere in memory, care must be taken in any far memory references. You should not expect that your driver will always be loaded in the same place every time.

INSTALLATION OF DEVICE DRIVERS

MS-DOS 2.0 allows new device drivers to be installed dynamically at boot time. This is accomplished by the INIT code in the I/O System, which reads and processes the CONFIG.SYS file.

MS-DOS calls the device drivers in the following manner:

- MS-DOS makes a far call to the strategy entry, and passes (in a request header) the information describing the functions of the device driver.
- Until the request header is marked "done" (see description of done bit in the section, Status Word), MS-DOS continues to make far calls to the interrupt entry point with no parameters.

This set of operations allows you to program an interrupt-driven device driver, for example, to perform local buffering in a printer.

THE REQUEST HEADER

When MS-DOS calls a device driver to perform a function, it passes a request header in ES:BX to the strategy entry point. This is a fixed length header, followed by data pertinent to the operation being performed. Note that it is the device driver's responsibility to preserve the machine state (for example, save all registers on entry and restore them on exit). There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver should set up its own stack.

Figure 11-2 illustrates the format of request header.

BYTE length of record Less the length of this request header
BYTE unit code The subunit operation is for the minor device (no meaning on character devices).
BYTE command code
WORD status
8 bytes reserved here for two DWORD links. One will be a link for MS-DOS queue, the other for the device queue.

Figure 11-2 Request Header Format

Unit Code

The unit code field identifies which unit in your device driver the request is for. For example, if your device driver has 3 units, then the possible values of the unit code field would be 0, 1, and 2.

Command Code Field

The command code field in the request header can have one of the values shown in Table 11-1.

Table 11-1 Request Header Command Codes

COMMAND CODE	FUNCTION
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB (Block only, NOP for character)
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS (Char devs only)
7	INPUT FLUSH (Char devs only)

Table 11-1 Request Header Command Codes (cont'd)

COMMAND CODE	FUNCTION
8	OUTPUT (write)
9	OUTPUT (write) with verify
10	OUTPUT STATUS
11	OUTPUT FLUSH
12	IOCTL OUTPUT (Only called if device has IOCTL)

MEDIA CHECK and BUILD BPB are used with block devices only.

MS-DOS first calls MEDIA CHECK for a drive unit. Then, MS-DOS passes its current media descriptor byte. (Refer to the section BUILD BPB (BIOS Parameter Block) for information on the media descriptor byte.) MEDIA CHECK returns the following status information:

- Media not changed. Current BPB and media byte are OK.
- Media changed. Current BPB and media are wrong. MS-DOS invalidates any buffers for this unit and calls the device driver to build the BPB with the media byte and buffer.
- Not sure. If there are dirty buffers (buffers with changed data, not yet written to disk) for this unit, MS-DOS assumes the BPB and media byte are OK (media not changed). If buffers are clean, MS-DOS assumes the media has changed. It invalidates any buffers for the unit and calls the device driver to build the BPB with the media byte and buffer.

If an error occurs, MS-DOS sets the error code accordingly.

MS-DOS will call BUILD BPB under the following conditions:

- if "media changed" is returned
- if "not sure" is returned, and there are no dirty buffers.

The BUILD BPB call also gets a pointer to a one-sector buffer. What this buffer contains is determined by the NON IBM FORMAT bit in the attribute field. If the bit is zero (device is IBM format-compatible), then the buffer contains the first sector of the first File Allocation Table (FAT). The FAT ID byte is the first byte of this buffer. Note that the BPB must be the same, as far as location of the FAT is concerned, for all possible media because this first FAT sector must be read before the actual BPB is returned. If the NON IBM FORMAT bit is set, then the pointer points to one sector of scratch space (which may be used for anything).

Status Word

The following illustration represents the format of the status word in the request header.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  ┌──────────────────────────────────────────────────┐
  E              B   D                               │
  R  RESERVED    U   O   ERROR CODE (bit 15 on)      │
  R              S   N                               │
  └──────────────────────────────────────────────────┘
```

The status word is zero on entry and is set by the driver interrupt routine on return.

Bit 8 is the done bit. When set, it means the operation is complete. For MS-DOS 2.0, the driver sets it to 1 when it exits.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error. The error can be any of the following:

	0	Write protect violation
(NEW)	1	Unknown unit
	2	Drive not ready
(NEW)	3	Unknown command
	4	CRC error
(NEW)	5	Bad drive request structure length
	6	Seek error
(NEW)	7	Unknown media
	8	Sector not found
(NEW)	9	Printer out of paper
	A	Write fault
(NEW)	B	Read fault
	C	General failure

Bit 9 is the busy bit, which is set only by status calls.

For output on character devices:

> If bit 9 is 1 on return, a write request (if made) waits for completion of a current request.
>
> If it is 0, there is no current request, and a write request (if made) starts immediately.

For input on character devices with a buffer:

> If bit 9 is 1 on return, a read request (if made) goes to the physical device.
>
> If this bit is 0 on return, then there are characters in the device buffer and a read returns quickly. A 1 for bit 9 also indicates that you have typed something. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0, so that MS-DOS will not continuously wait for something to get into a buffer that does not exist.

One of the functions defined for each character and block device is INIT. This routine is called only once when the device is installed. The INIT routine returns a location (DS:DX), which is a pointer to the first free byte of memory after the device driver (similar to the Terminate and Stay Resident interrupt). This pointer method can be used to delete initialization code that is only needed once, thus saving on space.

In addition to a first free byte pointer, block devices return other information.

- The number of units determining logical device names. If the current maximum logical device letter is F at the time of the install call, and the INIT routine returns 4 as the number of units, then they will have logical names G, H, I and J. This mapping is determined by the position of the driver in the device list, and by the number of units on the device (stored in the first byte of the device name field).
- A pointer to a BPB pointer array. There is one table for each unit defined. These blocks will be used to build an internal DOS data structure for each of the units. The pointer passed to the DOS from the driver points to an array of n word pointers to BPBs, where n is the number of units defined. In this way, if all units are the same, all of the pointers can point to the same BPB, saving space. Note that this array must be protected (below the free pointer

set by the return) since the internal DOS structure will be built starting at the byte pointed to by the free pointer. The sector size defined must be less than or equal to the maximum sector size defined at default I/O System INIT time. If it isn't, the install will fail.

- The last item that INIT of a block device must pass back is the media descriptor byte. This byte means nothing to MS-DOS, but it is passed to devices so that they know what parameters MS-DOS is currently using for a particular drive unit.

Block devices may take several approaches; they may be dumb or smart. A dumb device defines a unit (and therefore an internal DOS structure) for each possible media drive combination. For example, unit 0 = drive 0 (single-sided), unit 1 = drive 0 (double-sided). For this approach, media descriptor bytes do not mean anything. A smart device allows multiple media per unit. In this case, the BPB table returned at INIT must define space large enough to accommodate the largest possible media supported. Smart drivers will use the media descriptor byte to pass information about what media is currently in a unit.

FUNCTION CALL PARAMETERS

All strategy routines are called with ES:BX pointing to the request header. The interrupt routines get the pointer to the request header from the queue that the strategy routines store them in. The command code in the request header tells the driver which function to perform.

NOTE

All DWORD pointers are stored with the offset first, then the segment.

INIT

Command code = 0

INIT - ES:BX ->

13-BYTE request header
BYTE # of units
DWORD break address
DWORD pointer to BPB array (Not set by character devices)

The number of units, break address, and BPB pointer are set by the driver. On entry, the DWORD that is to be set to the BPB array (on block devices) points to the character after the "=" on the line in the Configuration File (CONFIG.SYS) that loaded this device. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

NOTE

If there are multiple device drivers in a single .COM file, the ending address returned by the last INIT called will be the one MS-DOS uses. It is recommended that all of the device drivers in a single .COM file return the same ending address.

MEDIA CHECK

Command Code = 1

MEDIA CHECK - ES:BX ->

13-BYTE request header
BYTE media descriptor from DPB (Disk Parameter Block)
BYTE returned

In addition to the status word, the driver must set the return byte to one of the following:

-1	Media has been changed.
0	Don't know if media has been changed.
1	Media has not been changed.

If the driver can return -1 or 1 (by having a door-lock or other interlock mechanism), MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT for each directory access.

BUILD BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

13-BYTE request header
BYTE media descriptor from DPB (Disk Parameter Block)
DWORD transfer address (Points to one sector worth of scratch space or first sector of FAT depending on the value of the NON IBM FORMAT bit)
DWORD pointer to BPB

If the NON IBM FORMAT bit of the drvice is set, then the DWORD transfer address points to a one sector buffer, which can be used for any purpose. If the NON IBM FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

If the IBM compatible format is used (NON IBM FORMAT BIT = 0), then the first sector of the first FAT must be located at the same sector on all media. This is because the FAT sector will be read before the media is actually determined. Use this mode if all you want is to read the FAT ID byte.

In addition to setting status word, the driver must set the pointer to the BPB on return.

In order to allow for many different OEMs to read each other's disks, it is suggested you keep the information relating to the BPB for a particular piece of media in the boot sector for the medium. The format of the boot sector is illustrated in Figure 11-3. The information on bytes per sector and sectors per track are part of the Disk Parameter Block (CPB).

3 BYTE near JUMP to boot code
8 BYTES OEM name and version

WORD bytes per sector

B P B	BYTE sectors per allocation unit
	WORD reserved sectors
	BYTE number of FATs
	WORD number of root dir entries
	WORD number of sectors in logical image
B P B	BYTE media descriptor
	WORD number of FAT sectors

WORD sectors per track

WORD number of heads
WORD number of hidden sectors

Figure 11-3 Boot Sector Format

The three words at the end (sectors per track, number of heads, and number of hidden sectors) are optional. They are intended to help the I/O System identify the attributes of the medium. Sectors per track may be redundant (could be calculated from total size of the disk). Number of heads is useful for supporting different multi-head drives that have the same storage capacity but different numbers of surfaces. Number of hidden sectors may be used to support drive-partitioning schemes.

Currently, the media descriptor byte has been defined for two media types:

5¼-inch disks

Flag bits:

01H - on --> double-sided
All other bits must be on.

8-inch disks

FFG - APC format: single-sided, single-density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 1 reserved sector, 2 FATs, 68 directory entries, 77*26 sectors

For the IBM 3740, FEH is assigned and used for the same format.

FDH - IBM 3740 format, single-sided, single-density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 4 reserved sectors, 2 FATs, 68 directory entries, 77*26 sectors

FEH - double-sided, double-density, 1024 bytes per sector, soft sectored, 1 sector per allocation unit, 1 reserved sector, 2 FATs, 192 directory entries, 77*8*2 sectors

NOTE

The two media descriptor bytes that are the same for 8-inch disks (FEH) is not a misprint. To establish whether a disk is single- or double-density, a read of a single-density address mark should be made. If an error occurs, the media is double-density.

Although these media bytes map directly to FAT ID bytes (which are constrained to the 8 values F8-FF), media bytes can, in general, be any value in the range 0-FF.

READ or WRITE

Command codes = 3,4,8,9, and 12

READ or WRITE - ES:BX (Including IOCTL) ->

13-BYTE request header
BYTE media descriptor from DPB (Disk Parameter Block)
DWORD transfer address
WORD byte/sector count
WORD starting sector number
(Ignored on character devices)

In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must correctly set the return sector (byte) count to the actual number of bytes transferred.

The following applies to block device drivers:

Under certain circumstances the I/O System may be asked to perform a write operation of 64K bytes, which seems to be a "wrap around" of the transfer address in the I/O packet. This request arises due to an optimization added to the write code in MS-DOS. It will only manifest on user writes that are within a sector size of 64K bytes on files "growing" past the current EOF. It is allowable for the I/O System to ignore the balance of the write that wraps around if it so chooses. For example, a write of 10000H bytes worth of sectors with a transfer address of XXX:1 could ignore the last two bytes. A user program can never request an I/O of more than FFFFH bytes and cannot wrap around (even to 0) in the transfer segment. Therefore, the last two bytes can be ignored in this case.

NONDESTRUCTIVE READ NO WAIT

Command code = 5

NONDESTRUCTIVE READ NO WAIT - ES:BX ->

13-BYTE request header
BYTE read from device

If the character device returns busy bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer (hence the term "Non Destructive Read"). Basically, this call allows MS-DOS to look ahead one input character.

STATUS

Command codes = 6 and 10

STATUS Calls - ES:BX ->

13-BYTE request header

All the driver must do is set the status word and the busy bit as follows:

- For output on character devices: If bit 9 is 1 on return, a write request (if made) waits for completion of a current request. If it is 0, there is no current request and a write request (if made) starts immediately.
- For input on character devices with a buffer: A return of I means a read request (if made) goes to the physical device. If it is 0 on return, then there are characters in the devices buffer and a read returns quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0 so that the DOS will not hang waiting for something to get into a buffer that doesn't exist.

FLUSH

Command codes = 7 and 11

FLUSH Calls - ES:BX ->

13-BYTE request header

The FLUSH call tells the driver to flush (terminate) all pending requests. This call is used to flush the input queue on character devices.

THE CLOCK DEVICE

One of the most popular add-on boards is the real time clock board. To allow this board to be integrated into the system for TIME and DATE, there is a special device (determined by the attribute word), called the CLOCK device. The CLOCK device defines and performs functions like any other character device. Most functions will be "set done bit, reset error bit, return." When a read or write to this device occurs, exactly 6 bytes are transferred. The first two bytes are a word, which is the count of days since 1-1-80. The third byte is minutes, the fourth is hours, the fifth is hundredths of seconds, and the sixth is seconds. Reading the CLOCK device gets the date and time; writing to it sets the date and time.

MS-DOS 2.0 FILE ALLOCATION TABLE FORMAT

This section explains how MS-DOS uses the File Allocation Table (FAT) to convert the clusters of a file to logical sector numbers. The device driver is then responsible for locating the logical sector on disk. This information should not be used for any other purpose. System utilities should use the MS-DOS file management functions rather than interpret the FAT.

The File Allocation Table is used by MS-DOS to allocate disk space for a file, one cluster at a time. The FAT consists of a 12-bit entry (1.5 bytes) for each cluster on the disk. The first two FAT entries map a portion of the directory. These FAT entries contain indicators of the size and format of the disk. The second and third bytes always contain FFFFH.

The third FAT entry begins the mapping of the data area (cluster 002). Each entry contains three hexadecimal characters, which can be one of the following.

000	The cluster is unused and available.
FF7	The cluster has a bad sector within it. MS-DOS will not allocate such a cluster. The CHKDSK utility counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.
FF8-FFF	The cluster is the last cluster of a file.
XXX	XXX represents any other hexadecimal characters that are the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The File Allocation Table always begins on the first section after the sectors. If the FAT is larger than one sector, the sectors are contiguous. Two copies of the FAT are usually written, one following the other, for integrity. The FAT is read into one of the MS-DOS buffers whenever needed (open, allocate more space, reserved on the disk for MS-DOS and so on). For performance reasons, this buffer is given a high priority to keep it in memory as long as possible.

HOW TO USE THE FILE ALLOCATION TABLE

Obtain the starting cluster of the file whose cluster is to be changed to a logical sector from the directory entry.

Now, to locate each subsequent cluster of the file:

1. Multiply the cluster number just used by 1.5. (Each FAT entry is 1.5 bytes long.)
2. The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

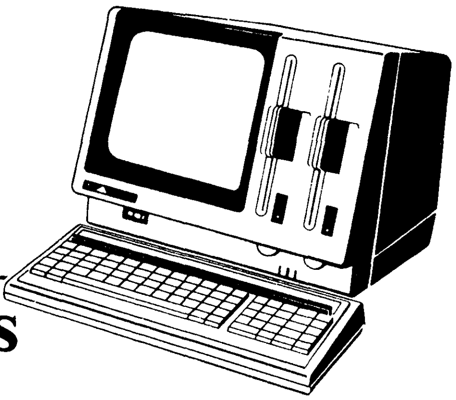
3. Use a MOV instruction to move the word at the calculated FAT offset into a register.
4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF. Otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.
5. If the resultant 12 bits are FF8H-FFFH, there are no more clusters in the file. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.
2. Multiply the result by the number of sectors per cluster.
3. Add the logical sector number of the beginning of the data area.

DEVICE DRIVER LOGIC EXAMPLES

For examples of device driver logic, refer to the assembly listings of the I/O System supplied on the MS-DOS listing diskette.



Appendix A
The ASCII Character Codes

Table A-1 lists the standard ASCII character codes with the corresponding decimal and hexadecimal values for reference.

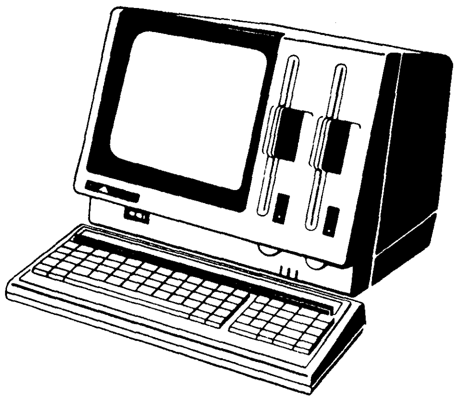
Table A-1 ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	'	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	
009	09H	HT	052	34H	4	095	5FH	-
010	0AH	LF	053	35H	5	096	60H	'
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH		103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH		015	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n

Table A-1 **ASCII Character Codes** (cont'd)

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	
038	26H	&	081	51H	Q	124	7CH	
039	27H	'	082	52H	R	125	7DH	
040	28H	(	083	53H	S	126	7EH	
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

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



Appendix B

APC Keyboard and Display Codes

The APC keyboard generates standard ASCII and other codes specific to the APC when keys are pressed.

Table B-1 shows the ASCII and APC graphic display characters, along with their corresponding decimal and hexadecimal codes.

Table B-1 Mnemonic Control Codes and ROM Generator Display Characters

SECOND HEX DIGIT	FIRST HEX DIGIT															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 00	DLE 16	SP 32	0 48	@ 64	P 80	. 96	p 112	128	144	160	∞ 176	α 192	¢ 208	224	240
1	SOH 01	DC1 17	! 33	1 49	A 65	Q 81	a 97	q 113	129	145	≥ 161	3 177	ν 193	ω 209	225	241
2	STX 02	DC2 18	" 34	2 50	B 66	R 82	b 98	r 114	130	146	• 162	τ 178	Δ 194	≈ 210	226	242
3	ETX 03	DC3 19	# 35	3 51	C 67	S 83	c 99	s 115	131	147	* 163	4 179	β 195	γ 211	227	243
4	EOT 04	DC4 20	$ 36	4 52	D 68	T 84	d 100	t 116	132	148	≤ 164	5 180	ξ 196	7 212	228	244
5	ENQ 05	NAK 21	% 37	5 53	E 69	U 85	e 101	u 117	133	149	/ 165	6 181	η 197	8 213	229	245
6	ACK 06	SYN 22	& 38	6 54	F 70	V 86	f 102	v 118	134	150	• 166	ϵ 182	θ 198	9 214	230	246
7	BEL 07	ETB 23	' 39	7 55	G 71	W 87	g 103	w 119	135	151	↑ 167	ρ 183	1 199	ι 215	231	247
8	BS 08	CAN 24	(40	8 56	H 72	X 88	h 104	x 120	136	152	½ 168	σ 184	± 200	ϕ 216	♠ 232	⌠ 248
9	HT 09	EM 25	) 41	9 57	I 73	Y 89	i 105	y 121	137	153	↓ 169	ψ 185	υ 201	≠ 217	♥ 233	⌡ 249
A	LF 10	SUB 26	* 42	: 58	J 74	Z 90	j 106	z 122	138	154	← 170	Ω 186	π 202	χ 218	♦ 234	250
B	VT 11	ESC 27	+ 43	; 59	K 75	[91	k 107	{ 123	139	155	→ 171	Γ 187	Λ 203	° 219	♣ 235	251
C	FF 12	FS 28	, 44	< 60	L 76	\ 92	l 108	¦ 124	140	156	+ 172	0 188	2 204	Φ 220	● 236	252
D	CR 13	GS 29	— 45	= 61	M 77	] 93	m 109	} 125	141	157	(173	ο 189	⊕ 205	ζ 221	○ 237	253
E	SO 14	RS 30	. 46	> 62	N 78	∧ 94	n 110	~ 126	142	158	) 174	κ 190	— 206	λ 222	238	254
F	SI 15	US 31	/ 47	? 63	O 79	— 95	o 111	DEL 127	143	159	¼ 175	Σ 191	- 207	μ 223	239	255

ASCII Character or Graphics Character → □ ← Decimal Code

ASCII codes NUL through US and DEL are the standard ASCII control codes. ASCII hexadecimal codes 20 through 7E display the characters that make up the APC default character set (also ASCII). The hexadecimal codes 80 to FF have been designated as graphics characters.

Table B-2 identifies the functions performed by the standard ASCII control codes. As noted in the table, some of the codes perform no operations on the APC itself, but can be used in data communications with other devices.

Table B-2 ASCII Character Mnemonics and Functions

MNEMONIC	ASCII DEFINITION	APC FUNCTION
NUL	Null	
SOH	Start of Heading	
STX	Start Text	
ETX	End Text	
EOT	End of Transmission	
ENQ	Enquiry	
ACK	Acknowledge	
BEL	Bell	Sounds Beep
BS	Backspace	Backspace
HT	Horizontal Tab	Tab
LF	Line Feed	Line Feed (cursor down)
VT	Vertical Tab	Cursor Up
FF	Form Feed	Cursor Forward
CR	Carriage Return	Carriage Return
SO	Shift Out	
SI	Shift In	
DLE	Data Link Escape	
DC1	Device Control 1	
DC2	Device Control 2	
DC3	Device Control 3	
DC4	Device Control 4	
NAK	Negative Acknowledge	
SYN	Synchronous Idle	
ETB	End Transmission Block	
CAN	Cancel	
EM	End of Medium	
SUB	Substitute	Clear Screen

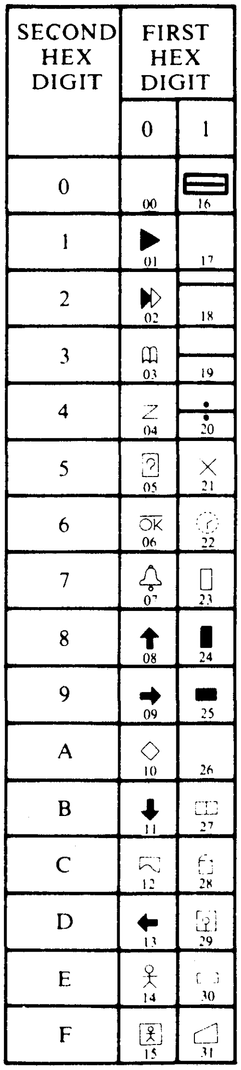
Table B-2 ASCII Character Mnemonics and Functions (cont'd)

MNEMONIC	ASCII DEFINITION	APC FUNCTION
ESC	Escape	Begin Escape Sequence
FS	Form Separator	
GS	Group Separator	
RS	Record Separator	
US	Unit Separator	
SP	Space	
DEL	Delete	

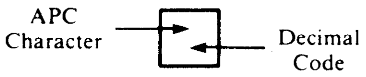
NOTE: The ASCII mnemonics for which no functions are given are not used on the APC.

Certain ASCII codes in the range 00H to 1FH have also been assigned special characters to display. Others are associated with a specific APC function. Table B-3 identifies these codes. For the functions assigned to some of the control codes shown in the table, refer to Table B-2.

Table B-3 APC Control Character Font



SECOND HEX DIGIT	FIRST HEX DIGIT 0	FIRST HEX DIGIT 1
0	00	⊟ 16
1	▶ 01	17
2	▶▷ 02	18
3	📖 03	19
4	Z 04	⋮ 20
5	⍰ 05	× 21
6	O̅K 06	⍰ 22
7	🔔 07	▯ 23
8	⬆ 08	▮ 24
9	➡ 09	▬ 25
A	◇ 10	26
B	⬇ 11	⬚ 27
C	12	28
D	⬅ 13	29
E	🯅 14	30
F	15	31



NOTE: Only characters that are not associated with a specific APC function are displayed on the screen.

In the next few pages, figures illustrate the APC keyboard as it appears to the user, and the GRPH1 and GRPH2 key assignments made to them.

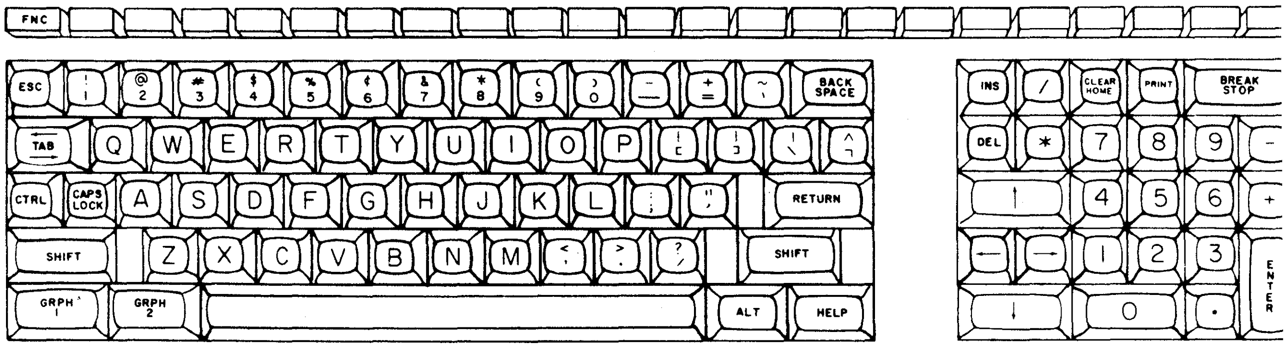
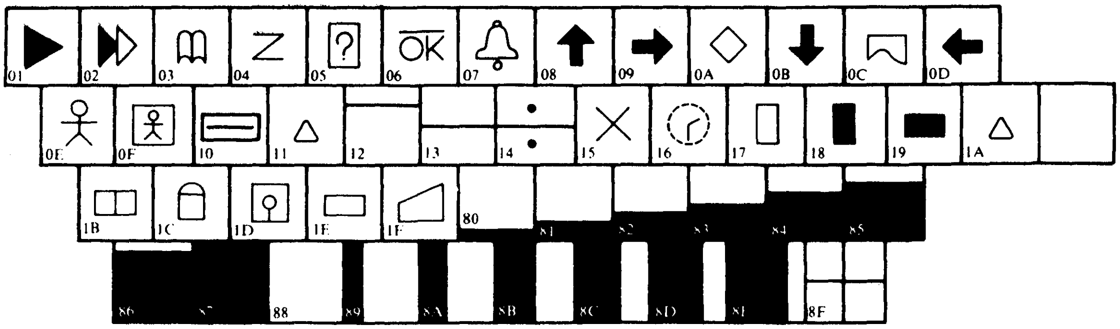
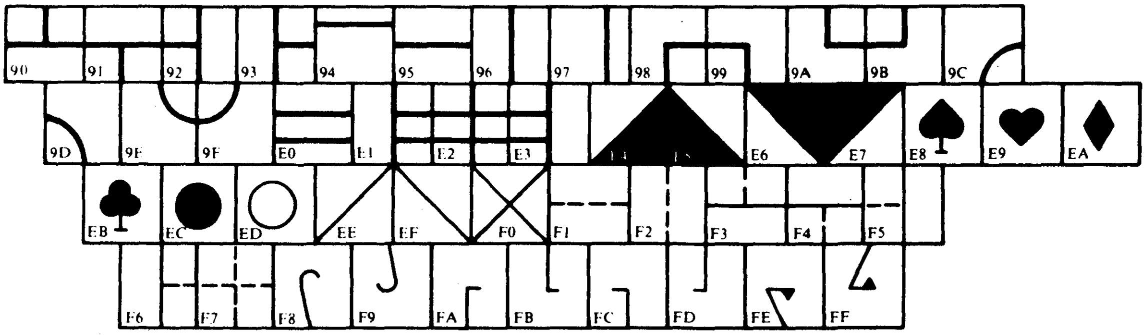


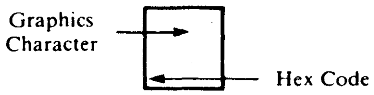
Figure B-1 The APC Keyboard



NOTE: Characters associated with a specific APC function are not displayed.

A. UNSHIFTED (SHIFT KEY UP)





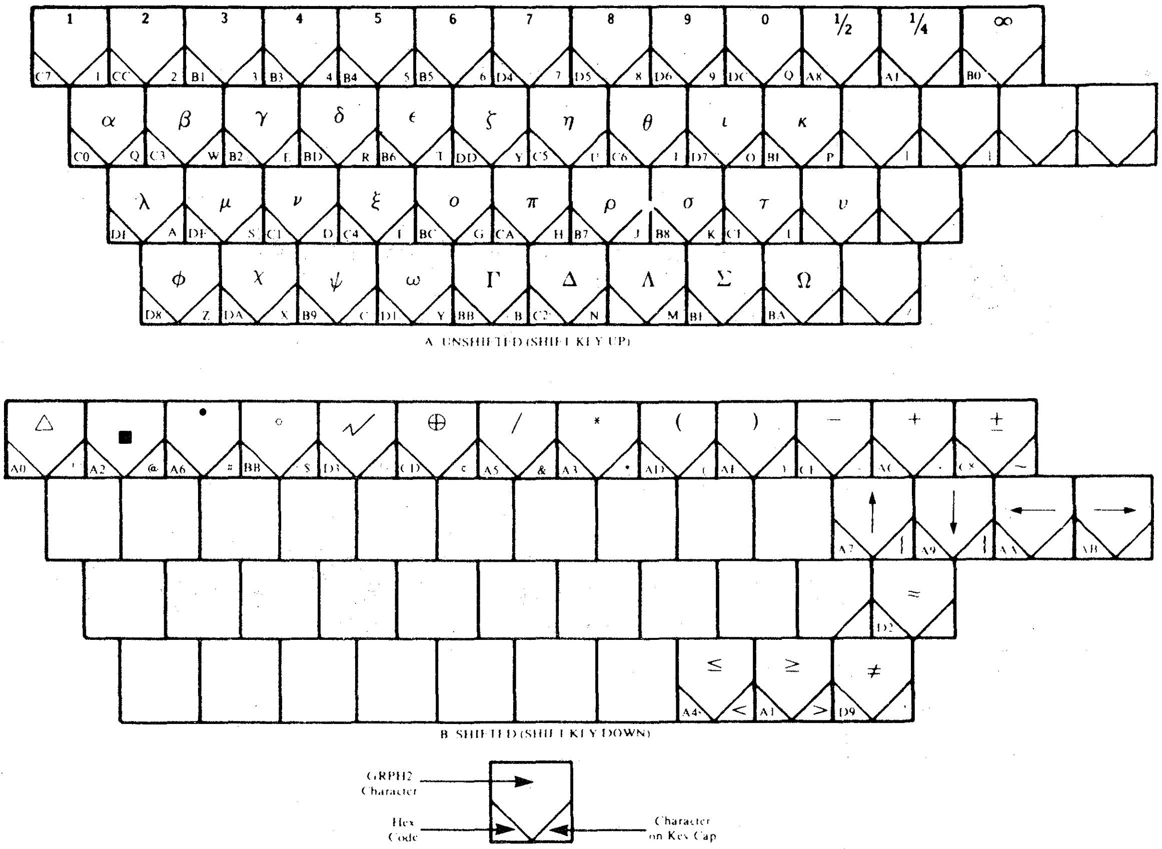
B. SHIFTED (SHIFT KEY DOWN)

NOTES:
1 GRPH1 CHARACTERS ARE PRODUCED WHEN THE GRPH1 KEY IS PRESSED.

2 GRAPHICS SYMBOLS ASSOCIATED WITH A SPECIFIC APC FUNCTION ARE NOT DISPLAYED ON THE SCREEN. INSTEAD, THE FUNCTION IS PERFORMED.

3 THE ALPHANUMERIC SYMBOLS ASSOCIATED WITH THE GRAPHIC SYMBOLS ARE THE HEXADECIMAL (HEX) CODES GENERATED BY PRESSING THE KEYS.

Figure B-2 The APC GRPH1 Characters


1 C7 1 · 2 CC 2 · 3 B1 3 · 4 B3 4 · 5 B4 5 · 6 B5 6 · 7 D4 7 · 8 D5 8 · 9 D6 9 · 0 DC 0 · ½ A8 · ¼ AF · ∞ B0
α C0 Q · β C3 W · γ B2 E · δ BD R · ε B6 T · ζ DD Y · η C5 U · θ C6 I · ι D7 O · κ BE P
λ DE A · μ DF S · ν C1 D · ξ C4 F · ο BC G · π CA H · ρ B7 J · σ B8 K · τ CF L · υ
φ D8 Z · χ DA X · ψ B9 C · ω D1 V · Γ BB B · Δ C2 N · Λ M · Σ BE · Ω BA
A UNSHIFTED (SHIFT KEY UP)
△ A0 ! · ■ A2 @ · • A6 # · ∘ BB $ · √ D3 % · ⊕ CD ¢ · / A5 & · * A3 * · (AD (·) AE) · − CE _ · + AC + · ± C8 ~
↑ A7 · ↓ A9 · ← AA · → AB
≈ D2
≤ A4 < · ≥ A1 > · ≠ D9
B SHIFTED (SHIFT KEY DOWN)
GRPH2 Character
Hex Code
Character on Key Cap

NOTE: GRPH2 CHARACTERS ARE PRODUCED WHEN THE GRPH2 KEY IS PRESSED

Figure B-3 The APC GRPH2 Characters

Advanced Personal Computer

NEC
NEC Information Systems, Inc.

USER'S COMMENTS FORM

Document: MS-DOS System Programmer's Guide

Document No.: 819-000104-3001 Rev. 00

Please suggest improvements to this manual.

Please list any errors in this manual. Specify by page.

From:

Name

Title

Company

Address

Dealer Name

Date:

Please cut along this line.

Seal or tape all edges for mailing-do not use staples.

FOLD HERE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 386 LEXINGTON, MA

POSTAGE WILL BE PAID BY ADDRESSEE

NEC Information Systems, Inc.
Dept: Publications
1414 Mass. Ave.
Boxborough, MA 01719

FOLD HERE

Seal or tape all edges for mailing-do not use staples.