MP/AOS

Macroassembler, Binder, and Library Utilities

DataGeneral

MP/AOS

Macroassembler, Binder and Library Utilities

Data General

NOTICE

Ordering No. 069-400210

Printed in the United States of America
Rev. 00, July 1982

Preface

This manual is in three parts.

Part 1, on the macroassembler, describes how to use this language processing program. First, it briefly reviews the program development and assembly processes. Then, it fully describes syntax; definition of symbols and macros; and assembler organization, control, execution and output. Pseudo-ops are described thematically and summarized in an alphabetically organized dictionary.

Part 2, on the binder, describes how to use the binder program to produce executable programs from object modules. It explains how to specify object modules, overlays, and library files as input; how to select output; and how to interpret printed output from the binder. This section also includes a thorough description of object block formats, which are necessary for the construction of system utitities such as assemblers and compilers.

Part 3, on the library file editor and symbol cross-reference analyzer, describes two programs that enable you to manage efficiently any number of object modules.

Command-Line Conventions

Throughout this manual, we use the following conventions to illustrate command-line formats:

COMMAND — Upper-case letters indicate a command mnemonic. Code this mnemonic into your program exactly as it appears in the command line.

argument — Lower-case italics represent an argument. Replace this symbol with the exact code for the argument you need.

{*optional*} — Curly brackets indicate an optional argument. (Command switches also appear in this format.) Do not include the brackets in your code; they only set off the option.

{arg_1 | arg_2} — A vertical bar indicates a choice of arguments (either arg_1 or arg_2).

Related Manuals

The following manuals also belong to the series of books published on the MP/AOS operating system.

MP/AOS Concepts and Facilities (DGC No. 069-400200) provides a concise but thorough introduction to the MP/AOS operating system for users who want to assess the system's advantages.

MP/AOS System Programmer's Reference (DGC No. 093-400051) documents MP/AOS system software and provides a complete dictionary of system calls and library routines.

MP/AOS Command Line Interpreter (CLI) (DGC No.069-400201) describes the interactive CLI program, the user's main interface to the MP/AOS system. A command dictionary provides command descriptions, formats, and examples.

Loading MP/AOS (DGC No. 069-400207) describes how to install MP/AOS software on ECLIPSE-line computers and how to load tailored systems.

MP/AOS System Generation and Related Utilities (DGC No. 069-400206) describes the generation of an MP/AOS system tailored to specific applications. It also describes the following utilities, providing sample dialogues as appropriate:

- SYSGEN, the interactive system generation utility;
- DINIT, the disk initializer;
- FIXUP, the disk repair utility;
- SPOOLER, which controls line printer operations;
- ELOG (error logger), the utility for interpreting the system log file.

MP/AOS Debugger and Performance Monitoring Utilities (DGC No. 069-400205) describes the following utilities, providing a dictionary of debugger commands and sample dialogues as appropriate:

- FLIT, the process debugger;
- PROFILE, which measures execution-time performance;
- OPM, the process monitor that displays current system resource allocation and status.

MP/AOS Advanced Program Development Utilities (DGC 069-400208) describes the following utilities:

- Text control system (TCS), a method for managing different versions of a single file,
- BUILD, an aid in creating a new version of a file from previously exising files,
- FIND, which locates occurrences of strings in text files.

MP/AOS SPEED Text Editor (DGC No. 069-400202) documents the features of SPEED, the MP/AOS character-oriented text editor.

MP/AOS SLATE Text Editor (DGC 069-400209) documents the features of SLATE, a screen- and line-oriented text editor.

MP/AOS File Utilities (DGC No. 069-400204) describes the following utility programs, providing sample dialogues for each:

- FEDIT, a file editor that permits modification of system files, including program and data files;
- FDISP, which can display the address and data contents of a file or compare two files, displaying the parts that differ;
- SCMP, which can compare two source programs line by line;
- MOVE, which allows the transfer of files among directories;
- AOSMIC, which allows manipulation of MP/AOS and MP/OS disks and files on an AOS system;
- FOXFIRE, which permits the transfer of files among MP/OS, MP/AOS, and AOS systems over asynchronous communication lines.

SP/Pascal Programmer's Reference (DGC No. 069-400203) documents an extended Pascal for system programmers. SP/Pascal has all of the features of MP/Pascal as well as special features targeted for the MP/AOS operating systems.

Books on three additional programming languages supported by MP/AOS have previously been published as part of the bookset for the MP/OS operating system:

MP/Pascal Programmer's Reference (DGC No. 069-400031) documents for system programmers a Pascal-based language targeted for the MP/OS operating system.

MP/FORTRAN IV Programmer's Reference (DGC No. 069-400033) documents for system programmers a language based on ANSI 1966 standard FORTRAN with extensions.

MP/Basic Programmer's Reference (DGC No. 069-400032) documents for new users a programming language based on ANSI standard Basic with extensions.

MP/OS

For information on Microproducts and a bibliography of documentation on the Microproducts line, see *Introduction to Microproducts,* DGC No. 014-000685.

For information on cross development between MP/OS and MP/AOS, see *MP/OS System Programmer's Reference,* DGC No. 093-400001.

Contents

Figures

Part 1

Part 2

Part 3

Appendices

Tables

Part 1

Part 2

Part 3

Appendices

The Macroassembler

Part

1

Review of the Macroassembler

This chapter begins by reviewing the steps involved in writing and running a program. It then describes the role of the macroassembler and how it relates to the organization of memory.

Program Development

There are five steps in developing an assembly language program:

1. Writing and editing the program. Normally, you enter the program from a console using one of Data General's text editors. The program in ASCII format is called a *source module* and, when stored on disk, a *source file.* By convention, source filenames usually end with the .SR extension.
2. Assembling the program. The macroassembler produces the *object module,* stored in a new file, called the *object file.* Object filenames end with the .OB extension. You must eliminate *assembly errors* from the source module before proceeding to the next step, which transforms the object file(s) into an executable program.
3. Binding the program. The binder transforms one or more object modules into an executable program. Output from the binder is called the *program file,* and its name ends with a .PR extension.
4. Executing the program. You execute the program by typing

   XEQ *program-filename* ↵

   at the console. When the program performs as you intend, this is the last step in the procedure.
5. Debugging the program. Two general types of errors may occur when you run your program: *runtime errors,* which the operating system detects, and errors in the logic of the program. To find the cause of either type of error, you can use the source listing and MP/AOS debugger (see *MP/AOS Debugger and Performance Monitoring Utilities,* DGC No. 069-400205).

Role of the Macroassembler

The role of the assembler is to process the information in the source module in such a way that the binder can use it to produce an executable program file. A brief preview of the actions of the binder may help you understand the nature of the information that the assembler incorporates into the object module.

As explained in *MP/AOS System Programmer's Reference* (DGC No. 093-400051), MP/AOS programs run in a *process space* of 32K addresses. Similarly, programs occupy a *logical address space* of 32K addresses. When a program is running, address 0 of the program's address space uses address 0 of the process's address space, address 1 uses address 1, and so forth. Hence, a process is defined as the system representation of a program, in some phase of execution, with a unique identifier and a separate address space.

Addresses in a program's or a process's address space are called absolute addresses. These addresses are absolute in the sense that they are unique and well-defined, but they are not physical addresses. For example, each process has an absolute location 100 distinct from location 100 in any other process. Each time the program is run, the program may place the contents of location 100 in a different physical address.

The process space is divided into three main areas.

- Lower page zero consists of addresses in the range 0-377_8.
- The *pure area* contains portions of the program, usually instructions, that are not modified during execution. The range of addresses in this area differs from program to program.
- The *impure area* contains portions of the program, usually data, that may be altered during execution. The range of addresses in this area differs from program to program.

The function of the binder, then, is to use information provided by the assembler to assign to each 16-bit word in the program file an absolute address in one of these process areas.

To provide the information for the binder, the assembler translates each line in the source program into one or more 16-bit words, calculates an address for each word, and places the words and their addresses in the object module. (Object module formats are discussed in detail in Part 2, Chapter 4.)

Each address assigned by the assembler consist of a *relocation base* and an offset from the relocation base. The assembler has a *location counter* associated with each relocation base. You control the assignment of relocation bases by incorporating assembler directives into your program. These directives determine the process area—pure, impure, or lower page zero—into which the translation of each line of your source program will be placed. The assembler calculates the offset part of the address by incrementing the appropriate location counter.

The addresses produced by the assembler are either absolute addresses or relative, *relocatable* addresses.

- Absolute addresses are not modified by the binder. The information in the relocation base and offset fully specifies the actual absolute location that the word will occupy in the program. Absolute addresses are most used to specify locations in lower page 0 with hardware-defined functions.
- Relocatable addresses determine the area that the word will reside in, but not the absolute address within that area. The binder computes the absolute address from the relocatable address.

The assembler has a location counter for each of four relocation bases. These bases can be thought of as four assembler partitions, as follows:

Partition	Type of Address and Code
Absolute	Non-relocatable (pure or impure code)
ZREL	Lower page zero, relocatable (impure code)
NREL 0	Normal relocatable (impure code)
NREL 1	Normal relocatable (pure code)

Chapters 3 and 4 contain further information on relocation rules and relocation bases or partitions.

The binder may combine more than one object module in a single program file. Names used in more than one module are *global symbols.* (Those used in only one module are *local symbols.* In addition to using global symbols, modules may communicate by sharing *common areas* which may be either named or unnamed. You use assembler directives explained in Chapter 4 for intermodule communication.

When you execute the program, the operating system places the program file in memory, translating the logical addresses in the program file to physical addresses in memory. Figure 1.1 illustrates the stages involved in converting an assembly language source module (or modules) into a running program.

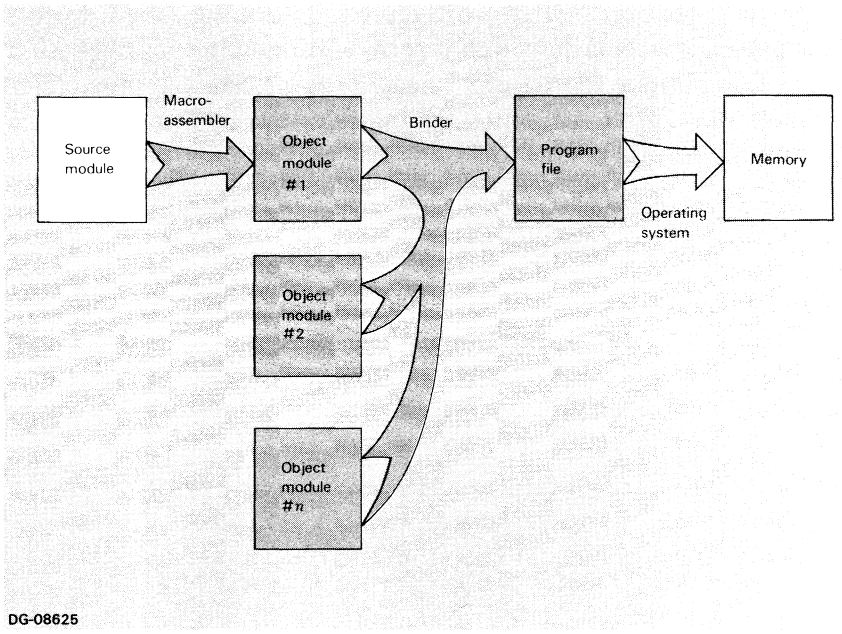


Figure 1.1 From source program to memory

Assembler Output

In addition to the object module, the assembler produces two kinds of printed output to enable you to see that your source program has been assembled as you intended. These are the *source listing* and the *error listing*.

The source listing has two sections: the *assembly listing* matches each line of the source program against the assembled (binary) version, so that you may see how each line was interpreted by the assembler; the *cross-reference listing* helps you find symbols that you define in the source program.

The *error listing* lists lines in your program that the assembler was either unable to interpret or recognized as logically inconsistent.

Finally, you may instruct the assembler to construct a *permanent symbol table* to allow you to re-use symbols defined in one assembly during another assembly. This file is not printable.

Figure 1.2 illustrates the several outputs from the assembler. More details and samples of the various listings are given in Chapter 6.

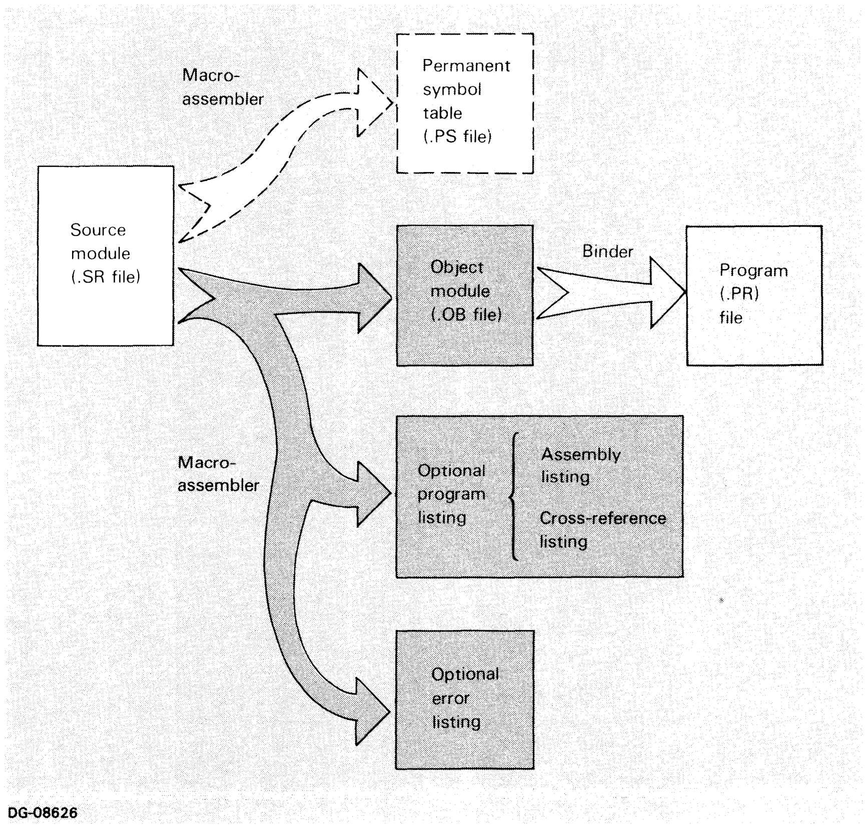


Figure 1.2 Assembler outputs

The Assembly Process

This chapter gives an overview of the actual assembly process. But first, it briefly reviews the elements of your input to the assembler. This input is in the form of one or more assembly language source modules (source files). The various outputs from the assembler are described and illustrated in Chapter 6.

The Source Program

An assembly language source module consists of a series of source lines or statements. A source statement is a sequence of ASCII characters terminated by an end-of-line character (New Line, Form Feed, or Carriage Return). Below, is a summary of source statements in terms of their format, functions (type of statement), and components.

Statement Format

In general, all source statements in your module should adhere to the following format:

```
label: statement body ;comment ↲
```

Each nonblank line in your source module must contain a value for at least one of these three fields.

Statement Label

A label symbolically names a memory location. By using labels, you can refer to locations without regard for numeric addresses.

If a source statement has a label, it must appear at the beginning of the source line and must be followed by a colon. For example:

```
BEGIN:  LDA 0,SUM
JUMP:   JMP @17
```

See "Defining Labels," Chapter 5, for further information on the use of labels.

Statement Body

The statement body may contain an assembly language instruction, macro or system call, pseudo-op, assignment, or data.

These five kinds of statement body are discussed below, under "Statement Types."

Comments

You may include comments in your program to facilitate program development, maintenance, and documentation. The assembler does not interpret comments, and therefore comments do not affect the generation of the object module.

Statement Types

The five types of source statements are as follows:

assembly language instructions
macros and system calls
pseudo-ops
assignments
data.

Each statement type must conform to the general statement format shown above. In addition, each of the five statement types also has its own specific syntax (see Chapter 3).

Assembly Language Instructions

Assembly language instructions perform specific operations at execution time. These instructions can be grouped into classes, which include:

I/O instructions are used to communicate with peripheral devices. If, however, your program will run under an operating system, you will generally use I/O system calls instead of I/O instructions to communicate with peripherals.

Memory reference instructions allow you to transfer data between memory and an accumulator or modify the program counter (PC) and/or a location in memory.

Arithmetic and logic (ALC) instructions perform operations on data residing in the accumulators.

Stack instructions allow you to manipulate the ECLIPSE hardware stack.

In addition to using instruction mnemonics that are a part of the ECLIPSE instruction set, you may define your own instruction mnemonics (see Chapter 5).

Macros and System Calls

A macro is a series of assembly language source statements that you assign a name. Whenever you wish to place that section of source code in your source module, you simply enter the macro name; the assembler substitutes the corresponding statements.

Included in the MP/AOS software package is a set of predefined macros, called system calls, that cause the system to perform functions on behalf of user programs. In some instances, such as when performing I/O under an operating system, you must use system calls instead of machine instructions.

Chapter 5 explains how to use macros in your program. System calls are fully documented in *MP/AOS System Programmer's Reference,* DGC No. 093-400051.

Pseudo-Ops

A pseudo-op is a *pseudo instruction* because your program never executes it; rather, the assembler does. Pseudo-ops are used in the construction of the object module, but do not directly correspond to any words in it. Pseudo-ops are either *assembler directives* or *value symbols.*

Assembler directives perform the following functions, among others:

- tell the assembler where in memory each word is to be placed;
- allow separately assembled source modules to communicate with each other;
- define macros.

Value symbols are internal assembler variables. For example, .PASS is a value symbol representing the stage (pass 1 or pass 2) the assembler has reached in assembling a program.

The syntax of pseudo-ops is explained in Chapter 3.

Assignments

An assignment statement associates a relocatable 16-bit value with a symbolic name. After associating a value with a symbol, you may use the symbol any time you wish to indicate the value.

Data

A data statement consists of a single number, symbol, or expression (see "Statement Components," below). When the assembler encounters a data statement, it evaluates the number, symbol, or expression and stores the value in memory.

Examples of data statements are:

```
0
322
32*5
SIX
```

The commercial "at" character (@) may be included in a data statement to indicate indirect addressing. Indirect addressing is explained in the Principles of Operation manual for your ECLIPSE computer. The syntax of @ is explained in Chapter 3.

Statement Components

A source statement consists of one or more syntactic units, called atoms. Each atom is a string of one or more ASCII characters that the macroassembler views as a single entity.

Expressions (made up of numbers, symbols, and operators) are not, strictly speaking, atoms but are considered as such for purposes of this discussion. There are, then, five types of atoms:

terminals and delimiters
numbers
symbols
expressions
instruction modification characters.

Delimiters and Terminals

Delimiters, also called end-of-line characters, separate the source statements in your module. The delimiters are New Line, Form Feed, and Carriage Return. In this manual, a curved down arrow (↲) represents any of these delimiters.

Terminals are characters that separate numbers, symbols, and expressions from each other within a single source statement. In the following statement, the comma, colon and semicolon are terminals:

```
BEGIN: LDA 0,SUM   ;load accumulator
```

Terminals and their use are explained in Chapter 3.

Numbers

In a source statement, a number is an integer or a floating-point constant (fraction and exponent).

The macroassembler recognizes three types of number representation:

- single-precision integer, stored in one word (16 bits);
- double-precision integer, stored in two words (32 bits);
- single-precision floating-point constant, stored in two words (32 bits).

Single-precision integers may appear in expressions and data statements. Double-precision and floating-point constants appear only in data statements.

Formats for number representation are explained in Chapter 3.

Symbols

A symbol is a name assigned to a binary word. Symbols may be defined during one assembly for use during subsequent assemblies. The definitions for re-usable symbols are stored in a *symbol file*. During assembly, the assembler constructs a *symbol table* containing the definitions of all symbols used by your program. Chapter 6 explains the procedures for building a symbol table.

There are three classes of symbols:

- **Permanent symbols** consist solely of pseudo-ops. Conversely, all pseudo-ops are permanent symbols. These can either direct the assembly process or represent internal values. The assembler uses the symbol's context in the source line to determine its intended use. You cannot redefine these symbols.
- **Semi-permanent symbols** are symbols defined using the symbol-definition pseudo-ops described in Chapter 5. These include the

ECLIPSE instruction set mnemonics in the symbol file furnished by Data General, as well as any instructions you have defined in a parameter file or in the source program you are assembling. Instructions for redefining semi-permanent symbols are contained in Chapter 5.

- **Nonpermanent symbols** are labels and symbols defined in assignment statements. Instructions for redefining nonpermanent symbols are contained in Chapter 5.

Expressions

An expression is a series of symbols and/or integers separated by operators. For example, Z+2 is an expression that consists of a symbol and an integer separated by the operator +.

Though an expression is not a single atom, it is heuristically useful to consider expressions as atoms when describing the assembler algorithm.

The use and format of expressions are explained in Chapter 3.

Instruction Modification Characters

The commercial "at" sign (@) is used to indicate indirect addressing in memory reference instructions and to set bit 0 of expression results for data words.

The number sign (#) is used in certain ALC instructions to indicate a no-load operation.

Summary

Figure 2.1 illustrates the three ways of looking at source statements described in this chapter (by format, type, and components).

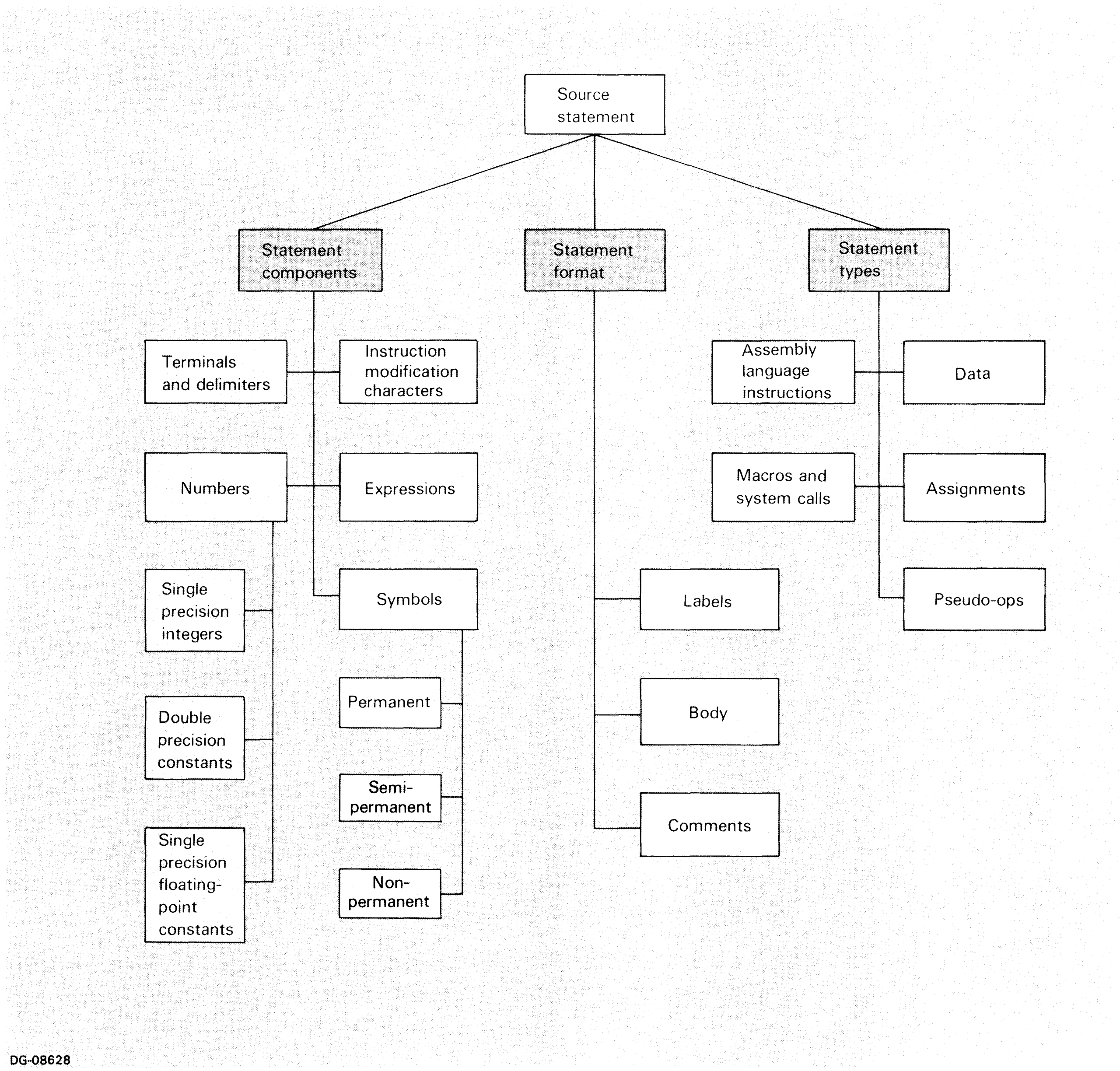


Figure 2.1 The source statement

Assembler Operations

An assembly language source module is a series of ASCII characters grouped into source statements. The macroassembler interprets those statements and produces a binary representation of the source module. The resulting binary module is called an object module.

To produce an object module, the assembler scans the source code twice and is therefore called a two-pass assembler. During its two passes through the source code, the assembler evaluates source statements for validity, interprets symbols, expands macros, assigns relocatable addresses, and performs the calculations and data manipulations needed to produce the object module.

During the first pass, the assembler fills in the symbol table, using the symbol file and whatever symbol definitions appear in your source program. (See Chapter 6 for more about the symbol file). Not all symbols are defined the first time they appear in the source code, as in this example:

```
        A+10
         .
         .
         .
A:3
```

The first time the assembler encounters the symbol A, it places that symbol in the symbol table and marks it as undefined. Later, upon finding that A is a label, the assembler places the value 3 in the symbol table.

The two passes are reflected in the error listing, which contains tallies of "pass 1 errors" and "pass 2 errors." The error listing is fully described in Chapter 6. The assembler allows a program to monitor its own assembly by using the .PASS pseudo-op (see Chapter 4).

The assembler assigns relocatable addresses to the translations of source statements according to assembler directives in the source program. During the second pass, the assembler assigns a relocatable address to each word that you have not assigned an absolute address. All relocatable addresses are expressed in terms of relocation bases and offsets. Chapter 3 explains the rules that the assembler uses when assigning these addresses.

For a better understanding of the actions performed by the assembler, refer to Part 2, Chapter 4, which describes object module formats. General forms of the two-pass algorithm appear in many assembler-design texts.

Syntax and Relocation Rules

This chapter lists the valid assembler character set and describes the syntax for each type of statement that can occur in the source program. It also explains the order of evaluation of operators within expressions and the determination of each expression's relocation property.

Character Set

Each source module consists of a string of ASCII characters. The MP/AOS macroassembler allows you to use the following characters in a source module:

- Upper- and lowercase alphabetic characters: A through Z and a through z. The macroassembler does not distinguish between upper case and lower case, except in text strings and string character constants (see .TXT in Chapter 7 and ''Special Integer-Generating Formats,'' Chapter 3).
- Numerals: 0 through 9
- Symbol name characters: $? _
- Format control and end-of-line characters: Carriage Return, Form Feed, New Line, space, horizontal tab.
- Special characters:

```
"   '   !   &   +   *
/   -   <   >   =   @   #   $
%   ,   _   .   :   ;   ^
\   E   B   D   **   ( )   [ ]
```

The special characters have special meanings to the macroassembler. Table 3.1 lists the meaning of each special character.

Do not use delete (ASCII 177_8), control characters, or characters with the parity bit set to 1 in your source module unless they are in comments or text strings (see .TXT, Chapter 7). If the macroassembler encounters one of these, it returns an error and ignores the illegal character. The null character (ASCII 000_8) is always ignored.

Appendix A lists the octal codes for each ASCII character.

Character	Meaning
: (colon)	Follows all labels
; (semicolon)	Precedes all comments
. (period)	A permanent symbol with the value and relocation property of the current location counter
	Indicates a decimal integer or a floating-point constant
	May appear in symbol names
, (comma)	Delimits arguments
+ (plus sign)	Addition operator
	Unary operator indicating a positive value
— (minus sign)	Subtraction operator
	Unary operator indicating a negative value
* (asterisk)	Multiplication operator
/ (slash)	Division operator
B (upper or lower case)	Single precision bit alignment operator (16 bits)
& (ampersand)	Logical AND operator
! (exclamation point)	Logical OR operator
> (greater than)	Relational operator
< (less than)	Relational operator
== (two equal signs)	Assigns a value to a symbol
= (equal sign)	Assigns a literal value
>= (greater than or equal)	Relational operator
= (less than or equal)	Relational operator
<> (not equal)	Relational operator
' (apostrophe)	Converts two ASCII characters to their octal values
" (quotation mark)	Converts an ASCII character to its octal value
^(uparrow)	Identifies formal arguments in a macro definition string
% (percent)	Terminates a macro definition string
_ (underscore)	Directs the assembler to ignore the special meaning of a character that appears in a macro definition string
	May appear in symbol names
	Break character in number strings
$ (dollar sign)	Generates unique labels within macros.
\ (backslash)	Generates numbers and symbols
D (upper or lower case)	Double precision (32-bit integer) indicator.
E (upper or lower case)	Exponential notation indicator
( ) (parenthesis)	May surround a number, symbol, or expression to alter operator priority
[] (square brackets)	May enclose arguments in a macro call or conditional label
** (double asterisks)	Suppresses listing of the source line
@ (at sign)	Indirect addressing indicator; directs the assembler to place a 1 in the indirect addressing bit
# (number sign)	No-load indicator; directs the assembler to place a 1 in the no-load bit

Table 3.1 Special Characters

Statement Format

Assembly language statements conform to the format

label: statement body   ;comment ↲

Each nonblank line in your source module must contain a value for at least one of these three fields.

If you do not want the source line listed, place double asterisks (**) on the line.

The assembler distinguishes between fields by searching for terminals. Table 3.2 lists the characters that terminate each field in the source statement.

Field	Terminal
Label	Colon (:)
Statement body	Semicolon (;) or line delimiter (New Line, Form Feed, or Carriage Return).
Comment	Begins with semicolon and ends with line delimiter.

Table 3.2 Statement field terminals

You may include extra spaces and tabs between the statement fields in your source line without affecting the assembler's interpretation of that line. Thus, the following lines are equivalent:

label:statement body;comment ↲

label: statement body   ;comment ↲

The maximum length for a source statement is 136 characters. The macroassembler truncates all lines that are too long.

The naming of labels is explained under "Symbols," later in this chapter. The formats for each type of statement body are explained under "Statement Types."

The comments in your source program facilitate program development and maintenance. The assembler does not interpret comments, and they do not affect the generation of the object module.

Comments may appear alone on a line, or following a label or statement body. When the assembler encounters a semicolon, it ignores all subsequent characters up to the line delimiter. Comments appear on the *assembly listing* output.

Statement Types

This section describes the syntax associated with the five types of source statements, namely assembly language instructions, macros, pseudo-ops, assignments, and data.

Assembly Language Instructions

Each assembly language instruction in the source module conforms to the syntax

inst {arg}. . .

where:

inst is an assembly language instruction mnemonic.

arg is an argument to the assembly language instruction. Not all instructions require arguments; some require more than one.

The following are examples of assembly language instructions:

```
LDA     0,1,1
JMP     LOOP
POPB
```

The *Principles of Operation* manual for your ECLIPSE computer describes each assembly language instruction.

Literals

A literal is a data value implicitly defined in an argument to a memory reference instruction. The format for a literal is

MRI {ac,}={exp | ext | inst}

where:

MRI is the mnemonic for a memory reference instruction.

{exp | ext | inst} is an expression, an external reference symbol (.EXTD or .EXTN), or an assembly language instruction. (The brackets and bars (|) simply indicate that you have a choice. They are not included in the command line.)

When the assembler encounters an equal sign in the displacement field of a memory reference instruction, it calculates the data value following the sign and stores the value in another location. Literals are grouped together in a *literal pool* whose location you may explicitly control.

You may use any absolute or relocatable expression, external reference symbol, or one-word instruction as a literal.

You may use the .LPOOL (literal pool) assembler directive to explicitly set aside a storage area for the literals generated by your program. If you use the .LPOOL directives in the .ZREL area of your program (page zero, relocatable), all parts of the program will be able to directly reference any literal in the "pool."

If you do not include the .LPOOL directive, the .END pseudo-op contains an implicit .LPOOL, so that every literal generated by the program is assured a location. See Chapter 7 for a complete description of both the .LPOOL and the .END pseudo-ops and examples of their use.

Literals defined at the time they are referenced in the MRI instruction occur only once within a literal pool, even if referenced by more than one instruction. Literals that are undefined when used in the MRI are given unique locations in the literal pool.

The following examples show uses of literals.

```
LDA     1,=3
JSR     @ = SUBR
LDA     2,=ADD 1,2
```

Macros

A macro is a series of assembly language source statements that you assign a name.

The syntax of a macro call is

macro-name arg . . .

where:

macro-name is the name you assign to the series of statements.

arg is an argument to the macro.

Chapter 5 explains how to create and use macros.

Pseudo-ops

Pseudo-ops are permanent symbols. They are used either as assembler directives, which instruct the assembler to do something, or value symbols, which represent internal assembler variables or values.

The syntax for an assembler directive is

.pseudo-op {arg}. . .

where:

.pseudo-op is the name of a pseudo-op.

arg is an argument to the pseudo-op.

By convention, pseudo-op names begin with a period, and every pseudo-op mnemonic is a permanent symbol. Assembler directives are the first character on a source line.

When a pseudo-op is used as a value symbol, at least one atom (other than a label) must precede it. For example:

```
X=.RDX
```

This statement assigns the value of the current input radix to the variable X.

Depending on how you use them in your source program, certain pseudo-ops can be used either as assembler directives or value symbols. The macroassembler uses the symbol's context in the source line to determine its intended use.

If the source line begins with a pseudo-op, the macroassembler interprets it as an assembler directive. If you use the pseudo-op as an argument or as an operand in an expression, the macroassembler interprets it as a value symbol. To clarify these rules, consider the following examples.

You may use the pseudo-op .RDX either to specify a new radix for numeric input (as an assembler directive) or to represent the value of the current input radix (as a value symbol). In the line

```
.RDX 10
```

the symbol .RDX begins the source line. Therefore, the macroassembler interprets it as an assembler directive and sets the current input radix to 10 (decimal). However, in the source line

```
(.RDX)
```

the character "(" signifies the beginning of an expression. Thus, the macroassembler interprets .RDX as a value symbol and generates a storage word with the numeric value of the current input radix (in this case, 10).

Other Assembler Directives

Three assembler directives do not conform to the format shown above:

- Double asterisks (**) in a source line (usually, at the beginning of the line) instruct the assembler to suppress the listing of that line. This directive is listed along with pseudo-ops that control listings in Chapter 6 (Table 6.7).
- A backslash (\) followed by a symbol name instructs the assembler to generate an integer or symbol name. This directive is explained in Chapter 5.
- A dollar sign ($) appended to a label generates unique labels within a macro. This symbol is explained in Chapter 5.

Assignments

An assignment statement associates a 16-bit value with a symbolic name.

The syntax of an assignment statement is

user-symbol = {integer | symbol | expression | instruction}

where:

user-symbol is a user-symbol conforming to the rules for symbol names;

integer is any 16-bit constant (you may not place a double-precision or floating point number here);

symbol is any local user symbol or value symbol;

expression is any valid expression;

instruction is any valid instruction.

Do not include the brackets or bars in your command line. They merely set off your choice.

Examples of assignment statements are

```
A=322
E=.RDX
G=ADD   0,1
```

If you place an instruction on the right hand of an assignment statement, the assembler assigns the assembled value of the instruction to the user symbol. The assembled value must not be longer than 16 bits.

Data

The syntax for the data statement is

{@}{number | symbol | expr}

where:

@ is the indirect bit symbol.

{number | symbol | expr} is any 16-bit number, valid local user symbol, or expression. (The brackets and bars indicate that you have a choice. They are not inserted in the command line.)

The @ symbol causes a 1 to be placed in bit 0, the indirect addressing bit.

The assembler generates a 16-bit storage word for each data statement.

Indirect addressing is explained in the *Principles of Operation* manual for your ECLIPSE computer.

Statement Components

This section explains the syntax associated with each of the components of source statements — delimiters and terminals, numbers, symbols, and expressions.

Delimiters and Terminals

Delimiters are characters that separate the source statements in your module. The delimiters, also called end-of-line characters, are New-Line, Form Feed, and Carriage Return.

In this manual all delimiters are represented by a curved down arrow (↲).

Terminals are characters that separate numbers, symbols, and expressions within a single source statement. Operators form a subset of terminals and are used to construct expressions (see "Expressions," below). Table 3.3 lists all terminals, including operators.

Symbol	Description
=	Assigns a value to the symbol preceding this sign Indicates a literal reference
:	Indicates the preceding symbol is a label
;	Indicates the beginning of a comment string
+ − * /	Arithmetic operators
== >< <= >= <>	Relational operators
& !	Logical operators
B	Bit alignment operator
()	May enclose a number, symbol, or expression or an assembler symbol
[]	May enclose the arguments in a macro call or an assembler symbol
%	Terminates a macro definition string
_ (underscore)	Directs the macroassembler to ignore the special meaning of a character that it precedes in a macro definition string Break character in a number string.

Table 3.3 Terminals

Numbers

The macroassembler allows you to use three types of number representation:

- single-precision integer, stored in one word (16 bit);
- double-precision integer, stored in two words (32 bits);
- single-precision floating-point constant, stored in two words (32 bits).

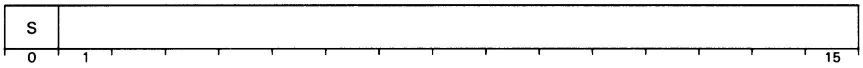
Single-precision integers may appear in expressions and data statements. Double-precision integers and floating-point constants appear only in data statements.

Single-Precision Integers

The macroassembler represents single-precision integers as single 16-bit words in the range 0 to $65{,}535_{10}$ (0 to 177777_8).

You can use two's complement notation to represent any signed integer in the range $-32{,}768_{10}$ to $+32{,}767_{10}$.

The first bit (bit 0) is the sign bit. If that bit is 0, the integer is positive; if it is 1, the integer is negative. Single-precision integers are represented as follows:



The format of a single-precision integer in your source module is

{*sign*} *d* {*d...*} {.} *break*

where:

sign is the integer's sign; use minus for negative numbers and plus for positive numbers. If you do not supply a sign, the macroassembler assumes that the integer is positive.

d is a digit in the range of the current input radix; the first digit must be in the range 0 through 9.

. is an optional decimal point. The macroassembler interprets the integer as decimal (base 10) if you supply the decimal point.

break terminates the integer. The break character may be any delimiter or terminal except :, [,], and (.

If a decimal point precedes the break character, the macroassembler evaluates the integer as decimal. If you omit the decimal point, the macroassembler evaluates the integer in the current input radix. You may set the input radix to any base from 2 to 20 (see the .RDX pseudo-op in Chapter 7).

When you select a radix of 11 or greater, your integers may contain letters that represent digits. For example, in base 16, the number 2F represents the value 47_{10}.

If the first digit of an integer starts with a letter, you must precede that integer with the digit 0. Otherwise, the macroassembler cannot distinguish the integer from a symbol. The following examples of legal hexadecimal (base 16) integers help clarify this rule.

```
0F
0A45
6A9
```

Take special care when using the bit alignment operator B with the integer. If you are using an input radix of 12 or greater, the macroassembler interprets B as a digit. If you want the macroassembler to interpret B as the bit alignment operator, place an underscore immediately before the B, or use parentheses. For example,

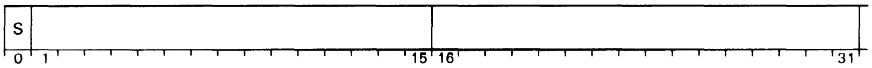
```
.RDX     16      ;Input radix equals 16.
49B7             ;Macroassembler interprets B as hexadecimal digit.
49_B7            ;Macroassembler interprets B as bit alignment operator.
(49)B7           ;Macroassembler interprets B as bit alignment operator.
```

Refer to "Operators in Expressions" in this chapter for a description of the bit alignment operator B.

Double-Precision Integers

The macroassembler represents double-precision integers in two consecutive words of memory (32 bits). Using two's-complement notation, you can represent any signed integer from $-2,147,483,648_{10}$ to $+2,147,483,647_{10}$. Unsigned double-precision integers may range from 0 to $4,294,967,295_{10}$.

The first bit of the first word (bit 0) is the sign bit. If that bit is 0, the integer is positive; if it is 1, then the integer is negative. Double-precision integers are represented as follows:



The general format for a double-precision integer in a source module is

{ sign }d{ d... }{ . }D break

where:

sign is the integer's sign; use minus for negative numbers and plus for positive numbers. If you do not supply a sign, the macroassembler assumes that the integer is positive.

d is a digit in the range of the current input radix; the first digit must be in the range 0 through 9.

. is an optional decimal point. The macroassembler interprets the integer in base 10 if you supply the decimal point.

D indicates a double-precision integer.

break terminates the integer. The break character may be any delimiter or terminator.

According to this definition, all of the following are legal integers:

```
25D     25D     1320D     -1D     +241D     -177777D
```

If a decimal point precedes the break character, the macroassembler interprets that integer as decimal. If you omit the decimal point, the macroassembler evaluates the integer in the current input radix.

The first digit in each integer must be in the range 0 through 9. If the input radix is 11 or greater, your integer may contain letters (e.g., 3F16). If the first digit of a number is a letter, precede that letter with a zero (i.e., use 0F5 instead of F5).

If the input radix is greater than or equal to 14, the letter D is interpreted as a digit. To force the assembler to interpret D as the double-precision indicator, precede the D with an underscore. For example, assuming radix 16:

```
12D    ;D represents digit.
12_D   ;D signals that 12 is a double-precision integer.
```

Special Integer-Generating Formats

Two special input formats convert ASCII characters to integers.

The first format converts a single ASCII character to its 8-bit binary value. The input format is

```
"a
```

where:

" is a quotation mark that directs the macroassembler to use the binary code for the following ASCII character.

a represents any legal ASCII character except New Line, Form Feed or Null (see "Character Set" at the beginning of this chapter for a list of the legal characters).

Input	Octal Value
"5	65
"A	101
"%	45

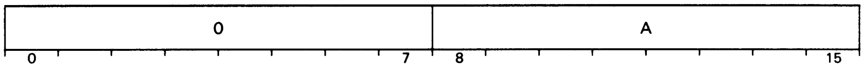
The accompanying examples illustrate the use of the quotation mark.

You may also use quoted characters as part of an expression. The accompanying examples illustrate this usage.

Input	Octal Value
"A+4	101+4
"C*5	103*5
"#−(%	43-45

When you use quotation marks with a delimiter ("↲), the macroassembler assembles the octal value for the delimiting character and also terminates that source line.

The macroassembler packs the value generated by this format in the rightmost byte of the word (i.e., in the least significant 8 bits). For example, the macroassembler stores "A as follows:



The second special integer-generating format converts up to two ASCII characters into an integer. The format is

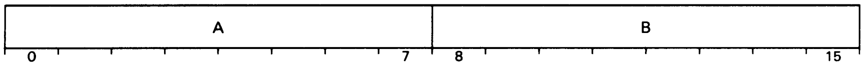
'string'

where:

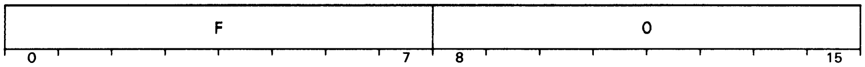
' is an apostrophe; the macroassembler requires you to enclose the ASCII characters in apostrophies;

string consists of any number of ASCII characters; the macroassembler uses only the first two characters in this string.

The macroassembler packs the octal values of *string's* first two characters from left to right in bits 0-15 of the integer. For example, the macroassembler stores both AB and ABCD as



If you supply only one character, the macroassembler places the corresponding octal value in the left byte of the word. Thus, the macroassembler stores F as



Two apostrophes without an intervening character string generate an integer containing all zeros (i.e., absolute zero).

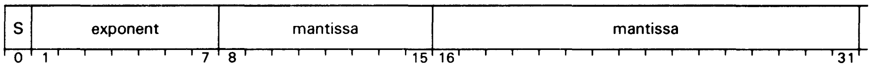
Source	Octal Value
"A	101
'AB'	40502
'BA'	41101
''	0
''+5-2	3

You may use the two special integer-generating formats wherever the macroassembler allows you to use integers. The accompanying simple expressions use the special formats. See "Generated Numbers and Symbols" in Chapter 5 for another related integer-generating format.

Single-Precision Floating-Point Constants

Floating-point constants represent fractional and exponential values. These numbers cannot appear in expressions or assignments. They may appear only in data statements.

The macroassembler uses two contiguous words of memory (32 bits) to represent a single-precision floating-point number. The number is represented as follows:



Bit 0 is the sign bit. If that bit is 0, the number is positive; if it is 1, the number is negative.

Exponent is the integer exponent of 16, expressed in excess−64 notation. The macroassembler represents exponents from -64_{10} to $+63_{10}$ with their binary equivalents from 0 to 127_{10} (177_8). The macroassembler represents a zero exponent as 64_{10} (100_8).

The macroassembler represents the mantissa as a 24-bit binary fraction. You may view the mantissa as six 4-bit hexadecimal digits. The range of the mantissa's magnitude is

$$16^{-1} < = \text{mantissa} < = (1 - 16^{6})$$

You may obtain the negative form of a floating-point number by complementing bit 0 (i.e., from 0 to 1, or from 1 to 0). The exponent and mantissa remain the same.

The range of a floating-point constant is

$$16^{-1} * 16^{64} < = \text{floating-point constant} < = (1 - 16^{6}) * 16^{63}$$

which is approximately

$$5.4 * 10^{-79} < = \text{floating-point constant} < = 7.2 * 10^{75}$$

The macroassembler normalizes all nonzero floating-point numbers. A floating-point number is normalized if the fraction (mantissa) is greater than or equal to 1/16 and less than 1. In other words, the binary representation of a normalized number has a 1 in one of the first four bits (8-11) of the mantissa. For example, if you specify the number 65.32, the macroassembler converts it to the base 16 equivalent of $.6532 * 10^{2}$.

Much of the floating-point number source format is optional. The minimum format is one digit, followed by either a decimal point or the letter E, followed by another digit. For example, 3.5 and 6E2 are both floating-point constants.

The complete source format for a single-precision floating-point number is one of the following:

{*sign*}d{*d..*}.*d* {*d...*}{E{*sign*}*d*{*d*}} *break*

{*sign*}*d*{*d...*}E{*sign*}*d* {*d*} *break*

where:

sign indicates the sign of a value (positive or negative) and is one of the following characters: + or −. If the sign appears before the number, then it defines the sign of that number. If a sign character appears after the letter E, then it defines the exponent's sign. If you do not supply a sign, the macroassembler assumes that the value is positive.

d is a digit in the range 0 through 9. The macroassembler always inteprets the mantissa and exponent as decimal (e.g., 26.5 equals .265*10^2 regardless of the current input radix).

. is an optional decimal point. If you include a decimal point but do not follow that point with either a digit or the letter E, the macroassembler treats the value as an integer, not a floating-point number.

E indicates floating-point number representation. You must follow the E with one or two digits representing the value of the exponent.

break terminates the floating-point number. The break character may be a semicolon, end-of-line character, tab, or space.

You may format the same floating-point number with the letter E, a decimal point, or both. For example:

Floating-Point Constant	**Assembled Value**
254.33	041376 052172
256.33E0	041376 052172
25433E-02	041376 052172
25433E-2	041376 052172
2543.3E-1	041376 052172
0.25433E03	041376 052172

The two octal numbers under the heading "Assembled Value" depict the two 16-bit words that represent the floating-point constant's value.

If the current input radix is 15 or greater, the macroassembler may interpret the letter E as a digit rather than the floating-point number indicator. If the decimal point is omitted, to avoid ambiguity, precede the exponential E with an underscore when representing a floating-point constant. For example,

```
.RDX        16  ;Input radix is 16.
-5E3            ;E is a hexadecimal digit and -5E3 represents
                ;an integer.
-5.E3           ;E indicates floating-point number
-5_E3           ;representation (i.e., -5*10^3).
```

The following examples show floating-point numbers and the corresponding values that the macroassembler stores.

Floating-Point Constant	Assembled Value
1.0	040420 000000
3.1415926	040462 041766
-1E0	140420 000000
+5.0E-1	040200 000000
+273.0E0	041421 010000
0.33E2	041041 000000

Symbols

The following section explains the syntax for naming symbols. Additional rules for defining and redefining symbols are explained in Chapter 5, "Defining Symbols."

Symbol Names

Every symbol must conform to the following syntax:

a{ *b...* } *break*

where

a is the first character of the symbol name and may be any upper- or lower-case letter, period, question mark, or underscore.

b represents the succeeding characters in the symbol name and can include upper- and lower-case letters, numerals (0 - 9), period, question mark, and underscore.

break terminates the symbol name; a break character may be any terminal or delimiter except left parenthesis or underscore.

The assembler does not distinguish between upper- and lower-case letters. For example, "START" is interpreted as identical to "start". Symbol names must be unique to eight characters if the /8 switch is used, and five characters if the switch is not used. Additional characters are ignored.

If you include the underscore character in a symbol that appears in a macro definition, or as an argument to a macro, precede that underscore character with another underscore. That is, inside the macro, use A__B to represent the symbol A_B (see "Macro Definition" in Chapter 5).

Expressions

This section explains expression syntax. Then it lists and explains the operators that may appear within expressions and the order of their operation.

Expressions conform to the format:

{*sign*}*operand operator operand . . .break*

where

sign is one of the unary operators, + or −.

operand may be a user symbol, value symbol, integer constant, or another (parenthesized) expression.

operator is a macroassembler operator; operands must precede and follow every operator except the unary operators.

break terminates the expression; the break character may be any terminal or delimiter.

For example, the following are legal expressions:

```
START-1        6*3-5         A+3*B/C
```

Do not include spaces or tabs within expressions, since these are terminals that indicate the end of an expression.

Operators in Expressions

Operators are terminals made of one or two characters that tell the assembler how to interpret the information in an expression.

Binary operators require two operands, one before and one after the operator.

Unary operators require a single operand, which follows the operator. Two unary operators may, however, be separated by parentheses.

Operators fall into three groups: arithmetic, logical, and relational. There is also one bit alignment operator. Table 3.4 lists the operators. The plus (+) and minus (−) may be unary or binary. All other operators are binary.

Type of Operator	Operator	Meaning
Arithmetic	+	Addition (2 + 3) or unary plus (+3)
	−	Subtraction (5 − 4) or unary minus (−4)
	*	Multiplication
	/	Division
Logical	&	Logical AND
	!	Logical OR
Relational	==	Equal to
	<>	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
Bit alignment	B	Single precision bit alignment (16 bits)

Table 3.4 Operators

Arithmetic Operators

The binary operators +,−,*, and / perform the common arithmetic operations of addition, subtraction, multiplication and division, respectively.

The unary plus and minus operators indicate whether the following expression is positive or negative (greater than or less than zero).

Logical Operators

The macroassembler provides two logical binary operators: & and !. The operator & directs the macroassembler to perform a logical AND operation; the operator ! represents the logical OR (inclusive) operation.

For a logical AND (&), the result in a given bit position is 1 only if both operands contain a 1 in that bit position. The following example shows how the macroassembler evaluates the logical expression 6&4:

Bit representation of 6:	000...110
Bit representation of 4:	000...100
Result of logical AND (&):	000...100

Thus, the resulting value of the expression 6&4 is 4 (100_2).

For a logical OR operation (!), the result in a given bit is 1 if either or both operands contain a 1 in that bit position. The following example shows the logical OR operation for the expression 6!4:

Bit representation of 6:	000...110
Bit representation of 4:	000...100
Result of logical OR (!):	000...110

The value of the expression 6!4 is 6 (110_2).

Both operands in a logical expression must be absolute expressions; using relocatable expressions as operands will cause an error.

Relational Operators

An expression containing a relational operator is a relational expression.

A relational expression evaluates to either absolute zero (false) or absolute one (true). Absolute zero has a zero in every bit; absolute one has zeros in all bits except the least significant bit (bit 15), which contains a one. These values are not relocatable.

The following examples show how the macroassembler evaluates relational expressions.

Expression	Assembled Value	Comment
5==6	000000	False
3==3	000001	True
7>1	000001	True
55<=41	000000	False
7<>6	000001	True

The octal number under the heading "Assembled Value" depicts the 16-bit word that represents the expression's value. Each expression yields a single-precision result.

Bit Alignment Operator

The macroassembler recognizes a bit alignment operator, B. This operator allows you to right justify an integer on a bit boundary.

The format for using the bit alignment operator is

operand B position

where:

operand is an integer, local symbol, or expression whose value you wish to align.

B is the bit alignment operator.

position is an integer, symbol, or expression whose value indicates the bit position for aligning operand. If it is an integer, it will be interpreted in decimal (rather than in the current input radix).

When you use a bit alignment expression, the macroassembler aligns the rightmost bit of operand at the bit position specified in *position*.

The bit alignment expression must not contain any spaces, etc.

The value of *position* must be in the range

$$0 < = \text{position} < = 15_{10}$$

The result of a B bit alignment expression equals

$$\text{operand} * 2^{(15. \text{ - position})}$$

When the operand is an integer and you use an input radix of 12 or greater, the macroassembler interprets the character B as a digit instead of an operator. To avoid ambiguity, place the operand value inside parentheses or precede it with an underscore. For example:

```
.RDX      16      ;Input radix equals 16
31B4              ;B is interpreted as a hexadecimal digit
(31)B4            ;B is recognized as the bit alignment operator
31_B4             ;B is recognized as the bit alignment operator
```

When the operand is a symbol, the assembler can misread B as part of the symbol; that is, the assembler may not recognize B as an operator. Therefore, if the operand preceding the bit alignment operator is a symbol, enclose that argument in parentheses. For example:

Assembled Value	Expression	Comment
000025	A=25	User symbol A has the value 25.
000006	AB9=6	User symbol AB9 has the value 6.
002500	(A)B9	Aligns the rightmost bit of 25_8 in bit 9.

When using the bit alignment operator in a lengthy expression, enclose both operands in parentheses to ensure that the assembler aligns the value correctly. For example:

Assembled Value	Expression	Comment
014000	(3B12)B(3+4)	(3B12) equals 30_8. Thus, the expression equals (30)B(7).

Priority of Operators

You may use more than one operator in an expression. The macroassembler evaluates operators according to their priority levels. It resolves high priority operators first and low priority operators last. Table 3.5 lists the priority levels of all operators.

Operators	Priority Level
B	3 (highest priority)
+ − * / & !	2
< <= > >= == <>	1 (lowest priority)

Table 3.5 Operator priority levels

If an expression contains operators of equal priority, the assembler evaluates them from left to right.

The following examples show how the macroassembler uses operator priority to evaluate expressions. The radix equals 8.

Assembled Value	Expression	Comment
000001	3*2==6	The macroassembler evaluates * first, then ==. The relationship is true.
000000	4<6−3	The macroassembler evaluates − first, then <. The relationship is false.
000005	4/2+3	/ and + are equal in priority so the assembler evaluates them from left to right (first /, then +).
000011	3*2−1+4	All operators are of equal priority so the macroassembler evaluates them from left to right.
000006	3!1*2	Both operators are of equal priority so the macroassembler evaluates them from left to right (3!1=3;(3)*2=6).
000001	4&2< >3!1	The macroassembler evaluates & first, followed by !, and, lastly, <>. The resulting relationship is true.
030000	2*3B4	B has higher priority than *, so the assembler evaluates 3B4 first. It then multiplies the result by 2.

You may change the order in which the macroassembler evaluates operators by including parentheses in your expression. The macroassembler always evaluates an expression in parentheses first. Within a set of parentheses, the macroassembler evaluates operators according to the priority sequence presented above. If you nest one set of parentheses inside another set, the macroassembler evaluates the innermost expression first.

The following examples show the use of parentheses in expressions. The radix equals 8.

Assembled Value	Expression	Comment
000006	2*(4−1)	MASM performs operations in the following order: (4−1)=3; 2*(3)=6.
000002	1+(6/2)/2	Order of operations: (6/2)=3. 1+(3)=4; (4)/2=2.
000004	(3*2)−(4/2)	Order of operations: (3*2)=6. (4/2)=2; (6)−(2)=4.
000001	(5<=5)+ (6==2)	Order of operations: (5<=5)=1. (6==2)=0; (1)+(0)=1.
000006	3*((3+1)/2)	Order of operations:(3+1)=4. ((4)/2)=2; 3*((2))=6.
000025	A=25	Radix equals 8.
000010	C=10	

Relocating Symbols

Each symbol is associated with a relocation base either explicitly or implicitly, according to the rules enumerated below.

1. Each externally defined symbol (defined using the .EXTD or .EXTN pseudo-ops) is assigned a unique, explicitly defined relocation base (see example at rule 3).
2. Labels are assigned the relocation base of the partition in which they appear. For example,

```
        .ZREL         ;the following words are in the
                      ;ZREL partition
A:      10            ;Each entry requires one word
                      ;of storage.  Thus the value 10 resides
                      ;at relative location 0, and the value
B:      20            ;20 resides at relative location 1
```

3. If you define one symbol with respect to another symbol that has a relocation base, then the second symbol is assigned the first one's relocation base. For example,

```
        .ZREL
M:      .10           ; M has the .ZREL base
        .NREL 0       ; Impure area

X:       5            ; X has impure NREL base
       A = X +3       ; A has the same base as X
       N = M + 1      ; N has the same base as M
```

4. If you define a symbol in terms of integers alone, the symbol is assigned the absolute relocation base. For example,

```
        B=3     ;B has the absolute relocation base.
```

5. All symbols defined by instructions are assigned the absolute relocation base. For example,

```
        F= LDA 0,1      ; F has absolute relocation base
```

6. All value symbols except .LOC, .POP, .TOP and the period(.) have absolute relocation base.

Addresses and Contents, Relocation Comparison

Remember that each address has relocation properties, as do the contents of each address. For example, consider the following source lines:

```
        A=5
FIVE:   A
```

When constructing the object module, the assembler assigns a relocatable address to the label FIVE. Thus, FIVE has a relocation base and offset associated with it. The value at FIVE is A, which is itself a symbol with relocation properties.

The contents of any address may be a symbol or expression; each symbol and expression has a relocation property. The example below develops this idea.

```
        .ZREL
Z:      A               ; Z has the .ZREL base, as does A
A:      10              ; Value at A, 10, has absolute base
                        ;
        .NREL   0       ;
NO:     B               ; NO has NREL 0 relocation base
        B=A+1           ; Value at NO has the same base as A
                        ; according to rule 3.
```

Labels are named addresses and are assigned the relocation base of the preceding .LOC, .NREL, or .ZREL. In the example above, the address A has the ZREL relocation base, and the value at A has absolute relocation, since it is a constant. Similarly, the symbol NO has a different relocation base than its contents, the symbol B.

Relocating Expressions

Expressions can be divided into two classes:

- absolute expressions;
- relocatable expressions.

Absolute expressions resolve to integer values, and are assigned the absolute relocation base.

Relocatable expressions resolve to relocatable values. That is, the result of a relocatable expression is not simply an integer; it contains a relocatable component that cannot be resolved until the program is bound.

Absolute Expressions

The simplest absolute expressions contain operands that have integer values. For example

```
6*6
(.PASS)
(.RDX)<=(6/2)
```

Absolute expressions can also contain relocatable operands if the resulting value has no relocatable components. That is, if all relocatable components cancel each other out, the expression is absolute. Consider the following example.

```
      .ZREL
A:    10
B:    20
      (B-A)+40
```

The macroassembler computes the values for A and B as follows:

$$A = RB_Z + 0$$
$$B = RB_Z + 1$$

where:

RB_z is the ZREL relocation base.

0 is the offset from RB_z that the macroassembler assigns to A.

1 is the offset from RB_z that the macroassembler assigns to B.

The values for A and B are relative to the ZREL relocation base. Thus, A and B have relocatable values.

The macroassembler evaluates the expression $(B-A)+40$ as follows:

$$\begin{aligned}(B-A)+40 &= ((RB_z + 1)-(RB_z + 0)) + 40\\ &= (RB_z - RB_z) + (1 - 0) + 40\\ &= (0) + 1 + 40\\ &= 41\end{aligned}$$

In this expression, the two relocatable components (RB_z) cancel each other out, leaving an absolute value (i.e., 41). Thus, the expression $(B-A)+40$ is an absolute expression, even though it contains relocatable operands.

Since all labels have relocatable values, you may never use labels in absolute expressions unless their relocatable components cancel out.

Since the assembler can completely resolve an absolute expression, it verifies that the expression's value is legal for the field it appears in. For example:

```
        LDA     0,23571
```

The absolute expression 23571 is too large to fit in the 8-bit LDA displacement field. Thus, the macroassembler returns an error for this instruction.

Relocatable Expressions

Relocatable expressions resolve to relocatable values. That is, the result of a relocatable expression is not simply an integer; it contains a relocatable component that cannot be resolved until bind time.

The assembler can reduce all valid relocatable expressions to one of the following syntaxes:

{ 2* }*rel-symbol* ± *abs-expr*

abs-expr ± *rel-symbol* { *2 }

where:

rel-symbol is a symbol whose value is relocatable.

abs-expr is an absolute expression.

The following example contains several relocatable expressions.

```
    .ZREL
A:  10                  ;A has the ZREL relocation base
    .NREL 0
B:  20                  ;B and C have the unshared (impure) NREL code
C:  30                  ;relocation base
    A + 20              ;Relocation symbol A plus absolute expression 20.
    B-5                 ;Rel-symbol B minus abs-expr 5.
    A + (B - C)         ;Rel-symbol A plus abs-expr (B - C).
    (10 + (B - C))-A    ;Abs-expr (10 + (B - C)) minus rel-symbol A.
```

Note that you may include more than one relocatable symbol in an expression as long as all but one of their relocation bases cancel out. In the example module, A+(B−C) contains three relocatable values. However B and C have the same relocation base (impure NREL code); thus, (B−C) has an absolute value.

You may multiply the relocatable symbol by two. For example:

```
    .ZREL
X:  10
Y:  20
    2*X
    (10) + (2*Y)
```

Both 2*X and (10) + (2*Y) are legal expressions.

An expression whose relocatable symbol value is multiplied by two is called byte-relocatable. In most cases, you use byte-relocatable expressions as byte-pointers (values that specify a byte's address). In the example above, the expression 2*X is a byte-pointer to the left byte at address X. For more information on byte-pointers, refer to the *Principles of Operation* manual for your ECLIPSE computer.

Resolving Relocatable Expressions

At assembly time, all relocatable expressions must resolve to a relocation base, a relocation operation, and an integer component (i.e., an absolute value).

An example will help clarify these rules:

```
    .ZREL   ;ZREL memory partition.
    5       ;The value 5 resides at location 0
X:  15      ;in this module; 15 resides at
    X + 4   ;location 1.
```

In this example, the value of label X equals the ZREL relocation base plus 1 word; i.e., X's value equals the third address in the ZREL partition. Thus, you can think of X's value as

$$X = \mathrm{RB}_z + 1$$

where:

RB_z is the ZREL relocation base.

1 is the offset from RB_z that the macroassembler assigns to X.

The macroassembler cannot completely resolve the expression X + 4 because the ZREL relocation base does not have a value. However, the macroassembler can partially evaluate the expression as follows:

$$\begin{aligned}(X - 4) &= ((\mathrm{RB}_z + 1) - 4)\\ &= (\mathrm{RB}_z - 3)\end{aligned}$$

At this point, the macroassembler cannot process the expression any further. Thus, it passes to the binder the absolute value 5 (integer component), the ZREL relocation base RB_z (relocatable component), and the unresolved operator +.

During the binding process, the ZREL relocation base for this expression receives a value. Then, the binder can fully resolve the values for the symbol X and the expression X − 4.

In most cases, you include only one relocatable operand in each expression (as in the example). The macroassembler does, however, allow you to include more than one relocatable value in a single expression. Again, the expression must resolve to a single relocation base, a + or − operator, and an integer component or you will receive an error. For example:

```
     .ZREL
Y:   10
X:   20
     .NREL 0
     30
Z:   40
     (X - Y) + Z
```

The expression (X − Y) + Z includes three relocatable operands. X and Y have the ZREL relocation base; Z has the unshared (impure) NREL base. The macroassembler evaluates this expression as follows:

$$\begin{aligned}(X - Y) + Z &= ((\mathrm{RB}_z + 1) - (\mathrm{RB}_z + 0)) + (\mathrm{RB}_n + 1)\\ &= ((\mathrm{RB}_z - \mathrm{RB}_z) + (1 - 0)) + (\mathrm{RB}_n + 1)\\ &= ((0) + (1)) + (\mathrm{RB}_n + 1)\\ &= \mathrm{RB}_n + 2\end{aligned}$$

RB_z is the ZREL relocation base, and RB_n is the unshared NREL code base. The values 1, 0, and 1 are the offsets for X, Y, and Z from their respective relocation bases.

Since both ZREL relocation bases cancel out, the expression is legal. After processing the expression, the macroassembler passes to the binder the absolute value 2 (integer component), the relocatable value RB_z (relocatable component), and the relocation operation.

The previous section explained that you may mutliply a relocatable symbol value by 2 to create a byte-relocatable expression; for example:

```
        .ZREL
        10
X:      10
        2*X
```

The expression 2*X serves as a byte pointer to the first byte at address X. The macroassembler evaluates this expression as follows:

$$\begin{aligned} 2*X &= 2*(RB_z + 1) \\ &= (2*RB_z) + (2*1) \\ &= (2*RB_z) + 2 \end{aligned}$$

The macroassembler cannot process this expression any further. Thus, it passes to the binder the relocation base ($2*RB_z$), the relocation operation (+), and the integer value 2. Any expression whose relocatable component equals two times a relocation base is byte-relocatable.

Table 3.6 displays different forms of expressions and shows how the assembler resolves each. The following notation is used in Table 3.6:

n and m — represent two different absolute values; they may be integers, symbols, or absolute expressions.

r and p — represent two relocatable values with the same relocation base; they may be symbols or expressions.

s — represents a relocatable value whose relocation base is different from that of r and p.

RB_r — is the relocation base associated with r's values.

r_{off} — is the offset of r's value from relocation base RB_r; i.e., $r = RB_r + r_{off}$.

All expressions involving the operators <, <=, >, >=, ==, or <> result in an absolute value of either zero (false) or one (true).

When operands in these expressions have different relocation bases, all comparisons result in a value of zero (false), except when the operator is <> (not equal to).

The logical operators & and ! require absolute operands.

Expressions	**Relocatable Component**	**Integer Component**	**Relocation Operation**
n+m	absolute	n+m	none
n−m	absolute	n−m	none
n*m	absolute	n*m	none
n/m	absolute	n/m	none
n<=m	absolute	n<=m	none
n&m	absolute	n&m	none
n!m	absolute	n!m	none
n+r	RB_r	$n+r_{off}$	+
n−r			
r−n	RB_r	$r_{off}-n$	+
r+n	RB_r	$r_{off}+n$	+
2*r	$2*RB_r$	$2*r_{off}$	+
r−r	absolute	0	none
r−p	absolute	$r_{off}-p_{off}$	none
n/r			
n*r			
r+r	$2*RB_r$	$2*r_{off}$	+
r*r			
r&r			
n&r			
r!r			
n!r			
r+p	$2*RB_r$	$r_{off} + p_{off}$	+
r*p			
s+r			
s−r			
s*r			
r/p			

Table 3.6 Relocatable expressions

NOTES: *Blank columns indicate the expression is illegal.*
Any expression with a relocatable component equal to $2*RB_r$ is byte-relocatable.

Partitioning Programs and Controlling Assembly

This chapter contains an overview of the pseudo-ops you can use to partition your program and to control assembly and binding.

The first section expands the discussion of assembler partitions given in Chapter 1. It lists and describes pseudo-ops that relate to partitions and types of code (pure and impure; shared and unshared). It also lists the pseudo-ops you can use to begin and end a module.

The second section describes the pseudo-ops that control assembly and binding, explaining how portions of a source module are assembled, how modules communicate with each other, and how information is relayed to the binder.

Refer to Chapter 7 for examples and instructions.

Partitioning Programs

You control the partitioning of your program with the location counter and memory management pseudo-ops described in this chapter. Depending on the pseudo-ops used, the assembler determines the offset for each program component in your source module, using one of five locations counters:

Absolute
ZREL
NREL 0
NREL 1
GLOC

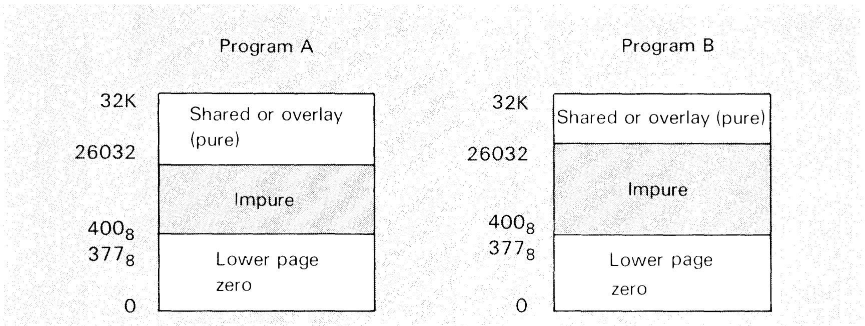
The assembler builds different "data blocks" in the object module for each location counter.

NOTE: *Each location counter can have several different and nonsequential or noncontiguous data blocks in the object module. The following discussion is general and does not strictly apply to .GLOC. (See the pseudo-op dictionary in Chapter 7 for further discussion.)*

The binder, in turn, places each relocatable unit of code in the pure or impure areas of the user process space. Pure code is placed in the shared or overlay segment of memory (depending on binder directives), and impure code is placed in the impure segment. These segments vary in size from program to program. As a consequence, an absolute location might turn out to be in the impure area of one program and the pure area of another program (although this is rare and not encouraged).

For example, as illustrated in Figure 4.1, absolute address 26037 may be in the impure area of one program and in the pure area of another program, depending on the amounts of code assigned to the NREL 0 (impure) and NREL 1 (pure) partitions in each program.

See the *MP/AOS System Programmer's Reference,* DGC No. 093-400051 and Part 2 of this manual for further information about process areas and program images.



DG-08633

Figure 4.1 Varying sizes of process segments

Assembler Location Counters

The Absolute Counter

The absolute address location counter is used for source lines to which you have assigned absolute addresses. Unlike relocatable addresses, these addresses are not modified by the binder. Absolute addresses may be anywhere in the logical address space — locations 0 through 32K-1 (077777_8).

Typically, you use absolute locations for words in lower page zero that have functions associated with them. These are the locations for auto-increment and auto-decrement (locations 20 to 27 and 30 to 37) and stack and frame values.

The ZREL Counter

The ZREL location counter is used for words that have relative addresses in lower page zero (locations 50_8 to 377_8). You may express any address in the ZREL partition in 8 bits. Thus, when referencing a location in the ZREL partition you may use any memory reference instruction (MRI) since all provide displacement fields of 8 or more bits. When the program is run, the ZREL portion of the program resides in an area of memory that is neither shared nor write-protected.

The NREL Counter

The NREL location counter is used for words that have relative addresses in the range 400_8 to 077777_8. Words in NREL 0 (impure) are located in an area of memory that is not shared or write-protected. Code in NREL 1 (pure) is write-protected, and is placed in either the shared or the overlay segment of the process space, depending on how the program is bound.

The GLOC Counter

The GLOC counter is used to partition code relative to a symbol in another module. The attributes of the code (pure, impure, etc.) are determined at bind time by the attributes of the external symbol.

Assembler Location Counters	Type of Code	Write Protected?
Absolute	Shared (pure) Overlay (pure, not shared) Impure (not shared)	Yes/No
ZREL	Impure (not shared)	No
NREL 0	Impure (not shared)	No
NREL 1	Shared (pure) Overlay (pure, not shared)	Yes
GLOC	Any	

Table 4.1 Location counter/code relationships

Location Counter and Memory Management

Generally, you use location counter and memory management pseudo-ops to assign memory locations to the statements in your source module. Specifically, the pseudo-ops in this category allow you to

- reserve a block of storage locations;
- set the location counter to a specific value;
- place source code in predefined memory partitions.

Table 4.2 lists the lcoation counter and memory management pseudo-ops.

The period pseudo-op (.) is a value symbol that represents the current location counter. A location counter is an assembler variable that designates the address and relocation base of the next memory location the macroassembler will assign.

The .ZREL and .NREL pseudo-ops direct the assembler to place the source lines that follow them in predefined memory partitions. Use .ZREL for relocatable data that must reside in lower page zero (locations 0-377_8). Use .NREL for relocatable data that can be anywhere else in the user process space (locations 400_8 to 077777_8). The .NREL pseudo-op allows you to place source code in either shared or unshared partitions.

When used as value symbols, .NREL and .ZREL return next values and relocations for the NREL 0 and ZREL location counters.

Use the .LOC pseudo-op to set the location counter to a specific value within a memory partition. Use .BLK to reserve an initialized fixed-length block of addresses. Such blocks are commonly used for data storage (tables, for example).

Use the .LPOOL pseudo-op to mark locations where literal pools can be built, if necessary.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.(period)	No	Yes	Represent the value of the current location counter
.BLK	Yes	No	Reserve a block of memory locations
.GLOC	Yes	No	Set location counter relative to an external symbol.
.LOC	Yes	Yes	Set a location counter Represent current location counter
.LPOOL	Yes	No	Reserve a variable block of memory locations
.NREL	Yes	No	Specify a pure or impure normal relocatable memory partition (predefined) Represent value and relocation of next NREL 0 location
.ZREL	Yes	No	Specify the lower page zero relocatable memory partition (predefined) Represent value and relocation of next ZREL location

Table 4.2 Location counter and memory management pseudo-ops

In addition to the pseudo-ops listed in Table 4.2, the .COMM and CSIZ pseudo-ops are used for memory allocation. They reserve common areas for intermodule communication.

Beginning and Ending Modules

Five pseudo-ops mark the beginning and end of modules.

The .TITLE pseudo-op provides a name for the object module. This is an internal name that is not the same as the file name of the object module. The name specified by .TITLE need not be the same as the name specified by .OB.

The .OB pseudo-op allows you to name the object module with a name other than the *sourcefile*.OB automatically assigned by the assembler. Use of this pseudo-op is related to the filenaming rules explained in Chapter 6.

The .REV pseudo-op helps you number revision levels of a program — for example, MP/AOS Rev 2.3 and MP/AOS Rev 3.0.

The .END pseudo-op terminates the source code you pass to the assembler. Also, by supplying an address to .END, you indicate where you want the program to begin at execution time.

Use .EOF or .EOT when you include more than one source file on the macroassembler command line. When you place this pseudo-op at the end of a file, you inform the assembler that the end of the current input file has been reached but more files will follow. All files, except the last one, should end with the .EOF or .EOT pseudo-op; the last one should end with .END

Table 4.3 lists the pseudo-ops that begin and end modules.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.END	Yes	No	Indicate end of program module; declare start address
.EOF or .EOT	Yes	No	Indicate end of file (tape)
.OB	Yes	No	Specify name of object (OB) file.
.REV	Yes	No	Assign revision level
.TITLE	Yes	No	Name an object module (recognized by binder)

Table 4.3 Pseudo-ops that begin and end modules

Controlling Assembly

Pseudo-ops that control assembly are either intramodule or extramodule.

Intramodule pseudo-ops give instructions to the assembler that are used during the assembly of a single module. These include:

- Loop and conditional assembly pseudo-ops that allow you to include or not include sections of source code in assembly depending on the evaluation of an absolute expression.
- Value symbol pseudo-ops that monitor the assembler's own actions.
- Assembler stack control pseudo-ops that allow you to store and retrieve values and relocation properties during an assembly.

Extramodule pseudo-ops are of two types:

- Intermodule communication pseudo-ops that coordinate references among different modules bound into a single program.
- Binder directive pseudo-ops that enable you to pass to the binder information it will need to build the program file.

Intramodule Assembly Control

Loop and Conditional Assembly

The pseudo-ops in this category allow you to

- assemble a series of source lines a specified number of times;
- conditionally assemble or bypass source lines based on the evaluation of an expression.

Table 4.4 lists the pseudo-ops you use to perform these functions.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.DO	Yes	No	Assemble the following source lines a specified number of times
.ENDC	Yes	No	Define the end of loop or conditional assembly lines
.GOTO	Yes	No	Unconditionally skip source lines
.IFE	Yes	No	Assemble the following source lines only if the value of the supplied expression equals zero
.IFG	Yes	No	Assemble the following source lines only if the value of the supplied expression is greater than zero
.IFL	Yes	No	Assemble the following source lines only if the value of the supplied expression is less than zero
.IFN	Yes	No	Assemble the following source lines only if the value of the supplied expression does not equal zero

Table 4.4 Loop and conditional pseudo-ops

The .DO and .ENDC are used in conjunction to assemble a section of source code repeatedly, that is, to implement a loop at assembly time.

The .IF pseudo-ops are used with the .ENDC pseudo-op to indicate sections of the source module that are to be assembled if an expression satisfies a certain condition. For example, the .IFE pseudo-op directs the assembler to process a section of code only if the supplied expression (the argument to .IFE) equals zero.

The .GOTO pseudo-op allows you to unconditionally skip a section of the source module. The argument to the .GOTO pseudo-op is a conditional label. These labels, set off in brackets, indicate the end of a conditional assembly. (See .ENDC in Chapter 7 for an example). You can use this pseudo-op in conjunction with those described above to identify sections of the source program to be conditionally assembled.

Assembler Monitoring

Three value symbols, .PASS, .MCALL, and .ARGCT, may be used as expressions for evaluation in conditional assembly. Table 4.5 summarizes these pseudo-ops.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.ARGCT	No	Yes	Represent number of arguments passed to a macro
.MCALL	No	Yes	Determine whether a macro has been called before current assembler pass
.PASS	No	Yes	Indicate the current assembler pass (0 = pass 1; 1=pass 2).

Table 4.5 Value symbols that control assembly

Chapter 6 explains how to use .ARGCT and .MCALL within macros. Refer to Chapter 7 for examples and instructions for all three pseudo-ops.

Assembler Stack Control

The assembler maintains a push-down stack for use at assembly time that is analogous to but distinct from the hardware stack which is used during program execution (run time). The last item placed on the stack is the first item retrieved from it.

Table 4.6 lists the three pseudo-ops you use to manipulate the stack.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.PUSH	Yes	No	Place value and relocation property of expression onto stack
.POP	Yes	Yes	Pop stack and represent value and relocation property of expression from stack
.TOP	No	Yes	Return value and relocation property of expression of stack. Do not pop stack.

Table 4.6 Stack control pseudo-ops

The .PUSH and .POP pseudo-ops are used to place and retrieve values from the stack. The .POP pseudo-op is also a value symbol representing the popped value. The .TOP pseudo-op is a value symbol equal to the value of the expression most recently placed on the stack.

Chapter 7 contains complete descriptions of these pseudo-ops and examples of their use.

Extramodule Pseudo-Ops

Intermodule Communication

Intermodule communication pseudo-ops allow you to define symbols and data in one source module and to reference that information from a separately assembled module. (Recall that symbols referenced in more than one module are *global symbols;* when assembling one module, the assembler assigns an external relocaton base to symbols defined in other modules.)

Intermodule communication pseudo-ops declare entry points, external symbols, and both labeled and unlabeled common areas. Table 4.7 lists the intermodule communication pseudo-ops.

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.COMM	Yes	No	Reserve a labeled common area for intermodule communication
.CSIZ	Yes	No	Reserve an unlabeled common area for intermodule communication
.ENT	Yes	No	Declare one or more global symbol entries
.EXTD	Yes	No	Delcare one or more external displacement symbols (external symbol value is 8 bits or less)
.EXTN	Yes	No	Declare one or more external normal symbols (external symbol value is 16 bits or less)
.EXTU	Yes	No	Treat undefined symbols as external displacement (.EXTD) symbols
.GADD	Yes	No	Add a scalar value to an external symbol value
.GREF	Yes	No	Add a scalar value to an external symbol without changing bit zero
.GLOC	Yes	No	Start the location counter at the value of an external symbol

Table 4.7 Intermodule communication pseudo-ops

The .COMM and .CSIZ pseudo-ops reserve common areas for intermodule communication. Using these pseudo-ops, you may create data storage areas that are accessible to each module in your program. Use .COMM when you wish to assign a name to the common area (labeled common area); use .CSIZ to create an unlabeled common area.

An .ENT symbol may represent either an address or a data value that is available for use by separately assembled modules. The separately assembled modules that reference the .ENT symbol must each declare it with an .EXTD or .EXTN pseudo-op; these pseudo-ops tell the assembler that a symbol is defined externally (in a separately assembled module).

The symbols that the .EXTD and .EXTN pseudo-ops declare differ in the number of bits necessary to represent their values. You may use an .EXTD symbol (external displacement) in any field that is no more than 8 bits wide. Use an .EXTN (external normal) to declare a symbol that requires a field at least 16 bits wide.

The .EXTU pseudo-op instructs the assembler to interpret undefined symbols as external displacement (.EXTD) symbols. This pseudo-op may be used in programs where undefined symbols are defined at bind-time by the use of run-time libraries.

The .GADD and .GREF pseudo-ops enable you to calculate a value by adding a scalar to a value defined in another module.

Sizing External Symbol Values

If a symbol is declared .EXTN, the assembler assumes that it is a 16-bit value. If the symbol is declared .EXTD, the assembler assumes it is an 8-bit value. The assembler assures that a symbol declared .EXTN is not used in the displacement field of a memory reference instruction.

If you use an externally defined symbol in a field that is not wide enough, the assembler reports an error. For example, if your source program includes the statement

```
.EXTN   A
```

the assembler assumes that the value of A is too large to be represented in 8 bits. Each time that you use A in this source module, the assembler checks that the corresponding field is not the displacement field of a memory reference instruction. If you use A in a memory reference instruction, for example

```
JMP   A
```

the assembler returns an error.

Binder Directives

Three pseudo-ops allow you to pass to the binder information that it will need to build the program's interface to the operating system. To use these pseudo-ops, you need to understand the concepts explained in the *MP/AOS System Programmer's Reference,* DGC No. 093-400051, and in Part 2 of this manual. These pseudo-ops are listed in Table 4.8.

Although these pseudo-ops do not affect the translation of the source module, they are assembler directives in the sense that they instruct the assembler to pass information.

Pseudo-Op	Assembler Directive	Value Symbol	Description
.ENTO	Yes	No	Declare an overlay entry
.FORCE	Yes	No	Include a library module
.TSK	Yes	No	Specify number of task control blocks (TCBs) in program

Table 4.8 Binder directive pseudo-ops

Defining and Data

This chapter is in three sections. The first section explains how to define and use macros. The second section explains how to define symbols other than macros and pseudo-ops and how to redefine symbols other than pseudo-ops. The third section reviews the use of pseudo-ops that define the attributes of numeric and string data.

Macros

Macros are programming constructs that shorten the work of writing programs. A macro is a named series of source lines.

Macro Definitions

To associate a name with a macro string, use the .MACRO pseudo-op. The format for using .MACRO is

```
  .MACRO macro-name ↵
   macro-definition-string
%
```

where:

macro-name is the name you will use to refer to this macro. *Macro-name* must conform to the rules for symbols presented in Chapter 3. You must follow *macro-name* with a delimiter to clearly separate the macro name from its definition.

macro-definition-string consists of zero or more source statements. The assembler substitutes these statements for *macro-name* in your module.

% terminates the macro definition string.

The following source code defines and uses a simple macro.

```
        .MACRO  FIVES    ;The name of the macro is FIVES.
        5                ;The macro definition string consists of
        5                ;two data entries
   %                     ;End of macro definition string; not part of
                         ;macro body.

        FIVES            ;When the assembler encounters the macro
                         ;name FIVES, it substitutes the macro
                         ;definition string in your module (in
                         ;this case, two consecutive data entries).
```

Macro Continuations

You may extend a macro definition by including more than one macro-definition string, each beginning with the .MACRO pseudo-op and ending with the per cent sign macro terminator. Each macro name must be the same, and there may be no intervening macro definitions.

For example,

```
        .MACRO  JOHN

                .
                .
                .
%

        [program code]

        .MACRO  JOHN

                .
                .
%
```

will generate a macro John, but,

```
        .MACRO  JOHN

                .

%
        .MACRO  MARY

%
        .MACRO  JOHN

%
```

generates an error.

Redefining Macros

You can only redefine macros by using the .XPNG pseudo-op (see below). However, .XPNG deletes the entire symbol table.

Partial Lines

In most cases, you will write macro definitions in the form of one or more lines. You place the % immediately after the last line of the macro definition as the first item of the next line. You can also define a macro as zero or more lines followed by part of another line. You do this by placing the % immediately after the last character of the partial line. Any characters following the % will not be part of the macro definition. For example,

```
.MACRO      Z           ;This macro definition
STA         0,SV%       ;is made of a partial line

.MACRO      W           ;This macro definition
STA         0,SV0       ;is made of two
STA         1,SV1       ;full lines
%
.MACRO      R           ;This macro definition
STA         2,SV2       ;is made of a full
STA         3,SV%      ;line and a partial line
```

Special Characters

Within the macro definition string, two characters have special meanings: underscore (_) and uparrow(↑). The underscore (ASCII code 137$_8$) directs the assembler to store the next character without interpreting it. Thus, you usually use the underscore to store characters that otherwise have special significance when in a macro definition string. In other words, if you precede the characters %, _, or ↑ with an underscore, the macroassembler does not interpret them.

For example, if you want to place a percent sign in a source line, you must precede it with an underscore. If you do not, the assembler interprets % as the end of the macro. Thus, if you want to place the string

% MEANS PER 100

in a macro, enter

_% MEANS PER 100.

Also, by using the underscore and percent in this fashion, you may write one macro that creates a second macro.

If you place an underscore before a character that the assembler would not interpret anyway (i.e., a character other than %, _, or ↑), the assembler ignores the underscore. For example, the assembler interprets

```
    .MACRO  X
    A_B           ;The assembler removes_
%                ;from the symbol A_B.
```

as equivalent to

```
      .MACRO  X
      AB
%
```

Inside a macro, to use a symbol containing an underscore, include an extra underscore in the symbol. The first underscore directs the assembler to store the second one as part of the symbol. Thus, to indicate the symbol A_B in a macro, enter A__B.

The second character that has a special meaning inside macros is the uparrow (↑) (ASCII 136_8). You use this character when defining a macro that accepts arguments.

The assembler returns all characters in the macro definition string, except the underscore (_), uparrow (↑), and percent (%), exactly as it interprets these characters in non-macro source code.

Arguments in Macro Definitions

You may include formal (dummy) arguments in the macro definition string. When you call the macro, you supply an actual value for each formal argument. When the assembler expands the macro, it replaces the formal arguments in the macro definition string with the actual arguments in the macro call.

Within the macro definition string, all formal arguments begin with an uparrow (↑) (ASCII 136_8). There are three formats for formal arguments:

↑*n* where *n* is a digit from 0 to 9

↑*a* where *a* is a letter from A to Z or a to z.

↑*?a* where *a* is a single character from the following set: A - Z, a - z, 0 - 9,?

A digit following ↑ represents the position of an actual argument in the macro call's argument list. That is, when the assembler expands the macro, it replaces all occurrences of ↑*n* with the n^{th} actual argument in the macro call.

For example, in the following macro, the formal argument ↑2 appears in the macro definition string. When you call the macro, the assembler replaces ↑2 with the second argument in the macro call.

```
        .MACRO  TWO     ;Define macro TWO.
        A = ↑2          ;A equals the second argument you
                        ;pass to macro TWO.
   %                    ;Macro terminator (not part of macro).

        TWO     3,4

        A = 4
```

In the above example, the assembler substitutes the second argument for ↑2 and therefore A now equals 4.

The ↑*n* format allows you to reference only the first nine arguments to the macro (↑1, ↑2, ..., ↑9). Since the assembler allows you to supply up to 63_{10} arguments, you must use the ↑*a* and ↑?*a* formats to represent arguments 10 through 63.

The *a* or ?*a* following ↑ is a user symbol whose value the assembler looks up when expanding the macro. The value of the symbol indicates the position of the actual macro argument that replaces it (as in ↑*n*). The value for *a* or ?*a* must be in the range 0-63.

The following example illustrates the use of ↑*a* and ↑?*a* within a macro definition string.

```
        D=1                     ;Initialize symbols D and ?N.
        ?N=3

        .MACRO      PLUS        ;Define macro PLUS.
        X=↑D + ↑?N              ;X is the sum of two arguments.
    %                           ;Macro delimiter.

        PLUS        2,4,5       ;Call macro PLUS with three arguments.
        X=2+5                   ;X is the sum of two arguments.
```

When the macroassembler expands the macro, D equals 1 and ?N equals 3. Thus ↑D evaluates to ↑1 and ↑?N evaluates to ↑3. After the call to macro PLUS, X has the value of the first argument plus the third argument (X=2+5).

A zero or negative value following an uparrow (e.g., ↑0, or ↑I, where I= −5) is unconditionally replaced by the null string (a string with no characters). Similarly, the macroassembler substitutes the null string for any formal argument value that is larger than the number of actual arguments you supply to the macro call. For example, the macroassembler substitutes the null string for ↑3 if you supply only two arguments when calling the macro. These rules apply to all three formal argument formats (i.e., ↑*n*, ↑*a*, and ↑?*a*).

Macro Calls

After defining a macro, you insert the macro name wherever you want to insert the macro definition string in your module. The macro name and arguments, which need not be kept on one line, are a *macro call.*

A macro call has one of three formats — the macro name with no argument, the name followed by an argument list, or the name followed by a bracketed argument list:

macro-name
macro-name *arg1...arg n*
macro-name *[arg1...arg n]*

where

macro-name is the name of the macro, and

arg_n is an actual argument that replaces the appropriate formal argument during macro expansion.

During macro expansion arg_1 replaces every occurrence of ↑1 (or ↑*a* where *a*=1 within the macro; arg_2 replaces every occurrence of ↑2 within the macro. In general, arg_n replaces every occurrence of ↑*n* (or the equivalent of the *n*th formal argument) within the macro.

You use the first form of a macro call for macros having no formal arguments within their definitions or for macros accepting null arguments.

You can use any of the macro forms to call a macro which requires arguments. However, if you use the first form, all formal arguments are replaced by null strings. In the second form, a New-Line character or semicolon terminates the argument list; in the third form, a right bracket (**]**) terminates the argument list. If your arguments do not fit on one line, use the third form. In that form, a New-Line character acts just like a comma character: it serves only as a terminal between arguments. You must be sure not to separate the last argument on a line and the New-Line character with any commas, since the macroassembler assumes the intervening commas represent other arguments. For example:

```
ABC [1,2⤸ 3,4]   ;Specifies 4 arguments
ABC [1,2,⤸ 3,4]  ;Specifies 5 arguments
```

In the first example, the New-Line character separates the second and third arguments. In the second example, the New-Line character separates the third and fourth arguments. In this example, the third argument follows the second comma and is a null argument.

NOTE: *If you use the third form and some characters follow your argument list, the macroassembler lists the characters after the macro expansion.*

Using the third form, if you begin your macro reference with the special atom **, then the macroassembler suppresses the listing of the first line of the macro call (the line containing the **). Arguments appearing on other lines, plus any characters following the argument list, still appear in the listing. If you begin a macro reference line with ** and suppress the listing of the macro expansion using the .NOMAC pseudo op, all lines in the macro reference and any characters following the argument string are suppressed.

Macro definition and reference:

```
        .MACRO      NIUN
        LDA         ^1,^2
        LDA         ^3,^4
        MOV         ^5,^6
        JMP         ^7
%
**      NIUN [1,MRI,
        2,SUON
                    1,3 ↲
                    MAR]      ;This is the comment string ↲
```

Expansion with .NOMAC 0:

```
00000 024000       LDA      1,JIR
00001 024000                MAR] LDA 1,MRI
00002 030000       LDA      2,SHON
00003 135000       MOV      1,3
00004 000000       JMP      MAR
                   ;This is the comment string
00005 044000       STA      1,KSRTH
```

Expansion with .NOMAC 1:

```
00006 024000       LDA      1,JIR
                   ;This is the comment string
00013 044000       STA      1,KSRTH
```

Listing Macro Expansions

Macro definitions replace macro calls in the object file and in macro expansion listings. The listing shows both macro calls and macro expansions; the object file, however, contains only the object code for the macro expansions with actual arguments.

For example, consider the trivial macro

```
   .MACRO   DSP
 ^1%
```

which simply expands to its first argument. A source line using the number 121 as an argument to the macro DSP would be

```
LDA    0,DSP[121],3
```

The listing line would show the macro and argument expanded to just the argument, that is, DSP[121] would expand to 121. The source listing would show both the macro and it expansion:

```
LDA    0,DSP[121] 121,3
```

The expanded line to be translated to the object file would be

```
LDA    0,121,3
```

You can use the pseudo-op .NOMAC to suppress the listing of macro expansions. If you suppress macro expansions, the load instruction in the example above appears in the listing exactly as it does in the source listing line.

Macro Related Pseudo-Ops

In addition to .MACRO and .NOMAC (described above), the assembler provides two other pseudo-ops you may use with macros: .ARGCT and .MCALL.

The .ARGCT pseudo-op is a value symbol that returns the number of actual arguments you pass to a macro. Use this symbol inside the macro definition string. For example:

```
        .MACRO X
        ↑1+↑2
        (.ARGCT)
%

        X 4,5     ;Pass two arguments to X
        4+5
        (.ARGCT)  ;At expansion time, the value for
                  ;.ARGCT is 2 because macro X was
                   ;called with two arguments.
```

The .MCALL pseudo-op is also a value symbol that you may use inside a macro definition string. This symbol has the value 0 during the first call to that macro on the current assembler pass. The symbol has a value of 1 during subsequent calls in the current pass and −1 outside a macro. For example:

```
        .MACRO   Y
        .IFE     .MCALL     ;Assemble all code up to .ENDC only
        JSR      @FIRST     ;if the value of .MCALL equals zero.
        .ENDC
%
```

The first time you call macro Y, .MCALL equals 0. Thus, the .IFE condition will be true and the macroassembler will assemble the statement in the conditional block (JSR @ FIRST). However, on subsequent calls to macro Y, .MCALL will equal 1 and the macroassembler will not assemble the .IFE block.

Loops and Conditionals in Macros

When you use a .DO loop or an .IF conditional inside a macro, be sure you include a corresponding .ENDC pseudo-op in that same macro. The assembler reports an error if it encounters the macro definition terminator (%) before .ENDC. In addition, the macroassembler takes one of the following actions:

- If there is a .DO statement inside a macro with no corresponding .ENDC, the assembler interprets it as .DO 0 or .DO 1, depending on the loop count.
- If you do not terminate an .IF conditional inside a macro, the macroassembler ends the conditional immediately before the macro definition terminator (%).

The following example shows the proper use of a .DO loop

```
        .MACRO     LOOP
        .DO        ↑1       ;When you call this macro, the first
            3               ;argument indicates how many times to
            4               ;assemble the .DO loop.
        .ENDC               ;The end of the .DO loop is inside the
%                           ;macro definition string.
```

The following code shows the correct use of an .IF pseudo-op inside a macro:

```
        .MACRO   COND            ;If the first argument you pass to this
        .IFE     ↑1              ;macro equals 0, the macroassembler assembles
        10                       ;the data entries 10 and 20.  Note that the
        20                       ;.ENDC  statement appears inside the macro.
        .ENDC
        30                       ;The macroassembler assembles data entries 30
        40                       ;and 40 regardless of the argument value you
                                 ;pass to this macro.
  %
```

Macro Examples

The first example is a macro which computes the logical OR of two values. To call this macro, use the form:

OR *acs acd*

where

acs is the source accumulator, and

acd is the destination accumulator.

```
          .MACRO     OR
          COM        ^1,^1       ;Complement AC^1
          AND        ^1,^2       ;Clear "on" bits of AC^1
          ADC        ^1,^2       ;OR result to AC^2
     %
```

FACT is a macro that computes the factorial of a number. The factorial of a number, *n*!, is the value:

$n! = n*(n-1)*(n-2)*(n-3)*...*(2)*(1)$

The macro uses the *recursive* formula

$n! = n*(n-1)!$

to calculate the factorial value, where *n* is the input integer. *Recursive* means that the macro calls itself repeatedly.

If the input integer is not 1, the macro cannot immediately return the value of the factorial because (*n*-1)! has to be calculated. The macro saves the value of the input integer, decrements it, and uses the decremented value as the new input integer when the macro calls itself to calculate (*n*-1)!.

If *n*-1 is not 1, the macro repeats the decrement and call procedure. When the macro calls itself with an input integer of 1, the macro can calculate 1!, return to the next higher level, calculate the next factorial using the saved value of the input integer for this level, return to the next level, and so on, until it calculates (*n*-1)! and finally *n*!. The format used for calling this macro is

FACT *n i*

where

n is the number to be factorialized, and

i will be the factorial of *n*.

The FACT macro is shown below.

```
      .MACRO        FACT
      .DO           ↑1==1           ;Is input integer 1?
      ↑2=            1              ;If so, set the initial value of the
      .ENDC                         ;factorial to be 1.
      .DO           ↑ 1 < > 1       ;If input integer is not 1, then
      FACT          ↑1-1,↑2         ;decrement it and
                                    ;call FACT again. Macroassembler
                                    ;saves the old value of the input integer
                                    ;for use when the macro returns to
                                    ;this level.
      ↑2=              ↑1*↑2        ;When input integer is 1, the value of
                                    ;the factorial becomes the current
                                    ;value of the factorial times the
                                    ;value of the input integer at
                                    ;this level.
      .ENDC
%
```

```
                FACT    6,I
000000          .DO     6 = =1
                I=      1
                .ENDC
000001          .DO     6 < >1
                FACT    6-1,I
000000          .DO     6-1==1
                I=      1
                .ENDC
000001          .DO     6-1 < >1
                FACT    6-1-1,I
000000          .DO     6-1-1 = =1
                I=      1
                .ENDC
000001          .DO     6-1-1 < >1
                FACT    6-1-1-1,I
000000          .DO     6-1-1-1==1
                I=      1
                .ENDC
000001          .DO     6-1-1-1 < >1
                FACT    6-1-1-1-1,I
000000          .DO     6-1-1-1-1 = =1
                I=      1
                .ENDC
000001          .DO     6-1-1-1-1 < >1
                FACT    6-1-1-1-1-1,I
000001          .DO     6-1-1-1-1-1 = =1
000001          I=      1
                .ENDC
000000          .DO     6-1-1-1-1-1 < >1
                FACT    6-1-1-1-1-1-1,I
                I=      6-1-1-1-1-1*I
                .ENDC
000002          I=      6-1-1-1-1*I
                .ENDC
000006          I=      6-1-1-1*I
                .ENDC
000030          I=      6-1-1*I
                .ENDC
000170          I=      6-1*I
                .ENDC
001320          I=      6*I
                .ENDC
                .END
```

The next example shows a macro which allows you to structure your programs with IF-THEN-ELSE statements. The macro requires five arguments. The first two are accumulators. The third is a condition that allows you to test the other two accumulators. Table 5.1 below describes the values the third argument can assume and the test each value specifies.

Value	Test Performed
0	Test if first accumulator > second accumulator
1	Test if first accumulator = second accumulator
2	Test if first accumulator < second accumulator
3	Test if first accumulator <> second accumulator

Table 5.1 Third argument values

If the test condition is true, then the macro jumps to the address given as argument 4 (the THEN address). If the test condition is not true, the macro jumps to the address given as argument 5 (the ELSE address). So the format of the actual macro call is

IF ac_1, ac_2, *test*,adr_t, adr_f

where

ac_1 and ac_2	are the two accumulators,
test	specifies the test you want to make,
adr_t	is the address the macro returns to if the test condition is true, and
adr_f	is the address the macro returns to if the test condition is false.

The IF-THEN-ELSE macro is shown below.

```
    .MACRO  IF
    .DO     ↑3=0              ;This is the GREATER
                                    ;THAN routine.
    SUBZ#   ↑1,↑2,SZC       ; If ↑1 >↑2 then go to
                                    ;THEN routine.
    JMP     ↑4                ; Else go to ELSE routine.
    JMP     ↑5
    .ENDC
    .DO     ↑3=1              ;This is the equal routine.
    SUB#    ↑1,↑2,SNR       ; If ↑1=↑2 then go to
                                    ;THEN routine.
    JMP     ↑4                ; Else go to ELSE routine.
    JMP     ↑5
    .ENDC
    .DO     ↑3=2              ;This is the LESS THAN
                                    ;routine.
    SUBZ#   ↑1,↑2,SEZ       ; If ↑1 < ↑2 then go to
                                    ;THEN routine.
    JMP      ↑4
    JMP      ↑5               ; Else go to ELSE routine.
    .ENDC
    .DO      ↑3=3             ;This is the NOT EQUAL
                                    ;routine.
    SUB#     ↑1,↑2,SZR      ; If ↑1 < >↑2 then go to
                                    ;THEN routine.
    JMP      ↑4               ; Else go to ELSE routine.
    JMP      ↑5
    .ENDC
%
```

Defining Symbols

You may define symbols with values

as labels
in assignment statements
as macros
with instruction definition pseudo-ops.

You may also build symbol names, using generated symbols and numbers.

In addition to the symbols that you define, there are symbols whose definitions are supplied by Data General in the internal and permanent symbol tables.

Procedures for defining labels, assignments, and macros are explained above and in Chapter 3. This section explains the rules for the redefinition of these types of symbols and for the automatic generation of symbol names.

The rules for the redefinition of symbols can be summarized as follows:

- Pseudo-ops cannot be redefined.
- Labels cannot be redefined.
- Macros can be extended, but not redefined without the use of .XPNG, which erases all symbol definitions.
- Semi-permanent symbols can be redefined. If the /M switch is used, semi-permanent symbols can be redefined only by using .XPNG.
- Symbols used only in assignments can be redefined.
- Externally defined symbols cannot be defined differently.

These rules are developed below.

Semi-Permanent Symbols

Semi-permanent symbols are symbols defined, using the symbol-definition pseudo-ops listed in Table 5.2. Refer to Chapter 7 for how to use these pseudo-ops. These symbols may be found in the permanent symbol file (MASM.PS, for instance) or in your source.

As their name implies, semi-permanent symbols are not as easily redefined as assignment symbols. Unless you wish to allow multiple definitions of the same symbol within a program, you cannot redefine a semi-permanent symbol without deleting the entire symbol table. (Use the /M switch described in Chapter 6 to prohibit multiple definition of semi-permanent symbols. The procedure for deleting a permanent symbol table and rebuilding a new one is also explained in Chapter 6.)

Pseudo-Op	Instruction
.DALC	Defines an ALC instruction
.DCMR	Defines a commercial MRI instruction
.DEMR	Defines an extended MRI instruction without accumulator field.
.DERA	Defines an extended MRI instruction requiring an accumulator field.
.DEUR	Defines an extended user instruction
.DFLM	Defines a floating load or store instruction requiring an accumulator
.DFLS	Defines a floating load or store instruction requiring no accumulator
.DIAC	Defines an instruction requiring an accumulator
.DICD	Defines an instruction requiring an accumulator and a count
.DIMM	Defines an instruction requiring an immediate field and an accumulator
.DIO	Defines an I/O instruction requiring a device code and no accumulator
.DIOA	Defines an I/O instruction specifying an accumulator and device code
.DISD	Defines an instruction with source and destination accumulators which does not allow skips
.DISS	Defines an instruction with source and destination accumulators which allows skips
.DMR	Defines an MRI instruction requiring a displacement and an index
.DMRA	Defines an MRI instruction requiring an accumulator, displacement, and index
.DTAC	Defines an instruction with source and destination accumulators
.DUSR	Defines a user symbol
.DXOP	Defines an instruction requiring source, destination, and operation fields
.XPNG	Removes all symbol definitions except pseudo-ops

Table 5.2 Symbol definition pseudo-ops

Redefining Symbols in Assignments

If you define a symbol in an assignment statement, you may redefine it at any point in your program. For example,

```
START:  A=3
        A=LDA 0,0
        A=9
```

At the end of this sequence, the symbol A has the value 9.

Defining Labels

A label symbolically names a memory location. By using labels, you can refer to locations without using numeric addresses.

Labels appear at the beginning of the source line and must be followed by a colon. All labels must conform to the rules for symbol names.

The following source lines show how to use labels:

```
BEGIN:    LDA    0,SEVEN
JUMP:     JSR    @17
SEVEN:    7
```

The value of a label equals the value of the location counter at the label. Since the macroassembler computes the label value prior to processing the rest of the source line, a label usually equals the address of the next storage location that the assembler creates.

According to these rules, the label BEGIN in the first line of the previous example receives the address of the assembled LDA instruction as its value. Location SEVEN contains the value 7.

Since some source lines do not generate storage words, a label is not necessarily associated with the source statement it appears in. For example,

```
          NREL      1
START:    .TITLE    MOD1
          NREL      0
```

Here, the first statement assigns the title MOD1 to the source module. This statement does not generate a storage word. Therefore, the label START receives as a value the address of the next location assembled.

Similarly, a label may appear alone on a line, in which case its value equals the address of the next storage location assembled.

```
LABEL1:
       LDA   0,1
```

In this example, the value of LABEL1 equals the address of the assembled LDA instruction.

```
LABEL1:
LABEL2:      LDA    0,1
```

Here both LABEL1 and LABEL2 equal the address of the LDA instruction.

You may place more than one label on a source line; all labels will receive the same value. For example:

```
LOOP1:LOOP2:LOOP3:  ADD 0,1
```

LOOP1,LOOP2, and LOOP3 all equal the memory address of the assembled ADD instruction.

Do not label lines that contain the following pseudo-ops: .GLOC, .LOC, .NREL, .ZREL. These pseudo-ops may alter the value of the location counter and labelling them may cause errors.

Generated Numbers and Symbols

The assembler offers two formats to automatically generate numbers and symbol names. The backslash (\) is used to generate numbers and symbols that increment by 1. The dollar sign ($) is used to generate alphanumeric labels.

Backslash

To generate number and symbols with the backslash, use the following format:

\symbol

At assembly time, the assembler replaces \symbol with a 3-digit number representing the value of *symbol*. The assembler uses the current input radix for this substitution and truncates the value of *symbol* to three characters, if necessary.

\symbol may stand alone in code to form an integer, or it may follow characters that, together with the value of \symbol, form a number or symbol. For example,

```
        A=2         ;Initialize A and B.
        B=1234
X\A:    1           ;X\A evaluates to the symbol X002
X\B:    1           ;X\B evaluates to the symbol X234
                    ;(the 1 is truncated from 1234).
        C=\A+\B     ;\A equals 002 and \B equals 234
                    ;so C equals 236.
        450.\A      ;450.\A evaluates to 450.002
```

The assembly listing for a generated number or symbol shows the replacement value and the *\symbol* designation. For example, the above section of source code would appear as follows in the assembly listing:

```
      000001          .NREL     1

      000002          A = 2
      001234          B = 1234
00000!000001 X\A002:  1
00001!000001 X\B234:  1
      000236          C = \A002+\B234
00002!041434          450.\A002
      020010
```

Table 5.3 shows the correspondence between source code, the assembly listing, and the cross-reference listing.

Source Code	Assembly Listing	Cross-Reference Listing
ONES=111	ONES=111	ONES
A\ONES	A\ONES 111	A111

Table 5.3 Generated symbols in source and listings

You may increment the symbol used for *\symbol* just as you would increment any other value. For example, the following code creates labels for a table. (The listing follows the code.)

Source Code

```
        .RDX           8
**      X=0                  ;Initialize counter X (** means
                             ;suppress listing of this line).
TABLE:  .DO            64.   ;Assemble this loop 64 (decimal) times.
A\X:           0             ;Create labels A000, A001, ..., A077.
**             X=X+1         ;Increment counter X.
**      .ENDC                ;End of .DO loop.
```

Assembly Listing

```
        000010          .RDX      8

        000100  TABLE:  .DO       64.
00000   000000  A\X000: 0
00001   000000  A\X001: 0
00002   000000  A\X002: 0
    .
    .
    .
00076   000000  A\X076:
00077   000000  A\X077: 0
```

Dollar Sign

The macroassembler replaces each occurrence of $ (except those in text strings) with three characters from the set 0-9, A-Z. The macroassembler determines which three characters to use by converting the count (in radix 36) of the total number of all macro calls to ASCII. In nested macros, the macroassembler saves the replacement value for $ in the outer macro when the inner macro is expanded.

Generally you should not use $ as the first character in a label, since the first replacement character may be a digit.

The following example shows how to use $ to generate labels.

```
                         .MACRO   LABGN
                L$:      0
                %

                         LABGN
00000   000000  L$001:   0

                         LABGN
00001   000000  L$002:   0

                         LABGN
00002   000000  L$003:   0
```

Specifying Data Attributes

The assembler recognizes several pseudo-ops that define the attributes of numeric and string data.

Radix Control

The radix control pseudo-ops allow you to specify both the input radix and the output radix for your source module. The assembler uses the input radix to interpret numeric expressions in your source code and the output radix to display numeric values in the various output listings.

Table 5.4 describes the two radix control pseudo-ops.

Input and output radices are entirely distinct, and you set them independently, using the .RDX and .RDXO pseudo-ops. You may specify input radices in the range 2-20 and output radices in the range 8-20. The default radix for both input and output is 8 (that is, octal).

You may use .RDX and .RDXO as value symbols. They return the current input and output radices, respectively.

Pseudo-Op	Assembler Directive	Value Symbol	Description
.RDX	Yes	Yes	Set or represent radix for numeric input conversion
.RDXO	Yes	Yes	Set or represent radix for numeric output conversion

Table 5.4 Radix control pseudo-ops

For information about number and integer-generating formats, see Chapter 3.

Text Strings

The text string pseudo-ops assemble character strings into their equivalent ASCII codes. That is, using these pseudo-ops, you may enter an ASCII string in your source code and have the assembler store its binary representation in memory.

Storage of a character requires 7 bits of an 8-bit byte. The leftmost (parity) bit can be set to 0, 1, even, or odd parity.

You can specify ASCII source text strings within source modules in several ways. Depending on which text string pseudo-op you use, you can set parity and change the way the assembler packs the text string. Table 5.5 lists the pseudo-ops that affect text strings.

Pseudo-op	Assembler Directive	Value Symbol	Description
.TXT	Yes	No	Set leftmost bit to 0
.TXTE	Yes	No	Set leftmost bit of each character for even parity
.TXTF	Yes	No	Set leftmost bit to 1 unconditionally
.TXTM	Yes	Yes	Specify left/right or right/left bytepacking within words
.TXTN	Yes	Yes	Terminate an even length byte string with no null bytes or two null bytes
.TXTO	Yes	No	Set leftmost bit of each character for odd parity

Table 5.5 Pseudo-op that affect text strings

Using .TXT, you can direct the assembler to store the binary equivalent of an ASCII text string in memory. The assembler always packs two characters into each 16-bit word when it stores a text string (i.e., one character per 8-bit byte).

By default, the assembler packs character bytes left to right within memory words. You may override this convention by issuing the .TXTM pseudo-op.

If a string has an odd number of characters, the assembler places a null (all zero) byte in the word with the last character. If the string has an even number of characters, the assembler places either no null bytes or two null bytes after the last character, depending on your directions (see .TXTN).

If you wish to store only one or two ASCII characters in memory, you do not have to use pseudo-ops. "Special Integer-Generating Formats" in Chapter 3 describes alternative methods for storing characters.

NOTE: *Do not use _, ↑, and % as characters in a .TXT string within a macro body. Also do not use :, ;, $, and \ as string delimiters.*

Executing the Assembler

This chapter explains how to execute the assembler and interpret its printed output. The first part of the chapter explains the command line used to invoke the assembler and the switches used on that command line.

The second part of the chapter explains how to interpret the printed output from the assembler.

Operating Procedures

The CLI command line that invokes the macroassembler is

```
XEQ MASM {function switch} pathname {arg-switch}
{pathname{arg-switch}}...
```

where:

XEQ — is a CLI command that executes a program. The single character X is an acceptable abbreviation for XEQ.

MASM — is the name of the macroassembler program (less .PR extension).

function switch — is one or more optional global switches (see Table 6.1).

pathname — is the pathname of a source file. You must include at least one source file in each MASM command line. If you include more than one source file, make sure that all but the last one end with the .EOF or .EOT pseudo-op; end the last file with .END.

arg-switch — is the /S local (argument) switch explained below.

When you enter the MASM command, the macroassembler assembles one or more source files and, depending on the command line switches you use, produces an object file and a variety of listings.

If you do not use any function switches, the object file has the name of the first filename without the local /S switch on the command line; the assembler does not produce an assembly listing; and it reports all errors to the console.

Simplest Use of Assembler

The simplest use of the assembler generally involves the inclusion in your source program of three assembler directives:

1. A title directive, with the format

   .TITLE *name*

   If you do not include this directive, the assembler assigns the title .MAIN. The module's title is then unlikely to be the same as the object filename.

2. A location directive, either .LOC, .ZREL or .NREL. If you do not include a location directive, the assembler begins assigning locations at absolute location zero.

3. A directive to indicate the end of the program, with the format:

   .END *label*

   where *label* is a symbolic name for the starting address for execution of the program.

If you are familiarizing yourself with assembly language programming of Data General computers, you can build simple test programs around this skeleton. Functional programs of course require a more developed framework, including a stack with stack and frame pointers, and so forth.

To assemble the source module, enter the command line

```
XEQ MASM source-file
```

where *source-file* is the name of the file that contains your source module. You may include or omit the .SR extension.

The assembler creates an object file with the name *source-file*.OB, and reports assembly errors to the console.

Chapters 3 through 5 explain all the assembler directives you may include in the source module. This chapter explains the command line switches you may use to modify the action of the assembler.

Command Line Switches

Command line switches are of two kinds: function switches and argument switches. Function switches (also called global switches) are applied to the program name argument, MASM, and set parameters that hold for an entire assembly. Argument switches (also called local switches) are applied to source file names and affect the assembly of that file only.

Function Switches

Table 6.1 describes the function switches you may use in a MASM command line.

Switch	Action
/8	Makes symbol names significant up to eight characters. Without this switch, they are significant to five characters. Pseudo-ops are always significant to five characters.
/B =*filename*	Names the object file *filename*.OB. The assembler does not add the .OB extension if the filename already has one. This switch overrides the .OB pseudo-op and the naming convention based on source file name.
/E	Reports errors to the console.
/E =*filename*	Creates file *filename* and reports all assembly errors to that file. If *filename* already exists, the macroassembler appends new error messages to the end of that file.
/F	Generates or suppresses formfeeds as required to produce an even number of assembly listing pages.
/K	Keeps the macroassembler's temporary symbol table file at the end of assembly. If this switch is not used, the temporary symbol file is deleted after the source module has been assembled.
/L	Produces an assembly listing and directs it to the line printer (@LPT).
/L =*filename*	Identical to the /L switch except that the listing will reside in *filename*. If the file already exists, the macroassembler appends the new listing to the end of that file.
/M	Causes the assembler to treat all semi-permanent symbol redefinitions as errors. Normally, the assembler allows you to redefine semi-permanent symbols.
/N	Tells the macroassembler not to create an object file. The assembler checks for errors and produces listings, but does not create an .OB file. Use this switch to check for errors or create a listing.
/O	Directs the assembler to include all source lines in the assembly listing, regardless of any directives to suppress listing that may appear in the source program. That is, the /O switch overrides the .NOCON, .NOLOC, .NOMAC and two asterisk (**) directives.
/P	Includes semi-permanent symbols in the cross-reference listing.
/PS =*filename*	Directs the assembler to use *filename* as the permanent symbol file for the current assembly. If you do not use this switch, the assembler uses MASM.PS as the permanent symbol file. If you use this switch with /S, the assembler creates a new permanent symbol table and places it in *filename*.
/R	Produces an object file even if there is an assembly error. By default, the assembler does not produce an object file when there are errors.
/S	Directs the macroassembler to create a permanent symbol file, called MASM.PS, for use in future assemblies. The assembler skips the second pass, does not produce an object file.
/S=ps_file	Has the same function as the /S switch, except that ps_file is used as the name of the new permanent symbol file. Ps_file may be a filename or pathname. The original copy of MASM.PS remains unchanged, and the old copy, if any, of ps_file is deleted.

Table 6.1 MASM command line function switches

Argument Switch

The assembler has only one argument switch, /S. When appended to a filename that is input as a source module to the assembler, it instructs the assembler to skip that file on the second pass of the assembly.

Any file specified with this switch must not generate any storage words. This switch is typically used on parameter definition and macro definition files. Skipping such a file on the second assembly pass does not hinder the assembly of the other files in the command line. It merely decreases the size of the output listing and reduces assembly time.

Filenames

MP/AOS employs filename conventions to help you know a file's contents at a glance. Filenames end with a period and an extension that indicates their contents.

Table 6.2 summarizes the filename conventions that are relevant to assembly language programming. The table lists only the filename extensions, not the complete filename.

Extension	Contents of File
.OB	Object binary file (generated by language processor)
.PR	Executable program file (generated by binder)
.PS	Permanent symbol file (generated by assembler)
.SR	Assembly language source file

Table 6.2 Filename extensions

Normally the names of your source files end with the .SR extension, as in *filename*.SR. However, you do not need to specify the .SR extension in the command line. The assembler always searches for *filename*.SR first, in any case. If the assembler does not find it, it searches for *filename*.

For example, the following command lines are functionally equivalent:

```
XEQ MASM FILE1 FILE2


           XEQ     MASM FILE1.SR FILE2.SR
```

The object file normally receives the name of the first source file that does not include the /S switch in the command line. The .SR extension is replaced by the .OB extension. You may specify a different name for the object file by using the /B= switch or the .OB pseudo-op. Table 6.3 shows the file-naming priority employed by the macroassembler.

Do not confuse the object module's title, which you set using the .TITLE pseudo-op with the the object file's name. (See Chapter 4 for an explanation of module titles.)

Priority	Object Filename	Description
1 (highest)	/B=*filename*	The object file receives the name you specify with the/B= function switch on the assembler command line.
2	.OB *filename*	The object file receives the name you specify in an .OB pseudo-op in a source file.
3 (lowest)	Default filename	The object file receives the name of the first source file on the assembler command line, that does not have a local /S switch.

Table 6.3 Object filenames

Building a Permanent Symbol Table

The assembler builds a temporary symbol table file during each assembly. To build this table, the assembler uses the symbol file MASM.PS or whatever file you specify with the /PS= switch.

The two procedures described below explain how to add symbols to an existing symbol file and how to replace a symbol file with an entirely new one.

Adding to an Existing Symbol File

To add to an existing symbol file,

1. Write one or more *parameter files.* Such files contain symbol and macro definitions but no pure code.
2. Assemble the parameter file(s), using the /S switch (and the /PS switch to specify the symbol file if it is not MASM.PS) The /S switch causes the assembler to create a new symbol file by adding symbols defined in the parameter file(s) to those in the symbol table. This new symbol file is named MASM.PS (or the name you specified with the /S = switch), regardless of the name of the input symbol file. The assembler performs the first pass only, and does not produce an object file.

Creating a New Permanent Symbol File

To create a new permanent symbol file,

1. Write one or more *parameter files.* Such files contain symbol and macro definitions, but no code. Include the .XPNG pseudo-op at the beginning of the first of these files.
2. Assemble the parameter file(s), using the /S switch to specify the symbol file (or /S= switch if it is not MASM.PS). The .XPNG psuedo-op causes the assembler to delete the existing symbol table and to create a new one. The /S or /S= switch causes the assembler to build a new permanent symbol file. The new symbol file is named MASM.PS or *filename*, depending on whether you

used the /S or /S= switch. The assembler performs the first pass only, and does not produce an object file.

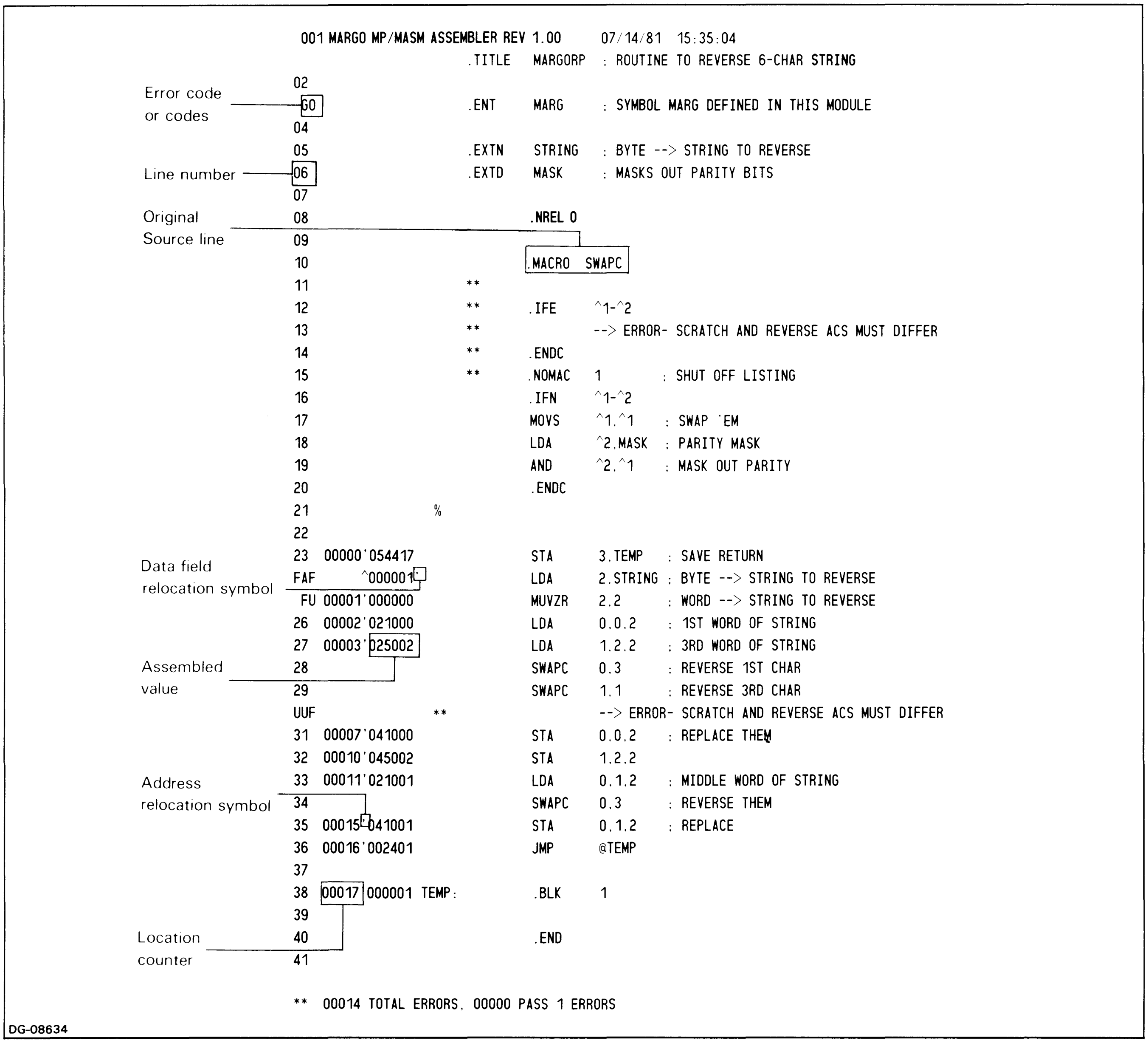
When building a permanent symbol file (.PS file), use the /8 switch (for 8-character symbols) whenever possible. This allows the permanent symbol file to be used for both 5- and 8-character assemblies. Otherwise, the .PS file is likely to work properly for 5-character assemblies only.

Interpreting Printed Output

The assembler produces four types of output; two of these (the object file and the permanent symbol file) are not printed but are disk files. The two types of printed output produced by the assembler are the assembly and cross-reference listing and the error listing. Both of these listings are optional, as explained below.

Assembly Listing

The assembly listing shows you how the assembler interpreted your source file. The listing is a series of lines, each divided into several fields. Table 6.4 lists the information contained in each field in an assembly listing, and Figure 6.1 is a sample assembly listing.



```
001 MARGO MP/MASM ASSEMBLER REV 1.00    07/14/81  15:35:04
                              .TITLE   MARGORP  ; ROUTINE TO REVERSE 6-CHAR STRING
02
60                            .ENT     MARG     ; SYMBOL MARG DEFINED IN THIS MODULE
04
05                            .EXTN    STRING   ; BYTE --> STRING TO REVERSE
06                            .EXTD    MASK     ; MASKS OUT PARITY BITS
07
08                                     .NREL 0
09
10                                     .MACRO  SWAPC
11                            **
12                            **       .IFE     ^1-^2
13                            **                --> ERROR- SCRATCH AND REVERSE ACS MUST DIFFER
14                            **       .ENDC
15                            **       .NOMAC   1         ; SHUT OFF LISTING
16                                     .IFN     ^1-^2
17                                     MOVS     ^1,^1     ; SWAP 'EM
18                                     LDA      ^2,MASK   ; PARITY MASK
19                                     AND      ^2,^1     ; MASK OUT PARITY
20                                     .ENDC
21                       %
22
23  00000'054417                       STA      3,TEMP    ; SAVE RETURN
FAF       ^000001'                     LDA      2,STRING  ; BYTE --> STRING TO REVERSE
 FU 00001'000000                       MUVZR    2,2       ; WORD --> STRING TO REVERSE
26  00002'021000                       LDA      0,0,2     ; 1ST WORD OF STRING
27  00003'025002                       LDA      1,2,2     ; 3RD WORD OF STRING
28                                     SWAPC    0,3       ; REVERSE 1ST CHAR
29                                     SWAPC    1,1       ; REVERSE 3RD CHAR
UUF                      **                     --> ERROR- SCRATCH AND REVERSE ACS MUST DIFFER
31  00007'041000                       STA      0,0,2     ; REPLACE THEM
32  00010'045002                       STA      1,2,2
33  00011'021001                       LDA      0,1,2     ; MIDDLE WORD OF STRING
34                                     SWAPC    0,3       ; REVERSE THEM
35  00015'041001                       STA      0,1,2     ; REPLACE
36  00016'002401                       JMP      @TEMP
37
38  00017 000001 TEMP:                 .BLK     1
39
40                                     .END
41

**  00014 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

DG-08634

Figure 6.1 Assembly listing

Columns	Information Contained
1-3	If there are assembly errors on this line, error codes appear in these columns. The first error is reported in column three, the second in column two, and the third in column one. If there are more than three errors, error flags do not appear here but are recorded in the total error count. If there are no errors, these columns contain a two-digit number indicating the line number on current listing page and a blank space.
4-8	The value of the location counter (address), if applicable. If the current source statement generates more than one word of code, the value in these columns is the address of the first word. If the source statement does not generate any storage words, these columns are left blank.
9	A one-character symbol indicating the relocation base of the address in columns 4-8 (see Table 6.5).
10-15*	Data field for the 16-bit word of the assembled instruction or expression, or the 16-bit word or the value on the right side of an assignment statement or pseudo-op argument. If the assembled value requires two or more 16-bit words (as in .TXT or extended instructions) the second word is listed in columns 10-15 on the next lines of the listing.
16	A one-character symbol indicating the relocation base of the value in columns 10-15 (see Table 6.6).
17. . .	The source statement exactly as written or expanded by macro calls.

Table 6.4 Assembly listing fields

**Certain source statements do not generate storage words in the object module. For these lines, the listing data field contains the value of an argument or other relevant expression. For example:*

```
000001 .NREL 1
```

The data field contains the value of the argument to NREL, namely 1.

Character	Meaning
(space)	Absolute relocation
—	Page zero relocation
=	Page zero byte relocation
'	NREL 0 word relocation (impure code)
!	NREL 1 word relocation (pure code)
"	NREL 0 byte relocation (impure code)
&	NREL 1 byte relocation (pure code)
$	External symbol relocation
U	Undefined symbol (on cross-reference listing)

Table 6.5 Address and data field relocation base symbols

Assembly Listing Control

The assembler does not automatically produce an assembly listing. If you want one, you must include either the /L or the /L=*filename* on the MASM command line. The /L switch directs the assembler to send the listing to @LPT; the /L=*filename* switch sends the listing to a specified file.

The assembler provides several directives that allow you to manipulate the contents and formats of the assembly listing. Table 6.6 lists switches and pseudo-ops that affect the assembly listing.

These modifiers affect only the listing, not the object file.

Pseudo-Op	Description
**	Suppress listing of source line that contains this directive.
.EJECT	Begin a new page in assembly listing
.NOCON	Enable or suppress the listing of conditional source lines
.NOLOC	Enable or suppress the listing of source lines that lack location fields
.NOMAC	Enable or suppress the listing of macro expansions
.RDXO	Specify the radix for numeric values in the output listings.

Table 6.6 Pseudo-ops that control assembly listings

Cross-Reference Listing

The cross reference listing provides an alphabetic list of symbols and their values, and also shows the page and line numbers of the assembly listing in which the symbols appear.

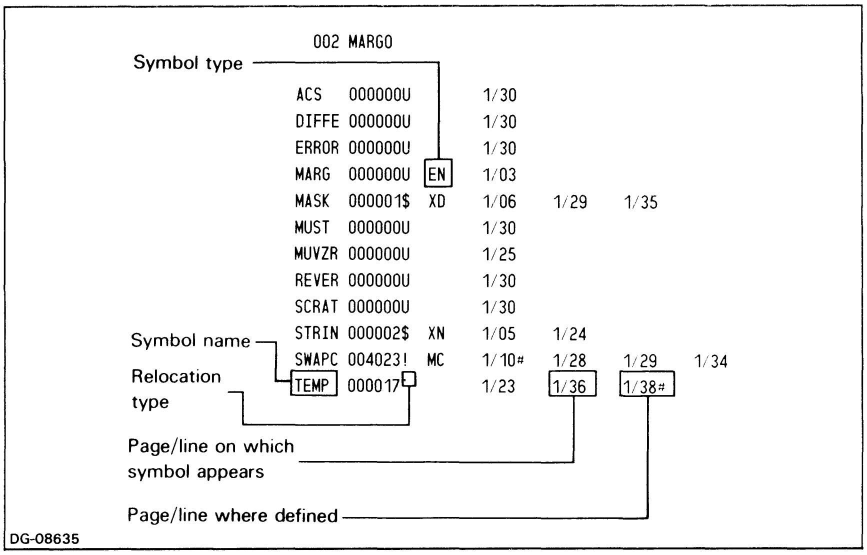
In addition, the cross-reference listing also indicates the page and line on which you defined (or redefined) the symbol (if applicable). The macroassembler signals the defining reference(s) by placing a number sign (#) after the appropriate page/line indicator.

The cross-reference listing includes several symbol type flags that provide additional information about the symbols in your program. Table 6.7 lists the assignment mnemonics and their meanings.

These mnemonics appear in the cross-reference immediately after the symbol's value. Figure 6.2 is a sample cross reference listing.

Character	Meaning
(spaces)	Local symbol
EN	Entry symbol (defined in .ENT statement)
EO	Overlay entry (defined in .ENTO statement)
XD	External displacement (defined in .EXTD statement)
XN	External normal (defined in .EXTN statement)
NC	Named common (defined in .COMM statemet)
MC	Macro name

Table 6.7 Cross-reference assignment mnemonics



```
        002 MARGO

       ACS   000000U        1/30
       DIFFE 000000U        1/30
       ERROR 000000U        1/30
       MARG  000000U  EN    1/03
       MASK  000001$  XD    1/06    1/29    1/35
       MUST  000000U        1/30
       MUVZR 000000U        1/25
       REVER 000000U        1/30
       SCRAT 000000U        1/30
       STRIN 000002$  XN    1/05    1/24
       SWAPC 004023!  MC    1/10#   1/28    1/29    1/34
       TEMP  000017         1/23    1/36    1/38#
```

Figure 6.2 Cross-reference listing

Cross-Reference Listing Control

The assembler produces a cross-reference listing only when it generates an assembly listing. Thus, to receive a cross-reference listing, you must include the /L or /L=*filename* switch on the MASM command line. The cross reference appears after the assembly listing in the same file.

The cross-reference listing normally contains only user symbols. By using the */P* switch on the command line, you can direct the macroassembler to include semi-permanent symbols in the listing.

Error Listing

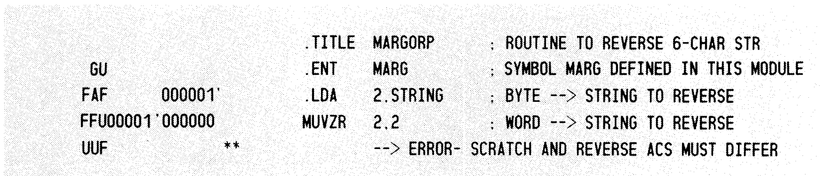
The error listing shows the title of the source file and lists all lines that contained errors. It does not contain any information in addition to that found in the assembly listing, but it provides you with a short summary of problem areas. Figure 6.3 shows the error listing generated by the program shown in Figure 6.1.

Error Listing Control

To produce an error listing, include either the /E or /E= *filename* switch. The /E switch sends output to the console. The /E = *filename* switch sends the output to *filename*.

When you produce an assembly listing (i.e., issue the /L or */L=filename* function switch), the macroassembler reports all errors to the assembly listing file and, if you include the /E switch, to the error listing file. Within the assembly listing, the error count appears after the source code listing.

Table 6.8 lists error codes. Appendix B contains complete descriptions of error conditions.



```
                              .TITLE  MARGORP      ; ROUTINE TO REVERSE 6-CHAR STR
   GU                         .ENT    MARG         ; SYMBOL MARG DEFINED IN THIS MODULE
  FAF      000001'            .LDA    2.STRING     ; BYTE --> STRING TO REVERSE
  FFU00001'000000             MUVZR   2.2          ; WORD --> STRING TO REVERSE
  UUF             **                  --> ERROR- SCRATCH AND REVERSE ACS MUST DIFFER
```

DG-08636

Figure 6.3 Error listing

Symbol	Error Code
A	Address error
B	Bad character, bad line
C	Macro error
D	Radix error
E	Equivalence error
F	Format error
G	Global reference error
K	Repetitive or conditional assembly error
L	Location counter error
M	Multiply-defined symbol error
N	Number error
O	Field overflow error or stack error
P	Phase error
Q	Questionable line error
R	Relocation error
U	Undefined symbol error
V	Variable label error
X	Text error
Z	Illegal use of external

Table 6.8 Error codes

Pseudo-Op Dictionary

This chapter lists alphabetically by mnemonics all pseudo-ops that can be used with the macroassembler. The name and command-line format of the pseudo-op are given in each case, followed by a brief explanation of its use and, where relevant, an example.

In examples showing assembly listings, the line numbers have been omitted. Also, for ease of reading, extra spaces have been inserted between certain fields. See the preface for conventions used in command-line formats.

. (period) Current Location Counter

The period symbol (.) has the value and relocation property of the current location counter. The location counter is an assembler variable that holds the address and relocation base of the next memory location the macroassembler will assign.

Example

```
           000001   .NREL 1
000000  !  000003   3
           000003   .LOC .+2   ;Set the location counter to
000003  !  020010   LDA 0,10   ;its current value plus two
                               ;words; the relocation base
                               ;does not change.
```

.ARGCT

Number of Arguments Passed to Macro

The pseudo-op .ARGCT is a value symbol. Its value equals the number of arguments you passed to the macro containing it. For example, if you pass three arguments to a macro, then the symbol .ARGCT has the value 3 for that macro expansion.

If you use .ARGCT outside a macro, its value is -1.

Example 1

```
         000000     .NREL 1
                    .MACRO ARG       ;Define macro ARG.
                    ↑1+↑2
                    (.ARGCT)
                %

                    ARG  4,5         ;Call ARG with 2 arguments
000000 ! 000011     4+5
000001 ! 000002     (.ARGCT)         ;(value of .ARGCT is 2).
```

Example 2

```
                    .MACRO ARG       ;Define macro ARG.
                    .IFE   .ARGCT    ;If you call ARG with no
                    10               ;arguments, assemble the
                    .ENDC            ;value 10.  Otherwise,
                    .IFN    .ARGCT   ;assemble the value of
                    ↑1               ;the first argument.
                    .ENDC
                %
```

.BLK Reserve a block of memory

.BLK *abs-expr*

Reserves a block of memory words. *Abs-expr* specifies the length (in 16-bit words) of this block. *Abs-expr* must be a non-negative absolute expression.

The assembler increments the current location counter by *abs-expr* when it encounters .BLK in your source.

Example

```
                          .NREL      0
00000 ' 040405             STA       0, F
00001 ' 044405             STA       1, F + 1
00003 ' 050405             STA       2, F + 2
00004 ' 054405             STA       3, F + 3
00005 ' 000004 F:         .BLK 4
00011 ' 034510             LDA       3,110
```

Reserve a Labeled Common Area

.COMM

.COMM *usym abs-expr*

Reserves a labeled (or named) common area for intermodule communication. A common area is a data storage area that you may access from separately assembled modules in your program.

The assembler assigns the name *usym* (user symbol) to this common area. The assembler regards *usym* as an entry point and, therefore, you should not redefine this symbol anywhere in your program.

Specify the size of the common area (in 16-bit words) in the *abs-expr* argument. This argument must be a positive absolute expression.

To reference this common area from another module in your program, use .COMM, or .EXTN, to declare *usym* as externally defined. If you issue the same .COMM statement in two separately assembled modules, the binder resolves them to the same area in memory. If the areas allocated are of different sizes, the binder uses the largest.

Example

```
.TITLE   A        ;Module A.
.COMM    X,30     ;Reserve a common area named X
                  ;of length 30 words.
.COMM    Y,20     ;Common area Y contains 20 words.
         .
         .
         .
.END


.TITLE   B        ;Separately assembled module B.
.COMM    X,30     ;X refers to the same common area as
         .        ;declared in module A.
         .
         .
.END


.TITLE   C        ;Separately assembled module C.
.EXTN    X        ;X is defined in a different module
         .        ;and, in this case, refers to the
         .        ;starting address of the common
         .        ;area declared in module A.
.END
```

.CSIZ Reserve an Unlabeled Common Area

.CSIZ *abs-expr*

Reserves an unlabeled common area for intermodule communication. A common area is a data storage area that you may access from separately assembled modules in your program.

The size of this unlabeled common area is equal to the number of 16-bit words you specify in the *abs-expr* argument. This argument must be a non-negative absolute expression.

The binder assigns the name ?CLOC to the starting address of your unlabeled common area. To reference this common area from a separately assembled module, declare ?CLOC, using an .EXTN or .EXTD pseudo-op.

If you include more than one .CSIZ pseudo-op in a single source module, the macroassembler uses the largest value as the size of the unlabeled common area. Similarly, if separately assembled modules issue .CSIZ pseudo-ops, the binder uses the largest value.

Example

```
.TITLE   A            ;Module A.
.EXTN    ?CLOC
.CSIZ    100          ;Reserve an unlabeled common area
         .            ;100 words long.
         .
         .
.END


.TITLE   B            ;Separately assembled module B.
.EXTN    ?CLOC        ;?CLOC is the starting address of
         .            ;the unlabeled common area declared
         .            ;in module A.
         .
.END
```

.DALC

Define ALC Instruction

.DALC *usym*={*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) You must specify source and destination accumulators, and you may specify an optional skip field. Specify these fields with the following format:

usym{c}{sh}{#} acs acd {skip}

The fields assemble as shown below.

	ACS	ACD		SH	C	#	SKIP
0	1 2	3 4	5 7	8 9	10 11	12	13 15

NOTE: *The skip character # may be anywhere as a break character. It assembles as a 1 in bit 12.*

If you define a 3-character *usym* with this pseudo-op, you can add mnemonics to manipulate the value of carry and to shift the specified data, thus forming a 4- or 5-character instruction mnemonic. Table 7.1 list the mnemonics that affect the carry. Table 7.2 lists the mnemonics that perform shift operations.

Table 7.3 lists the mnemonics that specify an optional skip test.

Mnemonic	Sets Bits 10-11 to	Action
—	00	No action
Z	01	Sets carry to zero
O	10	Sets carry to one
C	11	Complements carry

Table 7.1 Carry mnemonics

Mnemonic	Sets Bits 8-9 to	Action
—	00	No action
L	01	Shifts data left one bit
R	10	Shifts data right one bit
S	11	Swaps data bytes

Table 7.2 Shift mnemonics

Mnemonic	Sets Bits 13-15 to	Action
—	000	No skip
SKP	001	Skip unconditionally
SZC	010	Skip if carry bit is zero
SNC	011	Skip if carry bit is one
SZR	100	Skip if ALC result is zero
SNR	101	Skip if ALC result is nonzero
SEZ	110	Skip if either ALC result or carry bit is zero
SBN	111	Skip if both ALC result and carry bit are nonzero

Table 7.3 Mnemonics for optional skip field

Example

```
         103000  .DALC   ADD=103000
   00000 103000   ADD    0,0           ;These three
                                       ;statements
   00001 103002   ADD    0,0,SZC       ;specify fields
                                       ;correctly.
   00002 133001   ADD    1,2,SKP
 F 00003 123000   ADD    1             ;These two
                                       ;statements
FF 00004 103000   ADD                  ;do not specify
                                       ;fields correctly.
```

Define Commercial Memory Reference Instruction

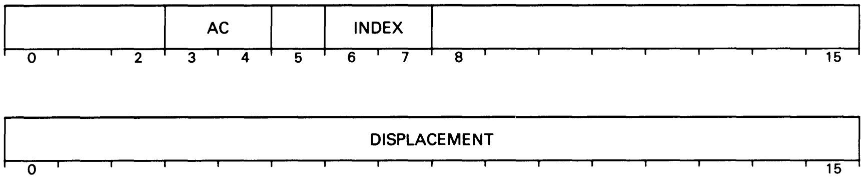
.DCMR

.DCMR *usym* = {*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* requires an accumulator and a displacement. You may specify an optional index field as well. Use the following format:

usym ac disp {index}

The three fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
         102170           .DCMR   ELDB=102170
         000001           .NREL   1
00000 !  112570           ELDB    2,PT1        ;AC2 contains the
         001376                                ;character A
00002 !  113570           ELDB    3,PT2        ;AC3 contains the
         001373                                ;character B
         .
         .
         .
00600 !  040502  ALPHA:    .TXT    "AB"
         001400&           PT1=    ALPHA*2
         001401&           PT2=    ALPHA*2+1
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

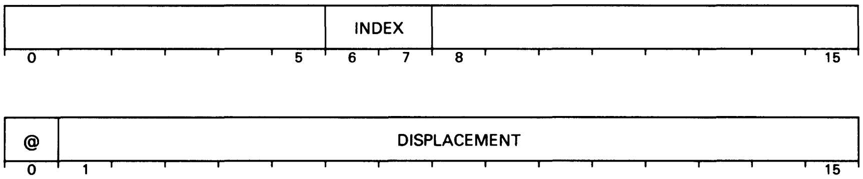
.DEMR Define Extended Memory Reference Instruction

.DEMR *usym* = {*inst* / *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* requires a displacement. You may specify an optional index field as well. Use the following format:

usym disp {*index*}

The two fields assemble as shown below.



NOTE: *You can use @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 of the second word to 1.*

Once defined, *usym* is an instruction mnemonic.

Example

```
          102070       .DEMR   EJMP=102070
                       .EXTN   ADDR
          000001       .NREL   1
  00000 ! 102070       EJMP    ADDR
          000001$
  00002 ! 102470       EJMP    .+3
          000002
F 00004 ! 102070       EJMP    2,.+3        ;This specifies an
          000002                            ;accumulator, which is
                                            ;incorrect.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define Extended Memory Reference Instruction Requiring an Accumulator

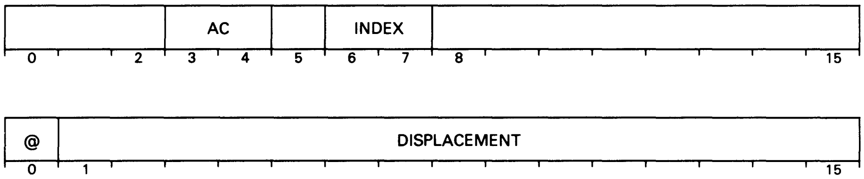
.DERA

.DERA *usym* = {*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* requires an accumulator and a displacement. You may specify an optional index field as well. Use the following format:

usym ac disp {index}

The three fields assemble as shown below.



NOTE: *You can use @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 in the second word to 1.*

Once defined, *usym* is an instruction mnemonic.

Example

```
             122070   .DERA   ELDA=122070
             000001   .NREL   1
    000000 ! 122470    ELDA    0,.+3
             000002
FFO 000002 ! 122070    ELDA    .+3         ;This specifies the
             000000                        ;wrong number of
                                           ;arguments.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

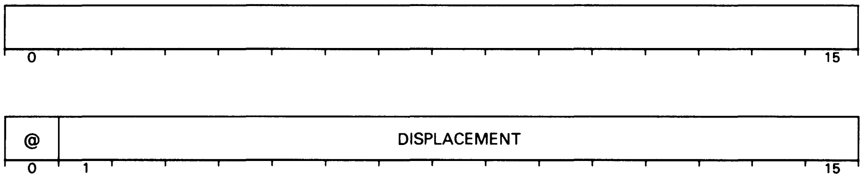
.DEUR Define Extended User Instruction

.DEUR *usym* = {*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* requires a displacement. Use the following format:

usym disp

The field assembles as shown below.



NOTE: *You can use @ anywhere in the displacement field to indicate indirect addressing. This atom sets bit 0 in the second word to 1.*

Once defined, *usym* is an instruction mnemonic.

Example

```
           163710        .DEUR   SAVE=163710
           061777        .DEUR   VCT=061777
           000001        .NREL   1
000000 !   163710        SAVE    4
           000004
           000000        SYMB    =0
000002 !   061777        VCT     SYMB
           000000
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define Floating Load or Store Instruction Requiring an Accumulator

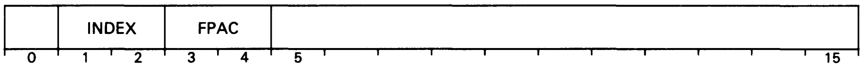
.DFLM

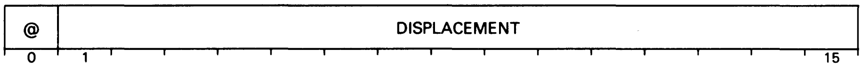
.DFLM *usym* = {*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*EXP* evaluates to the bit pattern of an instruction). *Usym* is a floating point load or store instruction requiring an accumulator and a displacement. You may specify an optional index field as well. Use the format shown below:

usym fpac disp {index}

The three fields assemble as shown below.





NOTE: *You can use the character @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 in the second word to 1.*

Once defined, *usym* is an instruction mnemonic.

Example

```
          102050   .DFLM  FLDS=102050
          000001   .NREL  1
  00000 ! 122050   FLDS   0,.+2
          000001
F 00002 ! 102050   FLDS   .+3          ;This specifies the
                                       ;wrong number
                                       ;of arguments.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

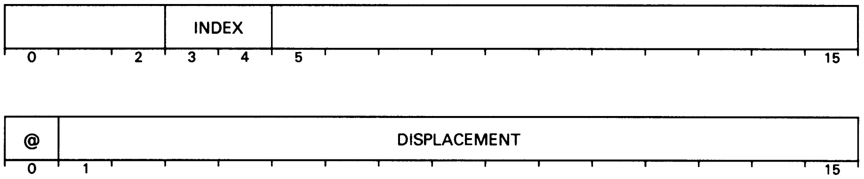
.DFLS Define Floating Load or Store Instruction

.DFLS *usym* = {*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) You may specify an optional index field as well. Use the following format:

usym disp {*index*}

The two fields assemble as shown below.



NOTE: *You can use @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 of the second word to 1.*

Once defined, *usym* is an instruction mnemonic.

Example

```
           123350     .DFLS  FLST=123350
           000000     .NREL  1
                      .EXTN  DR
   00000 ' 123350     FLST   DR
           000001$
F  00002 ' 123350     FLST                  ;This specifies no
           000000                           ;displacemennt
                                            ;field
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

.DIAC

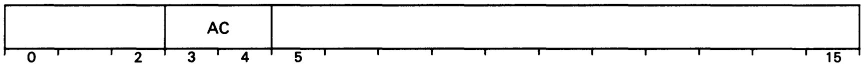
Define an Instruction Requiring an Accumulator

.DIAC *usym*={*inst* | *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is an instruction requiring an accumulator. Use the following format:

usym ac

The field assembles as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
      061000  .DIAC   CM=061000
      062000  .DIAC   MA=062000
00000 061000  CM      0             ;This command is
                                    ;DOA 0,0.
00002 066000  MA      1             ;This command is
                                    ;DOB 1,0.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

.DICD Define an Instruction Requiring an Accumulator and Count

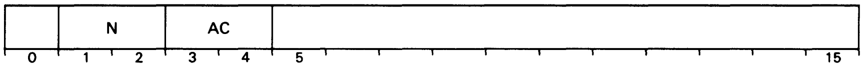
.DICD *usym*={ *inst* / *exp* }

Defines *usym* as a symbol with the value of an instruction or an expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is an instruction requiring count and destination fields. Use the following format:

usym n ac

where *n* is an integer between 1 and 4.

The two fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
        100010  .DICD   ADI=100010
  00000 104010  ADI     1,1
  00001 110010  ADI     1,2
  00002 160010  ADI     4,0
0 00003 104010  ADI     5,1         ;Specifies illegal
                                    ;count field.
F 00004 100010  ADI     1           ;Specifies too few
                                    ;arguments.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define an Instruction Requiring an Accumulator and an Immediate Value

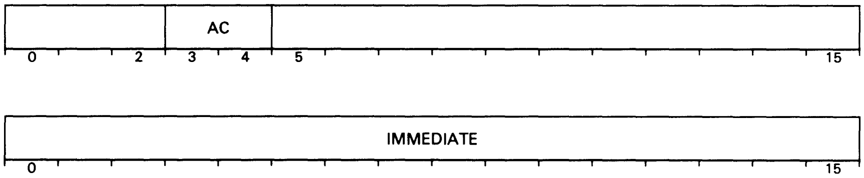
.DIMM

.DIMM *usym*={*inst* | *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* requires an accumulator and an immediate value. Use the following format:

usym immed ac

The fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
        163770  .DIMM   ADDI=163770
        000001  .NREL   1
  00000 ! 173770  ADDI    1002,2        ;This statement is
        001002                          ;correct.
F 00002 ! 163770  ADDI    0             ;Specifies incorrect
        000000                          ;number of arguments.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

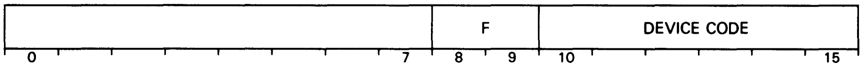
.DIO Define an I/O Instruction without an Accumulator

.DIO *usym* ={*inst* / *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction. *Usym* is an I/O instruction requiring a device code. Use the following format:

usym{f} device_code

The fields assemble as shown below.



NOTE: *If you define a three-character* usym with this pseudo-op, you can follow it immediately with one of the letters from Table 7.4. Each letter represents an optional function code which sets bits 8-9 of the instruction word. This procedure is shown in the example below.

Optional Mnemonic	Sets Bits 8-9 to	Action*
—	00	No action
S	01	Sets Busy flag, clears Done flag, starting device
C	10	Clears Done and Busy flags, idling device
P	11	Sets Done and Busy flags, pulsing I/O bus control line

Table 7.4 Function mnemonics

*The actions of these flags are device dependent. For a more detailed discussion of specific I/O devices, refer to the MP/AOS System Programmer's Reference, DGC No. 093-400051.

Once defined, *usym,* is an instruction mnemonic.

Example

```
.DIO   NIO=060000
.DIO   SKPBZ=063500
NIOS   14
NIO    14
SKPBZ  10
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define an I/O Instruction with an Accumulator

.DIOA

.DIOA *usym{f}={inst | exp}*

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is an I/O instruction which requires an accumulator and a device code. Use the following format:

usym{f} ac device_code

The fields assemble as shown below.

	AC		F	DEVICE CODE
0 2	3 4	5 7	8 9	10 15

NOTE: *If you define a three-character usym* with this pseudo-op, you can follow it immediately with one of the letters listed in Table 7.4. Each letter represents an optional function code which sets bits 8-9 of the instruction word. This procedure is illustrated in the example below.

Once defined, *usym,* is an instruction mnemonic.

Example

```
        060400  .DIOA   DIA=060400
        000001  .NREL   1
00000 ! 070410   DIA    2,TTI         ;Correct.
00001 ! 070610   DIAC   2,TTI         ;Correct.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

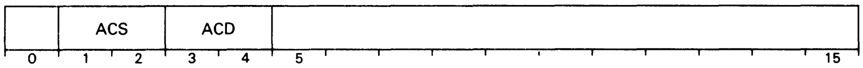
.DISD Define an Instruction with Source and Destination Accumulators

.DISD *usym*={*inst* | *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is an instruction requiring source and destination accumulators. You may not specify the load/no-load, carry, shift, or skip options. Use the following format:

usym acs acd

The fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
        102710          .DISD  LDB=102710
        000001          .NREL  1
00000 ! 030410           LDA   2,.PTR
00001 ! 146710           LDB   2,1         ;Byte addressed by AC2
                               .           ;is loaded into AC1.
                               .
                               .
00010 ! 000022&   .PTR:  .+1*2
00011 ! 040502           .TXT  "ABCDE"
        041504
        042400
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define a Skip Instruction with Source and Destination Accumulators

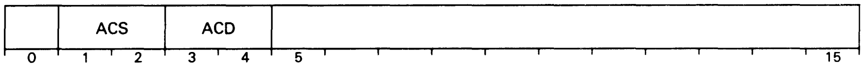
.DISS

.DISS *usym*={*inst* | *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the pattern of an instruction.) *Usym* is an instruction requiring source and destination accumulators. Note that this pseudo-op does not allow you to specify the load/no-load or skip options. Use the following format:

usym acs acd

The fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
         101010    .DISS   SGT=101010  ;This an ECLIPSE
                                       ;instruction
                   .NREL   1
  00000 ! 131010   SGT     1,2
F 00001 ! 121010   SGT     1           ;Not enough
                                       ;arguments
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

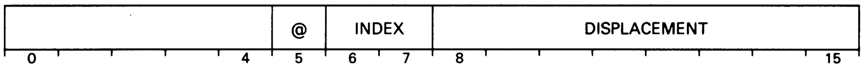
.DMR Define a Memory Reference Instruction with Displacement and Index

.DMR *usym*=*{inst | exp}*

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is a memory reference instruction requiring either a displacement or an address. You may also include an index. Use the following format:

usym displacement {index}

The fields assemble as shown below.



You can use the character @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 5 to 1.

Once defined, *usym* is an instruction mnemonic.

Example

```
          000001        .NREL  1
          000000        .DMR  JMP=000000
  00000 ! 000402   WIZ: JMP   .+2
F 00001 ! 000400        JMP   0,1,2        ;Incorrect number of arguments.
  00002 ! 003001        JMP   @1,2         ;Correct number of arguments.
                                           ;plus use
                                           ;of indirect flag.
  00003 ! 000775        JMP   WIZ          ;Correct number of arguments.
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

.DMRA

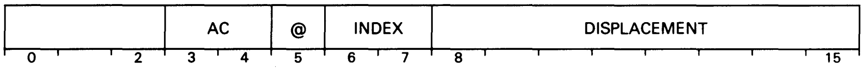
Define a Memory Reference Instruction Requiring an Accumulator

.DMRA *usym*={*inst* | *exp*}

Defines *usym* as a symbol with a value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is a memory reference instruction requiring an accumulator and a displacement. You may specify an optional index field as well. Use the following format:

usym ac displacement {*index*}

The fields assemble as shown below.



You can specify the atom @ anywhere in the instruction as a break character. This atom assembles a 1 in bit 5.

Once defined, *usym* is an instruction mnemonic.

Example

```
        000001  .NREL   1
        020000  .DMRA   LDA=20000
00000 ! 030204  LDA     2,.+4
00001 ! 025400  LDA     1,0,3
00002 ! 031401  LDA     2,1,3
00003 ! 033401  LDA     2,@1,3
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

.DO Assemble Source Lines Repetitively

.DO *abs-expr*

Directs the assembler to assemble a portion of your source module repetitively. The macroassembler assembles the source lines following .DO the number of times given in *abs-expr*. *Abs-expr* must be an absolute expression.

You must terminate the .DO loop with the .ENDC pseudo-op. Thus, the .DO portion of your module has the general form:

```
.DO abs-expr
      .         ;The assembler assembles these
      .         ;lines abs-expr times.
      .
.ENDC           ;Terminates the .DO loop.
```

You may use .DO to perform conditional assembly of source lines by passing a relational expression as an argument (pass an expression that contains <,>,==,<=,>=,or<>). If the relational expression is true, its value is 1 and the assembler assembles the .DO loop once. If the relational expression is false, its value is 0 and the assembler does not assemble the loop.

You may nest .DOs to any depth. Be sure the innermost .DO corresponds with the innermost .ENDC, etc. See "Loops and Conditionals in Macros" (Chapter 5) for more information.

Example 1

Source Code

```
         .NREL   0

FIRST:   1
SECOND:  3
SUM:     0
         .DO      3
         LDA     1,FIRST
         LDA     2,SECOND
         ADD     1,2
         STA     1,SUM
         .ENDC
         .END
```

Assembly listing

```
        000000          .NREL   0
00000 ' 000001  FIRST:  1
00001 ' 000003  SECOND: 3
00002 ' 000000  SUM:    0
        000003          .DO     3
00003 ' 024775          LDA     1,FIRST
00004 ' 030775          LDA     2,SECOND
00005 ' 133000          ADD     1,2
00006 ' 044774          STA     2,SUM
                        .ENDC
00007 ' 024771          LDA     1,FIRST
00010 ' 030771          LDA     2,SECOND
00011 ' 133000          ADD     1,2
00012 ' 044770          STA     2,SUM
                        .ENDC
00013 ' 024765          LDA     1,FIRST
00014 ' 030765          LDA     2,SECOND
00015 ' 133000          ADD     1,2
00016 ' 044764          STA     2,SUM
                        .ENDC
                        .END
```

Example 2

```
A=3
.DO      A==3    ;Assemble the following code once
3                ;if the value of A equals 3.  Otherwise,
.ENDC            ;do not assemble the code at all.
```

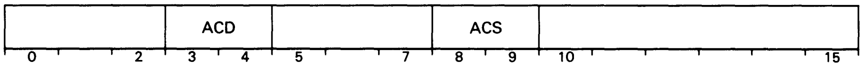
.DTAC Define an Instruction with Two Accumulator Fields

.DTAC *usym*={*inst* | *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an instruction.) *Usym* is an instruction requiring source and destination accumulators. Use the following format:

usym acs acd

The fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
       060401         .DTAC   LDB=060401
00000  030410         LDA     2,.PTR
00001  064601         LDB     2,1            ;Byte addressed
                      .                      ;by AC2 is loaded
                      .                      ;into AC1
                      .
                      .
00010  000022  .PTR:  (.+1)*2
00011  040502         .TXT    "ABCDE"
       041504
       042400
```

See .DUSR for an explanation of attributes common to all .D____ pseudo-ops.

Define a User Symbol for Cross Referencing

.DUSR

.DUSR *usym*={*inst* | *exp*}

Defines *usym* as a user symbol with the value of an instruction or expression.

Exp may be any legal macroassembler expression. It may not be a floating-point constant.

An instruction may be any legal ECLIPSE assembly language instruction. If you supply an instruction, the assembler computes the assembled value of that instruction and assigns it to *usym.* (Refer to "Assignments," Chapter 3, for more information about using instructions in assignments.)

Once defined, *usym* may be used anywhere you would use a single-precision operand.

.DUSR symbols are semi-permanent. They can be redefined only as another .D____ pseudo-op symbol. (If you used the /M switch in the command line, .DUSR can be redefined only by using the .XPNG pseudo-op, which deletes the entire symbol table.) It would be more correct to define *usym* with an assignment statement, if you don't want the symbol to be semi-permanent (see Table7.5).

The above information applies to all symbol-definition pseudo-ops of the form .D____. Each symbol definition pseudo-op has an implied syntax to be associated with the defined symbol.

.DUSR Assignments	Simple Assignments
.DUSR A=10	A=10
.DUSR B=A+20	B=A+20
.DUSR C=LDA 0,0	C=LDA 0,0
.DUSR D=.RDX	D=.RDX

Table 7.5 .DUSR vs. simple assignments

Example

```
          .TITLE     ASSGN
000000    .NREL      1
000010    A=10                  ;A, B, and C are user symbols.
000020    .DUSR      B=20
000030    .DUSR      C=30
000110    A=110                 ;You may redefine A, B, and
000120    B=120                 ;C at any time.
          .END
```

Define an Instruction with Two Accumulators and an Operation Number

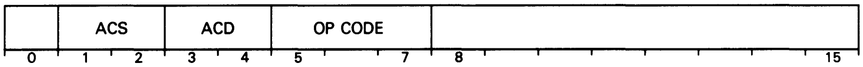
.DXOP

.DXOP *usym*={*inst* / *exp*}

Defines *usym* as a symbol with the value of an instruction or expression. (*Exp* evaluates to the bit pattern of an expression.) *Usym* is an instruction requiring source and destination accumulators and an operation number. Specify these fields with the following format:

usym acs acd op_no.

The fields assemble as shown below.



Once defined, *usym* is an instruction mnemonic.

Example

```
         100030  .DXOP   XOP=100030
         000001  .NREL   1
  00000 ! 130130  XOP     1,2,1
F 00001 ! 100030  XOP     1,2         ;Incorrect  number
                                      ;of arguments
```

.EJECT Begin a New Listing Page.

.EJECT

Directs the assembler to begin a new page in the assembly listing output (after listing the .EJECT source statement).

Example

Source Code

```
          .
          .
          .
          MOV   1,2
          .EJEC         ;Start a new listing page.
          LDA   1,0,1
          .
          .
          .
```

Assembly Listing

```
          Page 1
00000     131000    MOV     1,2
                    .EJEC
          Page 2
00001     024401    LDA     1,0,1
```

End-of-Program Indicator

.END

.END {*expr*}

Terminates your source program. The assembler does not process any source code that follows the .END pseudo-op, so this should be the last statement in your source.

If you assemble several modules at once, only the last one should include the .END statement (use .EOF to end the other modules). If you do not include the .END pseudo-op at the end of the last module on the assembly command line, the assembler supplies one for you (without an argument).

The optional *expr* argument specifies a starting address for execution of your program file. You must supply a start execution address in one of your source modules or the binder returns an error.

The .END pseudo-op can also be used to build a pool of literal values (see .LPOOL).

Example

```
        .TITLE   MOD1
        .NREL    1
START:  SUB      0,0
                 .
                 .
                 .        ;End of module MOD1.  Begin execution
        .END     START    ;of program at location START.
```

.ENDC End of Conditional or Repetitive Assembly

.ENDC {*conditional_label*}

If the syntax is .ENDC, this pseudo-op terminates line for repetitive assembly (lines following .DO), following .IFE, .IFG, .IFL, or .IFN).

If the syntax is ENDC *conditional_label*, this pseudo-op declares a label which marks the end of the next block of conditionally assembled code. If the first block (i.e., the statements between the .DO and the .ENDC) is assembled, then the second block (i.e., the statements between the .ENDC and the label) will not be assembled. If the first block is not assembled, then the second block will be assembled. If the second block follows a .DO block which is not assembled, then the second block will be assembled only once.

Example

```
        .IFN     ALPHA       ;Assemble only if
                             ;ALPHA is not zero
        SUB      0,0
        .ENDC    FAIL        ;End of conditional
                             ;assembly.  If ALPHA
                             ;does not equal 0,
                             ;then do not assemble
                             ;the code between
                             ;this .ENDC statement
                             ;and [FAIL]
        ADD      1,0
        MUL
        SUB      0,0
[FAIL]

        .DO      ALPHA       ;If ALPHA=0, then
                             ;do not assemble the
        ADD      0,0         ;code between .DO
                             ;and the next .ENDC
        .ENDC                ;End the .DO loop
```

Define an External Entry

.ENT

.ENT *usym*$_1$ {...*usym*$_n$}

Declares *usym* as a user symbol that you define in this source module but that you may reference from separately assembled modules.

You must define a user symbol in the module containing the .ENT declaration. This user symbol must be unique among all external symbols you define in the modules you intend to bind together. If the symbol is not unique, the binder issues a message indicating multiply-defined symbols.

To reference *usym* from a separately assembled module, use the .EXTD or .EXTN pseudo-op.

Example

```
          .TITLE    A         ;Module A.
          .ENT      PTR       ;PTR is defined in this module and
                              ;may be referenced by other modules.
          .ZREL
PTR:      TABLE
          .NREL     1
TABLE:    0
          .
          .
          .
          .END
```

```
          .TITLE    B         ;Separately assembled module B.
          .EXTD     PTR       ;PTR is defined in another module.
          .NREL     1
          LDA 0,@PTR          ;Reference to externally defined
                              ;symbol PTR.
          .
          .
          .
          .END
```

.ENTO Define an Overlay Entry

.ENTO *usym*

Associates *usym* with the node number and the number of the overlay in which this module resides. If the module is loaded outside an overlay, the value of *usym* is set to −1. You may reference the overlay from another module by using *usym*.

NOTES: *In the referencing module you must declare* usym *as an .EXTN.*
The value of usym *is assigned at bind time.*

Example

```
.TITLE  MOD
.NREL   1
.ENTO   OVMOD
.
.
.
.END
```

.EOF, .EOT

Explicit End-of-File (Tape)

.EOF
.EOT

These pseudo-ops provide the assembler with an explicit end-of-file or end-of-tape indicator. They indicate the end of one source module, but imply that more source modules follow in the current assembly. Thus, use .EOF or .EOT to terminate each source module in the MASM command line, except the last one, which should end with .END.

If you do not include .EOF or .EOT pseudo-ops in your source modules, the assembler automatically supplies them for you.

Example

```
        .TITLE    A         ;The first piece of source code
        .NREL     1         ;resides in file A.
START:            .
                  .
                  .
        .EOF                ;End of file A but not of source code.

        .TITLE    B
        .NREL     1         ;The second part of the source
                  .         ;code resides in file B.
                  .
                  .
        .END      START     ;End of current assembly.  Start program
                            ;execution at location START.
```

The MASM command line that assembles these two source modules is

```
XEQ MASM  A B ↵
```

.EXTD Define an External Displacement Reference: 8 Bits

.EXTD *usym*$_1$ {...*usym*$_n$}

Declares *usym* as a user symbol that you reference in this source module but that you define in a separately assembled module. You must declare *usym* with an .ENT pseudo-op in the module that defines it. Also, you cannot redefine *usym* within the current assembly.

You may use an .EXTD symbol as a displacement or lower page zero (absolute or ZREL) address in any memory reference (MRI) instruction. When used in this manner, the symbol's value must satisfy one of the following equations:

$0 <= \text{lower page zero address} <= 377_8$
$-200_8 <= \text{displacement} <= 177_8$

Be sure the value of the .EXTD symbol can fit into the corresponding field at bind time.

Example

```
          .TITLE   A     ;Source module A.
          .ENT     D
          .ZREL
D:        TABLE          ;D is a label in page zero relocatable (ZREL)
          .NREL    1     ;memory and, therefore, its value can be
TABLE:                   ;expressed in 8 bits.
                   .
                   .
                   .
          .END


          .TITLE   B     ;Separately assembled module B.
          .EXTD    D     ;D is defined externally and can be
          .NREL    1     ;expressed in an 8-bit field.
          LDA      0,D   ;Load the value at location D into AC0.
                   .
                   .
                   .
          .END
```

Define an External Narrow Reference: 16 Bits.

.EXTN

.EXTN *usym₁* {*...usymₙ*}

Declares *usym* as a user symbol that you reference in this source module but that you define in a separately assembled module. You must declare *usym* with a .ENT or .ENTO pseudo-op in the module that defines it. Also you cannot redefine *usym* within the current assembly.

When you use an .EXTN symbol in your program, make sure the corresponding field is not a memory reference instruction displacement field. Otherwise, the assembler returns an error.

Example

```
.TITLE   A       ;Source module A.
.ENT     N
N=3777           ;N's value may be expressed in 16 bits.
         .
         .
         .
.END
.TITLE   B       ;Separately assembled module B.
.EXTN    N       ;N is defined in another module and
.NREL    1       ;its value may be expressed in 16 bits.
ANDI     N,0     ;Reference to externally defined symbol N.
         .
         .
         .
```

.EXTU Treat Undefined Symbols as External Displacements

.EXTU

Causes the assembler to treat all symbols undefined after pass 1 as if they had appeared in an .EXTD statement.

NOTE: *This pseudo-op is not recommended for use in untested user programs.*

Example

```
                .TITLE   GEEZ
                .EXTU
00000 024000$   LDA      1,IBUS
                    .
                    .
                    .
                .END
```

Force Bind a Module from a Library.

.FORCE

.FORCE

Directs the binder to unconditionally bind this object file from a library into your program file.

Normally, the binder includes an object module from a library only if that module satisfies an external reference appearing in another module. However, if you use .FORCE in a module that resides in a library, that module will be bound whenever the library name occurs in a binder command line.

Example

```
        .TITLE    SQUARE     ;SQUARE is part of a library.  Whenever the
        .FORCE               ;library name appears in a bind command line,
        .ENT      CUBE       ;module SQUARE will be bound into the program
        .NREL     1          ;file.  Thus, the program will have access to
                  .          ;entry CUBE, even if that symbol is not
                  .          ;referenced.
CUBE:             .
        .END
```

.GADD Add a Scalar Value to an External Symbol Value

.GADD *usym, expr*

Generates a storage word whose contents are resolved at bind time. The binder searches for the value of *usym* and, if found, adds it to *expr* to form the contents of the storage word. If the binder does not find the value of *usym,* it generates a binder error. In this case the storage word will contain only the value of *expr*.

Usym must be a user symbol defined in some separately assembled file and must appear in that program in an .ENT, .ENTO, or .COMM statement.

Example

```
                         .TITL   ZEBRA
                         .ENT    ZEB
      000200             .LOC    200
00200 000201  ZEB:       .+1                ;ZEB has the value 200.
                         .END

                         .TITL   STRIPE
                         .EXTN   ZEB
00100 000006  ZEBL:      .GADD   ZEB,5+1    ;Value at ZREL will be 206 plus
                                            ;the ZEB relocation base
                                            ;at bind time
                         .END
```

.GLOC

Initialize Data Fields Relative to an External Symbol.

.GLOC *external-symbol*

The .GLOC pseudo-op defines a block of data whose starting address will equal the value of *external-symbol* at bind time. Define your data immediately after the .GLOC statement. The first occurrence of a .LOC, .NREL, .ZREL, .or .END pseudo-op terminates the data block. Another .GLOC statement also ends the current data block.

To reference the data block from a separately assembled module, include *external-symbol* in an .ENT or .COMM statement within that module. In the current module, you must declare *external-symbol* as externally defined with an .EXTD or .EXTN pseudo-op.

Within the block there can be no label definitions and location counter (period (.) or .LOC) references.

By using .GLOC, you may initialize your data locations at bind time instead of at run time. This can save memory space and reduce your program's execution time.

Example

```
.TITLE    A          ;Source module A.
.COMM     DATA,10    ;Common area DATA contains 10 (octal)
          .          ;words of memory.
          .
          .
.END


.TITLE    B          ;Separately assembled module B.
.EXTN     DATA
.GLOC     DATA       ;This statement directs the binder to
1                    ;initialize common area DATA with
2                    ;the values 1 through 4.
3
4
.END
```

.GOTO Suppress Assembly

.GOTO *usym*

Suppresses the assembly of lines until the scan encounters another *usym* enclosed in square brackets.

Example

```
                          .GOTO   START   ;Start assembly
                                          ;at START.
                          LDA     0,0,2   ;Do not assemble
                          MOVL    0,0     ;these instructions.
                          SUB     1,1
               [START]
00000 021001              LDA     0,1,2   ;Start assembly here.
00001 131100              MOV     1,2
                             .
                             .
                             .
```

Assign Expression Value to Symbol .GREF

.GREF *usym, expr*

Adds the value of *usym* to *expr,* except that a carry out of the low-order 15 bits is not allowed to alter bit 0, which keeps the value of bit 0 of *expr*. Otherwise, functions like the .GADD pseudo-op.

.IFE, .IFG, .IFL, .IFN

Perform Conditional Assembly

.IFE *abs-expr*
.IFG *abs-expr*
.IFL *abs-expr*
.IFN *abs-expr*

These pseudo-ops direct the macroassembler to either assemble or by-pass portions of your module on the basis of *abs-expr* (an absolute expression). The macroassembler assembles the source lines following an .IF pseudo-op if the value of *abs-expr* satisfies the condition defined by that pseudo-op.

The four .IF pseudo-ops define the following conditions:

.IFE *abs-expr* Assemble if *abs-expr* equals 0.

.IFG *abs-expr* Assemble if *abs-expr* is greater than 0.

.IFL *abs-expr* Assemble if *abs-expr* is less than 0.

.IFN *abs-expr* Assemble if *abs-expr* does not equal 0.

You must terminate the conditional assembly lines with the .ENDC pseudo-op.

Thus, the conditional portion of your module has the general form:

```
.IFx abs-expr     ;One of the four conditional pseudo-ops
.                 ;(.IFE, .IFG, .IFL, or .IFN).
.                 ;Assemble these source lines if abs-expr
.                 ;satisfies the .IFx condition.

.ENDC             ;Terminate conditional assembly.
```

When nesting .IFs, be sure the innermost .IF corresponds to the innermost .ENDC, etc.

Note that each .IF condition is a form of a .DO statement. For example, the statement .IFE A is equivalent to .DO A==0. Both direct the macroassembler to assemble the following code once if A equals 0.

In the value field of the assembly listing, the assembler places a 1 if *abs-expr* satisfies the pseudo-op condition and 0 if it does not satisfy the condition.

Example

```
          000000    A=0
          000001    .IFE A  ;A equals 0 so MASM assembles the
          000100    100     ;conditional portion of the module.
000200    200
         .ENDC    ;End of conditional.
```

.LOC Set the Current Location Counter.

.LOC *expr*
.LOC

In the first format, .LOC is an assembler directive. In the second, it is a value symbol.

When used as an assembler directive, the .LOC pseudo-op sets the current location counter to the value and relocation base given by *expr*. The location counter is an assembler variable that holds the address of the next location the assembler will assign.

The argument you supply to .LOC may be any legal assembler expression. If you do not supply an argument, the assembler returns an error.

As an example, if *expr* resolves to an absolute value, then the assembler sets the current location counter to that value and subsequent addresses are not relocatable (they are absolute).

When you use .LOC as a value symbol, it has the value and relocation property of the current location counter, with the exception noted below.

Example

```
      000100       .LOC    100     ; Absolute location 100
00100 000001       1               ; (miscellaneous code)
00101 000002       2
00102 000003       3
      000000       .NREL   0       ; Label A has NREL 0 relocation
00000'000000    A: 0
      000001       .NREL   1       ; Label B has NREL 1 relocation
00000!000000    B: 0
      000010'      .LOC    A+10    ; Set location counter to
00010'000001       1               ; relocation base of A (NREL 0)
00011'000002       2               ; Start assigning at 10th
00012'000003       3               ; address after A
      000025!      .LOC    B+25    ; Set relocation counter to
00025!000004       4               ; relocation base of B (NREL 1)
00026!000005       5               ; Start assigning at 25th
00027!000006       6               ; address after B
```

Exception

If .LOC is pushed to the assembler stack and subsequently used to restore the location counter, e.g.,

```
.PUSH  .LOC
.LOC   .POP
```

then the value is ignored and only the relocation property is changed. This allows you to save the current relocation mode within a macro and restore it correctly without affecting the relative location counter value, which may have been altered within the macro.

.LPOOL Permit Construction of a Literal Pool

.LPOOL

The .LPOOL pseudo-op marks locations where the macroassembler may deposit the values of literal expressions which have not yet been assigned addresses during the current assembler pass. The number of values in a literal pool determines the pool's size. Each value occupies one word. The macroassembler ignores an .LPOOL pseudo-op if no values are currently unassigned.

Since the .LPOOL pseudo-op marks a potential data area, make sure it does not appear in the path of executable code. Usually you place .LPOOL pseudo-ops after branching instructions, returns from subroutines, and other areas typically used for storing data. If you do not include an .LPOOL pseudo-op in a program that uses literals, then the literal pool will be constructed by the .END pseudo-op.

You can set the relocation property of literal pool addresses using the .LOC, .ZREL, and .NREL pseudo-ops.

Example

```
        000001      .NREL  1

00000 ! 020000-     LDA    0,=“A          ; Literal reference

                    .ZREL                 ; Handy place for literal pool

00000 - 000101      .LPOOL                ; Pool for value of "A

                    .EXTN  RIVER

        000100      BANK = 100

        000001      .NREL  1

                    .LPOOL                ; Unused literal pool

00001 ! 032404      LDA     2,@=RIVER     ; External value
00002 ! 024404      LDA     1,=-400/2
00003 ! 020403      LDA     0,=-200       ; Duplicate literal value
00004 ! 002403      JMP     @=BANK

00005 ! 000000$     .LPOOL                ; 3-word pool for 4 values
        177600
        000100
```

.MACRO Define a Macro.

.MACRO *usym*
macro-definition-string
%

The .MACRO pseudo-op defines *usym* as the name of *macro-definition-string* (zero or more source lines that you use repeatedly in your module). After defining the macro, you simply insert *usym* in your source module, and the assembler substitutes *macro-definition-string.*

When defining a macro, you must terminate the macro definition string with the percent character (%).

Chapter 5 provides a complete discussion of the assembler's macro facility.

Example

```
000001           .NREL 1
                 .MACRO TEST ;Define macro TEST.
                 ↑1          ;Macro definition consists of
                 ↑2          ;3 data statements that get
                 ↑3          ;values from the first 3
           %                 ;arguments passed to TEST.


                 TEST 4,5,6  ;Call TEST with 3 arguments.
000004           4           ;Macro definition consists
000005           5           ;of data statements that get
000006           6           ;values from first 3 arguments.
```

.MCALL

Indicate Macro Usage.

The .MCALL pseudo-op is a value symbol with the value 1 if the macro containing it was called previously on this assembly pass, and the value 0 if this is the first call to that macro on the current pass. Thus, you generally use .MCALL when you want the assembler to use one macro expansion the first time a macro is called and a different expansion for subsequent calls to that macro.

If you use .MCALL outside a macro, its value is −1.

Example

```
        .MACRO     Z
        .DO        .MCALL< >0   ;If this is not the first
                                ;macro call assemble
        JSR        @.X          ;this code
        .ENDC
        .DO       .MCALL= =0    ;If this is the first macro
                                ;call then generate
                                ;subroutine X
        .PUSH     .             ;Save the location counter
        .ZREL
.X:     X
        .LOC      .POP          ;Restore location counter
        JSR       X             ;Call subroutine X
        JMP       XEND          ;Jump to end
X:      .                       ;This would be the code
                                ;for X
        .
        .
        JMP       0,3           ;Return to main routine
XEND:
        .ENDC                    ;End the .DO loop
%                                ;End the macro
```

.NOCON

Inhibit or Re-enable the Listing of Conditional Lines.

.NOCON *abs-expr*

The .NOCON pseudo-op either inhibits or permits the listing of the conditional portions of the source module that do not meet the conditions given for assembly. That is, .NOCON either inhibits or enables the listing of false conditionals. If the value of *abs-expr* does not equal zero, the assembler inhibits listing; if the value of *abs-expr* equals zero, the assembler lists all conditionals. *Abs-expr* must be an absolute expression.

.NOCON does not affect the conditional portions of the source program assembled. Again, this pseudo-op influences only the listing of conditionals that are false (those .DOs and .IFs that are not assembled).

By default, the macroassembler lists all conditionals.

You may override the .NOMAC pseudo-op at assembly time by using the /O function switch on the MASM command line.

You may use .NOCON as a value symbol in your module. The value of .NOCON equals the value of the last *abs-expr* you passed to .NOCON.

The default value for .NOCON is 0.

Example

Source Code

```
A=3
.IFE A      ;False.  MASM lists false
10          ;conditionals, by default.
20
30
.ENDC

.NOCON 1    ;Inhibit listing of false conditionals.

.IFE A      ;False.  MASM will not list this portion
40          ;of code.
50
60
.ENDC

.IFN A      ;True.  MASM lists the assembled code
70          ;regardless of the .NOCON setting.
100
110
.ENDC
```

Assembly Listing

```
000003    A=3
000000    .IFE A   ;False.  MASM lists false
          10       ;conditionals, by default.
          20
          30
          .ENDC

000001    .NOCON 1 ;Inhibit listing of false conditionals.

          .ENDC

000001    .IFN A   ;True.  MASM lists the assembled code
000070    70       ;regardless of the .NOCON setting.
000100    100
000110    110
          .ENDC
```

.NOLOC Inhibit or Re-enable the Listing of Source Lines without Location Fields.

.NOLOC *abs-expr*

The .NOLOC pseudo-op directs the assembler to either inhibit or enable the listing of source lines that lack a location field. That is, .NOLOC controls the listing of source lines that would not have a location listed in the output. If *abs-expr* evaluates to a nonzero value, the assembler inhibits listing; if *abs-expr* equals zero, listing occurs. *Abs-expr* must be an absolute expression.

By default, the assembler lists all source lines, whether they have location fields or not.

You may override the .NOLOC pseudo-op at assembly time by using the /O function switch on the MASM command line.

You may use .NOLOC as a value symbol, in which case it has the value of the last *abs-expr* you passed to .NOLOC.

The default value for .NOLOC is 0.

Example

Source Code

```
.NOLOC   0
.NREL    1
.TXT     "ABCDEF"     ;This prints.
.NOLOC   1             ;This does not print.
.TXT     "GHIJ"       ;This line prints,
                       ;but the second
                       ;  line does not.
LDA      0,TEMP        ;This prints
.LOC     .+12          ;This will not print.
STA      0,TEMP        ;This prints.
.END                   ;This will not print.
```

Assembly Listing

```
        000000  .NOLOC    0
00000 ! 000001  .NREL     1
00001 ! 040502  .TXT      "ABCDEF"  ;This prints.
        041504
        042506
        000000
00005 ! 043510   .TXT      "GHIJ"   ;This line prints,

00010 ! 020411   LDA       0,TEMP   ;This prints.
00023 ! 040411   STA       0,TEMP   ;This prints.
```

.NOMAC Inhibit or Re-Enable the Listing of Macro Expansions.

.NOMAC *abs-expr*

The .NOMAC pseudo-op either inhibits or permits the listing of macro expansions. If *abs-expr* (an absolute expression) does not equal zero, the assembler does not include macro expansions in the listing; if *abs-expr* equals zero, listing occurs.

By default, the assembler includes all macro expansions in the listing output.

You may override the .NOMAC pseudo-op at assembly time by using the /O function switch on the MASM command line.

You may use .NOMAC as a value symbol, in which case it has the value of the last *abs-expr* you passed to .NOMAC.

The default value for .NOMAC is 0.

Example

```
                .MACRO MAC ;Define macro MAC.
                5
                6
          %


                MAC       ;Call macro MAC with default .NOMAC.
00000  000005   5
00001  000006   6


       000001   .NOMAC 1  ;Inhibit listing of macro expansions.

                MAC       ;MASM does not list MAC's expansion.

00004  000100   100       ;A line not part of macro.
```

Specify Normal Relocation

.NREL

.NREL *{expr}*

The .NREL pseudo-op causes subsequent source statements (up to the next .NREL, .LOC or .ZREL pseudo-op) to be assembled using either the pure code relocation counter (.NREL 1) or the impure code relocation counter (.NREL 0). The argument *expr* is optional; the default value is zero.

Use .NREL 1 for sections of the program that can be write-protected (typically, instructions). Use NREL 0 for sections of the program that will be modified during program execution (typically, data areas). Code in the .NREL 1 section may be shared among processes. Code in the .NREL 0 section may not be shared. Any portions of the program to be used in overlays must be in the pure area.

Example

```
        000000    .NREL    0       ;Impure area (note the relocation base
                                   ;indicator of subsequent address)
00000 ' 000000    0
00001 ' 000001    1
        000001    .NREL    1       ;Pure area (note the relocation base
00000 ' 020000-   LDA      0,=10   ;indicator of subsequent address)

        000000    .NREL    0       ;Impure area again
00002 ' 000005    5
00003 ' 000006    6
        000001    .NREL    1       ;Pure area again
00001 ! 024001-   LDA      1,=120
```

.OB Name an Object File.

.OB *filename*

The .OB pseudo-op directs the assembler to name the object file *filename*. The assembler appends the object file extension .OB onto *filename* unless that name already ends in .OB.

If more than one .OB pseudo-op appears in the source, the assembler returns an error.

If you include the /N switch in the MASM command line, directing the assembler not to produce an object file, the assembler ignores the .OB pseudo-op.

If you specify the /B= switch on the MASM command line, the assembler overrides the .OB pseudo-op, and the object file receives the name following the /B= switch.

In sum, the assembler uses the hierarchy given in Table 7.6 to name object files:

Priority	Object Filename	Description
1 (highest)	/B=*filename*	Switch on the MASM command line
2	.OB *filename*	Pseudo-op in the source module
3 (lowest)	Default name	The name of the first source module on the MASM command line that doesn't have the /S argument switch.

Table 7.6 Priority of object filenames

One of the primary uses of the .OB pseudo-op is in conditional code assembly. You may direct the assembler to assign a name to the object file according to the evaluation of some expression (see the example).

Example

```
.IFE    VAR     ;If the value of VAR equals 0, MASM
.OB     SYS1    ;names the object file SYS1.OB.
.ENDC
.IFN    VAR
                ;Otherwise, MASM names the
.OB     SYS2    ;file SYS2.OB.
.ENDC
```

.PASS

Number of Assembly Pass.

The macroassembler scans your source code twice during the assembly process. Each scan is called a pass.

The .PASS pseudo-op is a value symbol that returns the current assembly pass number. .PASS equals 0 on assembly pass one and 1 on pass two.

Example

This example defines parameters A, B, and C for later use in the assembly. Since the values of A, B, and C remain constant, MASM need not assemble them on pass two. This use of .PASS is similar to that of the /S argument switch (see Chapter 6).

```
.IFE     .PASS
A=0                ;MASM assembles these statements
B=1                ;on pass one only (i.e., when
C=2                ;.PASS equals 0).
.ENDC
```

.POP Pop the Value and Relocation Property of the Last Expression Pushed onto the Stack.

The permanent symbol .POP has the value and relocation property of the most recent expression you placed on the assembler stack (using .PUSH). When you use .POP, the assembler removes this entry from the stack.

If the assembler stack contains no values, then .POP has the value 0 and the absolute relocation property. In addition, the assembler returns an error for that source statement.

Example

```
      000025  A=25     ;Define A.
00000 000025  A        ;Assemble A's present value.

      000025  .PUSH A  ;Push A's value onto the assembler's
      000015  A=15     ;stack.  Assign A a new value
00001 000015  A        ;and assemble that value.

      000025  A=.POP   ;Assign A the value from the top
00002 000025  A        ;of the stack and assemble A's
                       ;new value.
```

See the .PUSH description for another example.

Push a Value and its Relocation Property onto the Stack. .PUSH

.PUSH *expr*

The .PUSH pseudo-op allows you to save on the assembler stack the value and relocation property of any valid assembler expression. You may continue to push expressions onto the stack until the stack is exhausted.

The assembler stack is the push-down type. That is, the last expression you place on the stack is the first one to be removed. Use the .POP pseudo-op to access information on the stack.

Example

```
      000010   .RDX 8      ;Input radix is 8 (octal);
00000 000010   (.RDX)      ;Assemble the current
                           ;input radix value.
      000010    .PUSH .RDX ;Save the input radix on the
                           ;stack.
00001 000012    .RDX  10   ;Set the input radix
                           ;to 10 (i.e., decimal)
      000012    (.RDX)     ;Assemble the current input
                           ;radix value.
      000010    .RDX  .POP ;Set the input radix to the
                           ;value on top of the stack
00002 000010    (.RDX)     ;(in this case, 8).  Assemble
                           ;the current input radix value.
```

See the .POP description for another example.

.RDX Set Radix for Numeric Input Conversion.

.RDX *abs-expr*

The .RDX pseudo-op defines the radix (base) that the macroassembler uses to interpret the numeric expressions in your source module. For example, if you specify an input radix of 16_{10}, the assembler interprets all numeric expressions in your module in hexadecimal.

The assembler always interprets *abs-expr* in decimal. This argument must be an absolute expression and its value must be in the following range:

$2 <= \textit{abs-expr} <= 20_{10}$

If you do not specify an input radix in your module, the default value is 8_{10} (i.e., octal).

If you specify a radix greater than 10, you must use letters to represent digits greater than 10 but less than the specified radix. For example, if you declare an input radix of 16_{10} (hexadecimal), use the digits 0 through 9 to represent the quantities 0 through 9_{10}, and use the letters A through F to represent the values 10 through 15_{10}.

If the input radix is greater than 10, your numeric constant might start with letters. In these cases, you must place a 0 before the initial letter of the numeric constant to distinguish it from a symbol. For example, if you specify an input radix of 16_{10} (.RDX 16), then you should express the value for decimal 15 as 0F, not simply F.

Regardless of the input radix, the assembler interprets any number containing a decimal point as decimal. For example, the numeric expression 12. always equals 12_{10}, regardless of the input radix. This feature allows you to combine decimal numbers in expressions with numbers of other radixes (e.g., 0F+12.−3A2).

Note that the input and output radixes are entirely distinct. Setting the input radix does not affect the output radix (set with .RDXO).

You may use .RDX as a value symbol. In this case, .RDX has the value of the current input radix.

Example

In this example, the output radix is 8 (i.e., octal). You may alter this setting with the .RDXO pseudo-op.

```
01          000010  .RDX 8    ;Input radix is 8.
02   00000  000123  123       ;MASM assembles 123 in octal.
03
04          000012  .RDX 10   ;Set input radix to 10.
05   00001  000173  123       ;MASM assembles 123 in decimal.
06
07          000020  .RDX 16   ;Set input radix to 16.
08   00002  000443  123       ;MASM assembles 123 in hexadecimal.
09   00003  000017  0F        ;Note the leading zero.
10   00004  000173  123.      ;MASM assembles 123. in decimal even
11                            ;though the input radix is 16.
12
13   00005  000020  (.RDX)    ;Assembles the current input radix
14                            ;value.
```

.RDXO Set the Radix for Numeric Output Conversion.

.RDXO *abs-expr*

The .RDXO pseudo-op defines the radix (base) that the assembler uses to represent numeric expressions in the assembly listing. For example, if you specify an output radix of 10_{10}, the assembler presents all locations and values in decimal, regardless of the input radix.

The assembler always interprets *abs-expr* as decimal. This argument must be an absolute expression, and its value must be in the following range:

$$8_{10} <= \textit{abs-expr} <= 20_{10}$$

If you do not specify an output radix in your module, the assembler uses 8_{10} (i.e., octal).

You may use .RDXO as a value symbol, in which case it equals the current output radix. The assembler always expresses the current output radix as 10.

Example

```
000010   .RDX 8     ;Input radix is 8.
000010   .RDXO 8    ;Output radix is 8.
000077   77
000022   22
000045   45
000012   .RDX 10    ;Input radix is 10.
 00010   .RDXO 10   ;Output radix is 10.
 00077   77
 00022   22
 00045   45
 00016   .RDX 16    ;Input radix is 16.
  0010   .RDXO 16   ;Output radix is 16.
  0077   77
  0022   22
  0045   45
000010   .RDXO 8    ;Output radix is 8.
000167   77         ;Input radix is 16.
000042   22
000105   45
 00010   .RDXO 10   ;Output radix is 10.
 00119   77         ;Input radix is 16.
 00034   22
 00069   45
 00010   (.RDXO)    ;Assemble the value of the current output
                    ;radix. MASM always represents this as 10,
                    ;regardless of the current output base.
```

.REV Set the Revision Level

.REV *maj_rev min_rev*

Identifies a program's revision level. You enter the major and minor revision levels as numbers in the current radix or as legal expressions. Revision levels are carried into the object file and then into the program file. Both the major and minor revision levels have a numeric range of 0 through 255_{10}. This pseudo op may appear anywhere in the source module.

If MASM finds no .REV then it puts the value −1 in the object module. The −1 would be interpreted as 255.255. The binder, however, recognizes −1 as an undefined revision number and doesn't use it.

Use the CLI REV command to obtain revision information of a program file.

```
.TITLE  QUIZ
.REV    12,1     ;Revision data is in octal (default
                 ;input radix)
```

.TITLE

Entitle an Object Module.

.TITLE *usym*

The .TITLE pseudo-op provides a name for your object module by placing *usym* in the module's title block.

The title you assign in the module appears at the top of each page in the assembly listing. *Usym* need not be unique from other symbols except for macro names.

If you do not include .TITLE in your source module, the assembler supplies the default name .MAIN.

Note that the name you assign in the .TITLE statement has no relation to the name of the object file (see the .OB pseudo-op).

Example

```
.TITL   MODULE 1
        .
        .
        .
```

.TOP Value and Relocation of Top Stack Word

.TOP

.TOP has the value and relocation property of the most recent expression pushed onto the variable stack. .TOP differs from .POP in that it does not pop the last pushed expression from the stack. If you have not pushed any expressions before you issue a .TOP pseudo-op, the macroassembler returns 0 (absolute relocation) and flags the .TOP line with the overflow error flag (O).

Example

```
000010    .RDX    8
            .
            .
            .
000010    .PUSH   .RDX
            .
            .
            .
000020    .RDX    16
            .
            .
            .
000010    .RDX    .TOP
            .
            .
            .
000012    .RDX    10
            .
            .
            .
000010    .RDX    .POP
```

Reserve a Number of Tasks. .TSK

.TSK *abs-expr*

The .TSK psuedo-op specifies the maximum number of tasks that your program can initiate at execution time. The argument you pass to .TSK must be an absolute expression. The range of values this argument can take depends on the parameters established at SYSGEN.

If several object files in the same binder command line contain .TSK declarations, the binder uses the largest.

Example

```
.TITLE    MOD
.TSK      5       ;This program may initiate
                  ;up to 5 tasks at runtime.
```

.TXT Store a Text String

.TXT *%string%*

This pseudo-op directs the assembler to store the binary equivalent of an ASCII text string in consecutive memory words. In the above syntax description, *string* is an ASCII text string and % represents a character that you use to delimit the string. The delimiter may be any character except

- a character that appears in text string *string* or
- New Line, Form Feed, Carriage Return, Tab, Space, Null, Colon, Semicolon, Dollar Sign and Backslash.

You may use New Line, Form Feed, and Carriage Return to continue a string from line to line, but the assembler will not store these characters as part of the text *string*.

When the assembler encounters .TXT, it interprets the first character after the break as the string delimiter. The assembler then stores the following characters in pairs in consecutive memory words until it encounters the delimiting character again. That is, the assembler generates one 16-bit storage word for every two characters in *string*.

The assembler allocates an 8-bit byte for each character (i.e., two characters per 16-bit word). By default, the assembler packs the characters of your string from left to right within memory storage words. You may alter this packing mode with the .TXTM pseudo-op.

If your string contains an odd number of characters, the assembler pairs a null (all zero) byte with the last character of the string. If your string contains an even number of characters, the assembler stores a null word (2 null bytes) immediately after the string. You may suppress this null storage word by using the .TXTN pseudo-op.

If you wish to use an expression to specify a character in a text string, enclose the expression in angle brackets (<>). The assembler evaluates this expression, truncates the result to 8 bits, and stores it in the appropriate byte of memory. You may not include relational operators within the expression.

By using angle brackets, you may store the ASCII codes for characters that you could not otherwise include in your text string. For example, you may not include a New Line character in your text string. However, you may include the ASCII code for New Line in a text string if your enclose that value in brackets, as follows:

```
.TXT "A<12>"
```

This statement directs the assembler to generate a storage word that contains the ASCII codes for A (101_8) and New Line (12_8). By default, the macroassembler stores a null (all zero) word in the following location.

.TXT sets the leftmost (parity) bit of each byte to 0, except when the byte is the result of an expression. See .TXTE for information about changing the parity bit.

Example

For examples, see .TXTE, .TXTM, and .TXTN.

.TXTE, .TXTF, .TXTO

Specify a Text String

.TXTE **string**
.TXTF **string**
.TXTO **string**

These pseudo-ops cause the assembler to scan the input following the character * up to the next occurrence of the character * in string mode. The character * may be any character not used within the string except Null, any new-line character, or any space character. The * delimits but is not part of the string. You may use a new-line character to continue the string from line to line or from page to page, but this character is not stored as part of the text string.

Every two bytes generate a single storage word containing binary codes for the ASCII bytes. Storage of a character of a string requires seven bits of an eight-bit byte. The leftmost (parity) bit may be set to 1, even parity, or odd parity as follows:

.TXTF	sets leftmost bit to 1 unconditionally.
.TXTE	sets leftmost bit for even parity on byte.
.TXTO	sets leftmost bit for odd parity on byte.

The packing mode can be altered using the .TXTM pseudo-op. If an even number of bytes is assembled, the null word following these packed bytes can be suppressed by the .TXTN pseudo-op.

Within the string, you can use angle brackets (<>) to delimit an arithmetic expression. The macroassembler evaluates the expression and masks it to eight bits. (This means you can set the parity bit for a .TXT string here. The macroassembler masks out the parity bit for .TXTE, .TXTO, and .TXTF strings later.) Angle brackets are the only means, for example, to store a new-line character as part of the text string (see example).

NOTE: *You cannot use relational operators within the expression.*

In the default condition, bytes are packed left to right, and a word of null bytes is generated at the end of the string.

Example

```
.TXT     #AB <12> CDE#   ;All the examples
                         ;will assemble.

.TXTE    $AB CD$


.TXTF    @AB   CD@


.TXTO    EAB CDE
```

.TXTM Change Text Byte Packing

.TXTM *abs-expr*

The .TXTM pseudo-op directs the assembler to pack bytes either left to right or right to left within memory words when it encounters a .TXT pseudo-op. If *abs-expr* evaluates to zero, the assembler packs the bytes right to left; if *abs-expr* does not equal zero, the assembler packs bytes left to right. The argument you supply to .TXTM must be an absolute expression.

If you do not use the .TXTM pseudo-op in your module, MASM packs bytes from left to right, by default.

You may use .TXTM as a value symbol. In this case, .TXTM represents the value of the last *abs-expr* you supplied to it.

The default value for .TXTM is 1.

Example

```
00000 000001  (.TXTM)          ;Default value of .TXTM = 1

00001 040502  .TXT ''ABCDE''   ;Bytes packed left-to-right
      041504
      042400

      000000   .TXTM  0        ;Pack bytes right-to-left
                               ; within  each word
00004 000000   (.TXTM)

00005 041101   .TXT ''ABCDE''  ;Bytes packed right-to-left
      042103
      000105
```

.TXTN

Determine Text String Termination.

.TXTN *abs-expr*

The .TXTN pseudo-op specifies whether or not the assembler will place a null word at the end of a .TXT text string that contains an even number of characters.

If *abs-expr* evaluates to zero, all text strings containing an even number of characters terminate with a 16-bit null word (all zeros). If *abs-expr* does not equal zero, the assembler does not place a null word after the last two characters in your string. The argument you supply to .TXTN must be an absolute expression.

If you do not use .TXTN in your module, the assembler terminates even length text strings with a null word, by default. When a string contains an odd number of characters, the assembler stores a null byte with the last character, in all cases.

You may use .TXTN as a value symbol. In this case, .TXTN represents the value of the last *abs-expr* you passed to it.

The default value for .TXTN is 0.

Example

```
      000000  .TXTN  0
00000 030462  .TXT   /1234/    ;End string with a word of zeros
      031464
      000000

00003 000000  (.TXTN)
      000001  .TXTN   1
00004 030462  .TXT    /1234/   ;Do not add zeros to this string
      031464
00006 000001   (.TXTN)
```

.XPNG Remove All Non-Permanent Symbol and Macro Definitions

.XPNG

Deletes the symbol table and macro definition table. This pseudo-op is primarily used in the following sequence:

1. You write a source file containing .XPNG followed by definitions of any symbols or macros.
2. You assemble the source file using the /S or /S= function switch in the MASM command line. This causes the assembler to stop the assembly after pass 1 and save the symbols in a new symbol table file.
3. Then you can use the new symbol file containing the symbols defined in step 2.

See "Building a Permanent Symbol Table," in Chapter 6 for further information.

Example

```
          .TITLE   XP
          .XPNG
020000    .DMRA    LDA=20000
040000    .DMRA    STA=40000
          .END
```

The CLI command form is

XEQ MASM/S XP ↵

The macroassembler symbol table now contains LDA and STA.

Specify Lower Page Zero Relocation.

.ZREL

.ZREL

The .ZREL pseudo-op directs the assembler to assign relocatable addresses in lower page zero to subsequent source lines in your module. Lower page zero relocatable (ZREL) memory extends from location 50_8 to 377_8. Thus, you may express any ZREL location in a displacement field of 8 or more bits (in any memory reference instruction).

The words following the .ZREL pseudo-op receive relocatable addresses starting with zero. If you change location counters during an assembly and later return, the assembler continues assigning ZREL addresses at the point where it left off.

At bind time, all ZREL code is contiguous in memory.

Example

```
                  .TITLE Z
                  .ZREL      ;Place the following code in lower
00000-000000      0          ;page zero relocatable (ZREL) memory
00001-000000      0          ;Note the address relocation base
      000001      .NREL 1    ;Pure code
00000!000001      1
00001!000001      1
                  .ZREL      ;Lower page zero
00002-000002      2          ;MASM continues assigning
00003-000002      2          ;ZREL addresses at the point where it left off
```

The Binder

2

Review of Binder Concepts

The binder is a utility program that forms executable programs from assembled or compiled *object modules.* This chapter reviews the steps in developing an executable program and the binder's role in that process. It lists the inputs to the binder and the various outputs, including the executable program files, error files, and load maps. It also sketches the algorithm employed by the binder to construct program files.

Program Development

The five steps in writing and running a program are as follows:

1. Writing and editing the program. Normally, you enter the program from a console using one of Data General's text editors. The program in ASCII format is called a *source module* or *source program* and, when stored on disk, a *source file.* By convention, source filenames usually end with an extension of two or more letters to indicate the language in which they are written.
2. Assembling or compiling the program. The macroassembler or compiler produces an intermediate binary file called an *object module,* stored in a new file, called the *object file.* Object filenames end with the .OB extension. You must eliminate *assembly errors or compiling errors* from the object module before proceeding to the next step, which transforms the object file(s) into an executable program.
3. Binding the program. The binder transforms one or more object modules into an executable program. Output from the binder is called the *program file,* and its name ends with a .PR extension.
4. Executing the program. You generally execute the program by typing

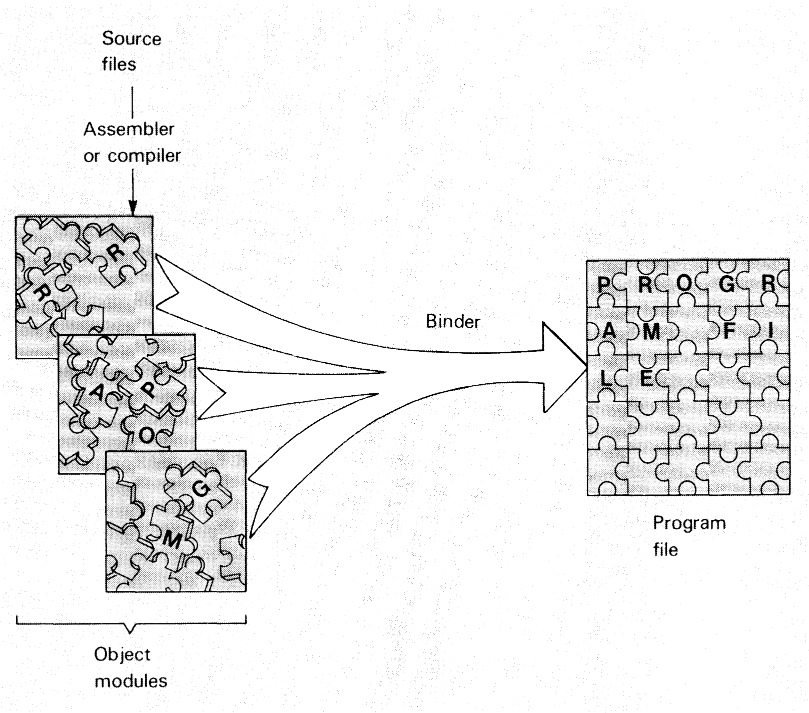
   ```
   XEQ program-filename ↲
   ```

   at the console. When the program performs as you intend, this is the last step in the procedure.
5. Debugging the program. Two general types of errors may occur when you run your program: *runtime errors*, which the system software detects, and errors in the logic of the program that do not permit the program to perform as you intend. To find the cause of either type of error you can use the MP/AOS FLIT debugger (see *MP/AOS Debugger and Performance Monitoring Utilities,* DGC No. 069-400205) and the source listings.

Binder Role

The MP/AOS binder is a utility program that forms executable programs from object modules. It accepts one or more object modules as input, resolves inter-module references, and determines the memory allocation for the program. To do this, the binder may also load object modules from one or more library files and control overlay organization. In addition to forming executable programs to run under the MP/AOS operating system, the binder can also generate new MP/AOS systems and error message text (ERMES) files.

The role of the binder is illustrated in Figure 1.1.



DG-08722

Figure 1.1 Binder in program development

Executable Programs

Object Modules

Object modules (sometimes called OBs) are the output of assemblers and compilers. These files contain the binary translation of the source modules' input to assembler or compiler. The object module also contains information that the binder needs to assign addresses in the logical address space to each word in the program.

The assembler or compiler assigns a *relocation base* and *offset* to code contained in the object module. (Part 1, Chapter 2 explains how the MP/AOS assembler performs this function.) The binder, in turn, assigns an absolute address to each relocation base and uses each word's offset from a base to calculate that word's address. To do this, the binder employs one of the relocation formulas described below (see "Binder Operations"). Chapter 4 describes how relocation bases and offsets are represented within object modules.

Object modules are made up of object blocks. Each OB block type has a different format, and all language processors use the same OB format conventions. Thus, to the binder, an object file produced by the assembler is indistinguishable from one produced by a compiler. Object blocks are described in detail in Chapter 4, but you do not need to be aware of object block formats unless you are writing a system utility, such as an assembler, compiler, or binder.

If you are writing in assembly language, you may embed instructions to the binder in the text of the assembly language source program.

Library Files

To avoid rewriting and reassembling or recompiling subroutines and other commonly used modules, you can store object modules in a *library file.* A library file is a file that contains object modules and indexing information that enables the binder to find quickly any module you may require.

The library file editor (LED) enables you to keep your library files current. The symbol cross-reference analysis (SCAN) program enables you to inspect quickly the contents of modules in library files. This program allows you to see in which module each symbol is used, and if any symbols are defined in more than one module. See Part 3 for a description of the SCAN program and the library file editor.

Logical Address Space

As explained in *MP/AOS System Programmer's Reference*, DGC No. 093-400051, MP/AOS programs run in a *process space* of 32Kwords. The binder assigns to each relocatable address supplied by the assembler or compiler an *absolute address* in the program's *logical address space.* The program's logical address space is divided into three main areas:

- *Lower page zero* consists of addresses in the range 0-377.
- The *pure area* contains portions of the program, usually instructions, that are not modified during execution. The range of addresses in this area differs from program to program.
- The *impure area* contains portions of the program, usually data, that may be altered during execution. The range of addresses in this area differs from program to program.

Chapter 3 explains how these areas are shown on the listing.

The Program

To make an executable program out of one or more object modules, the binder constructs the program exactly as it will be represented in memory. To do this, it resolves external symbol references, and assigns absolute addresses to the relocatable addresses supplied by the assembler or compiler.

Names used in more than one module are *global symbols* (those used in only one module are *local symbols).* In addition to using global symbols, modules may communicate by sharing *common areas* of the process space. The binder manages references to both named and unnamed common areas.

Executable programs produced by the binder are of two general types -- those that run under the MP/AOS operating system, and system programs that comprise the operating system. In most cases, however, system files can be more easily created with the SYSGEN utility described in *MP/AOS System Generation and Related Utilities,* DGC No. 069-400206.

Programs that run under the MP/AOS system are program (.PR) files with or without corresponding overlay (.OL) files. Chapter 2 describes overlay files and their relation to program files.

Binder Output Files

In addition to the .PR and .OL files, the binder produces three types of output — a program symbol table file, an error file and a listing file.

The symbol table (.ST) file contains a non-printable list of symbols used in the program, along with their values. This file is used by the FLIT debugger, PROFILE, and other programs.

The error file lists any errors detected by the binder. Chapter 3 lists and describes these errors.

The listing file contains a *load map* that summarizes the apportionment of your program to the available logical address space. You may also cause the binder to include in this file a list of object modules from a library file and a summary of their allocation and/or get a listing of global symbols.

Binder Operations

The following description of binder operations is intended to help you understand what the binder does internally and why it takes time to bind a program. You do not need this information to use the binder, nor do you need the partition numbers listed in Table 1.2 unless you are writing a code generator.

To produce a program file, the binder scans (takes a "pass" over) the object files twice. Table 1.1 summarizes the actions of the binder during each pass.

Pass	Action
1	Scans the object modules listed in the command line and gauges memory requirements of each. Organizes portions of object module into partitions. Builds symbol table.
Between passes	Builds a map of ZREL, pure and impure program areas. Assigns final relocation values to all values and partitions.
2	Relocates the contents of each partition relative to each partition's relocation base. Finishes building load map. Resolves the contents of the data elements. Builds the .PR, .OL, and .ST files.

Table 1.1 Binder passes

First Pass

During the first pass, the binder scans each object module supplied as input and determines the number and kind of relocation bases in each. A *partition number* corresponds to each relocation base as illustrated in Table 1.2. The numbers 0 to 7 are assigned by the language processor (assembler or compiler) to pre-defined partitions — absolute, relocatable lower page zero (ZREL), normal relocatable pure and impure (NREL). Each external symbol is assigned a partition number in the range 8 to 4095 in the order in which the binder encounters them.

During this pass the binder also builds a temporary table of relocation bases for each module and gathers the information needed to construct the symbol table (the .ST file).

Base Number	Description
0	Absolute (no relocation)
1	Lower page zero (ZREL) locations 0-47
2,3	Reserved for system use
4	Impure code (NREL 0)
5	Pure data
6	Impure data
7	Pure code (NREL 1)
8 to 4095	External relocation

Table 1.2 Relocation bases

Between Passes

Between the first and second passes, the binder starts to build the *load map,* which is a table showing the relocation bases assigned to the partitions in the object modules.

At this point, the binder also begins to relocate the symbols and data words assosciated with each partition. To do this, it positions the data and symbol elements within each partition.

Second Pass

During the second pass the binder resolves the value of each data word and constructs the final format of the ouput files you have specified, including the program file (.PR) and the load map.

Libraries and Overlays

Libraries provide a mechanism for storing object modules, helping you keep track of them and enabling the binder to find the ones you need. This chapter contains a summary of information about library use under MP/AOS.

Overlays are sections of a program that take turns in memory and reside on disk when not being used. This chapter contains an overview of the use of overlays under MP/AOS, and a preview of the instructions for using overlays presented in Chapter 3.

Libraries

Libraries are files that contain object modules and indexing information to enable the binder to find quickly the modules it needs. As you read the description that follows, remember the distinction between object files and object modules:

- *Object files* are the representation of object modules on disk or other secondary memory. They can be manipulated, using the MP/AOS file system.
- *Object modules* do not invoke the file system; a module is referenced by its title (which may be different from its filename). The file system of the operating system does not recognize titles, which are an internal identifier used by the binder.

Library Structure

Chapter 4 contains a detailed description of the internal structure of libraries. Briefly, libraries comprise a library start block (described below), object modules and a library end block, in that order. The library start block contains the summary information about each module in the library; this information allows the binder to determine which modules it needs and to find them. The body of the library consists of concatenated, unmodified object modules. The library end block is a simple marker that the binder recognizes as the end of the library; it contains no other information.

Library Start Block

The library start block contains three essential kinds of information:

- force-bind flags (see below);
- indexing information to allow the binder to find the modules it needs (see Chapter 4).
- the names of symbols defined and referenced in each module (see Chapter 4);

Force-Bind Flags

A *force-bind flag* is a bit that you can set to force the binder to bind a module from a library file into a program, even if the module contains no symbol referenced by the program.

Each object module in a library has two force-bind flags associated with it. One flag is in the object module and is used only when the module is incorporated into a new library. The other flag is in the library start block; this flag determines whether the module is to be automatically bound.

In assembly language programs, you can set the force-bind flag in the object module by using the .FORCE pseudo-op. (This is the only means provided by any existing language translator to set this flag.) You can set the force-bind flag in the library start block for any object module (not just assembly language modules), using the library file editor.

Part 3 describes in detail the mechanics of the force-bind flag.

Binding Libraries

The Binder processes each file (.OB or .LB) in the command line in the order of its appearance. The MP/AOS system library (OSL) is bound automatically and last, by default. When a library is bound, any library module that resolves an unresolved symbol or has the force-bind flag set is included in the program. The binder scans each library once per pass from beginning to end and binds the first module that declares one of your program's unresolved symbols as an entry symbol.

Because the binder scans each library only once per pass from beginning to end, the order of modules within the library is important. Chapter 3 explains the programming implications of the order of modules in libraries. You can use the library file editor to construct new libraries with the object modules arranged as you need them.

Refer to Chapter 3 for an example of how to specify libraries in the binder command line.

Library-Related Utilities

You build libraries using the library file editor (LED) and you can analyze library contents using the Symbol Cross-Reference Analyzer (SCAN). The symbol analyzer enables you to identify multiply-defined symbols and symbols without entry declarations. (See Part 3 for a description of LED and SCAN.)

Overlays

Under MP/AOS, running programs (.PR files) occupy a process space made up of a ZREL, a pure area, and an impure area and limited in size to 32 Kwords. Overlays are portions of a program that occupy the process space for only a portion of the time that the program is running.

The area of the process space that an overlay uses is called an *overlay node.* Programs may have several overlay nodes, and each node may accommodate several overlays, one at a time. When not occupying an overlay node, overlays reside on disk in an *overlay file.*

The binder creates program and overlay files from object files according to instructions that you place in the binder command line (see Chapter 3). In order to be used in an overlay, an object module must contain only "pure code" (program statements that are not modified while the program is running). Aside from this restriction, any object module may be used in either a program or an overlay file. Thus, you may bind the same object modules in different ways. You can then use the binder listings (see Chapter 3) to determine which combination of program and overlay file is best.

***NOTE:** For information about how to write the overlay-handling sections of your program, see the programmer's reference manual for the language you are using.*

When you bind overlays, the binder creates one program file and one overlay file. The overlay file has the same name as the program file, but with the suffix .OL instead of .PR. The binder allocates space in the program file for the overlay nodes, with each node allocated enough space to hold its largest overlay. Using information from the object modules that are in the overlays, the binder resolves the overlay control symbol references within the program (which includes overlaid code, if necessary).

When you execute a program that uses overlays, the overlay file must be in the same directory as the .PR file. The command line for an overlaid program is the same as for a non-overlaid program.

Using the Binder

This chapter explains how to execute the binder program and how to interpret its printed output. It explains the command line and switches that tell the binder how to use object modules to create program, overlay, and system files, what type of load map to produce, and where to display or store output. The chapter also contains annotated binder listings and examples of command lines.

Binder Command Line

To execute the binder, you type the CLI command line described below. You supply as arguments to the command the names of the object files and libraries that you wish to bind. Using function switches, you specify the type of executable output and the type of listing that you wish produced, and whether or not you want a symbol table as output. And, if you are binding a program that uses overlays, you use an overlay designator in the command line to indicate which modules are to be used in each overlay node.

The format of the binder command line is

XEQ BIND{*/switch*} *obname* {ov} *obname2* {ov}...{*V*}

where:

XEQ is the CLI command to execute the program (you may abreviate XEQ to X);

switch is one of the function switches listed in Table 3.1 or described in Appendix C.

obname is the name of an object module or library file (with optional .OB or .LB extension);

ov is an overlay designator (See "Overlays," below);

V is a symbol defined by including the symbol name, followed by /VAL= and an octal value. For example, including XYZ/VAL=1234 on the command line defines a symbol XYZ and assigns it the value 1234.

Switch	Effect
/ALPHA	When used with the /L= switch, appends to the load map an alphabetically sorted list of global symbols. Used without the /L= switch, has no effect.
/E=*filename*	Puts error messages into *filename*. If you do not use this switch, the error messages go to the console.
/ERMES	Creates an MP/AOS error message file.
/KERNEL	Builds the kernel section of the MP/AOS operating system.
/L=*filename*	Puts the load map into *filename*. If you do not use this switch, no load map is generated.
/LIBLIST	When used with the /L= switch, lists individually all modules bound from libraries. Used without the /L= switch, has no effect.
/N	Prevents binding of the system library file OSL.LB.
/NUMERIC	When used with the /L= switch, appends a list of global symbols, sorted by numeric value. Used without the /L= switch, has no effect.
/P=*filename*	Use *filename* instead of the name of the first object file in the command line as the root for program file names.
/REV=*value*	Sets the program's revision number to *value*. The number may contain a period to separate major and minor revision numbers.
/SUPER	Builds the supervisor section of the MP/AOS operating system.
/TASKS	Specifies the maximum number of tasks that the program may have active at one time.
/SYS	Builds an MP/AOS system file.

Table 3.1 Binder command line switches

NOTE: *Additional switches used for cross-development on other operating systems are described in Appendix C.*

Filename Arguments

The binder looks in your working directory and then through your searchlist for each filename you list as an argument to the command line. The .OB and .LB extensions are optional — if you do not specify an extension, the binder first looks for each filename with the .OB extension, then with the .LB extension, and finally with no extension. If you do specify one of these extensions, the binder looks only for the file as you have typed it.

Order Of Filenames In Command Line

The binder converts relative addresses assigned by the assembler or compiler in such a way that the first module listed on the command line receives the lowest absolute addresses, and other modules are added in their order on the command line. Figure 3.1 illustrates how the binder builds the pure area (NREL 0) of a program file from corresponding areas of the input modules. In this example, the modules were specified on the command line in the order ABC. The other areas of the program (ZREL, NREL 1) are built analogously.

In many cases, the order in which you list input modules will not affect the way the program runs; if, however, your program does not run as you intend, consider the effect of binding the same modules in a different order.

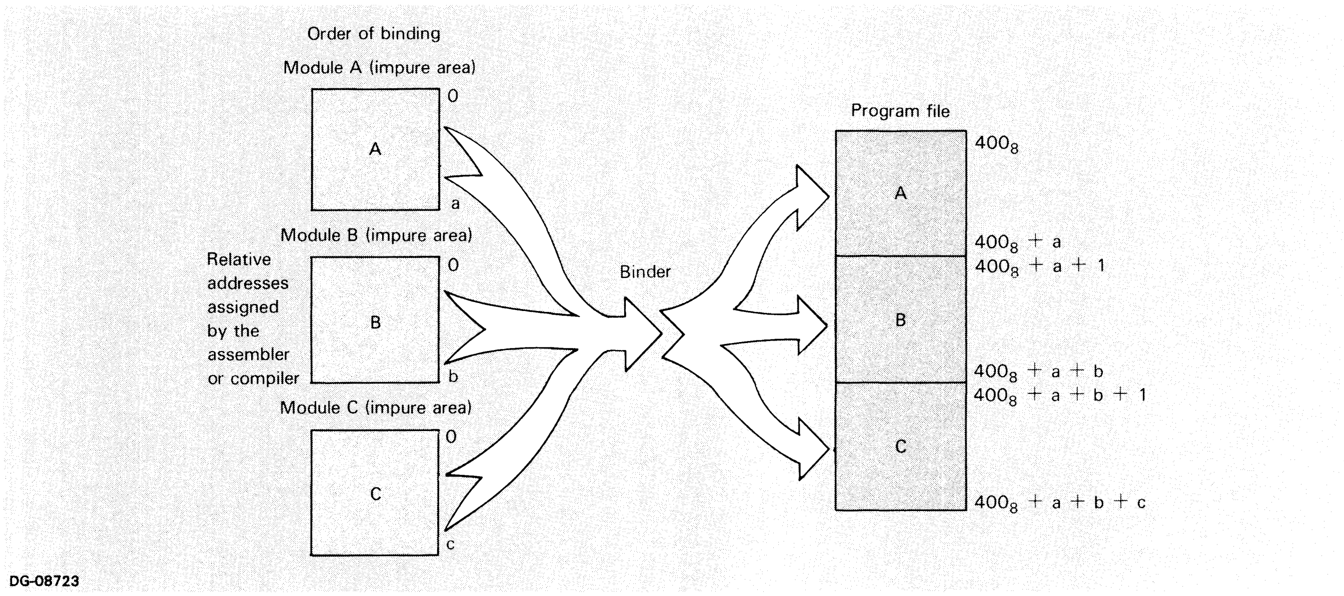


Figure 3.1 Binder input modules (impure area) into program file. Other areas are built analogously.

The name of the first file in the command line determines the names of all the program files (not load map or error files). To name the output file(s), the binder strips off whatever extension is on the name of the first file in the command line and appends the appropriate name extension (.PR, .OL, .ST, etc.) You may override this naming procedure by using the /P=*filename* switch.

Symbols in Input Modules

The binder reports an error when an already defined symbol is declared as an entry symbol in a non-library module. See below for an explanation of how the binder handles symbols that are defined in more than one library module.

Using Libraries

When you specify a library name or names in the binder command line, and if there are unresolved symbols after all other modules have been bound, the binder checks each library specified. The binder scans each library once per pass from beginning to end and binds the first module that declares one of your program's unresolved symbols as an entry symbol.

Symbols in Library Modules

If the binder uses a library module that declares an entry symbol that has already been declared an entry symbol elsewhere, it ignores the redefinition and does not report an error. This allows you to create *default entries* that are used only if the symbol has not been defined in a previous module.

Order of Modules in Libraries

The order of modules in libraries is important for two reasons.

As explained above, the binder reads the library from beginnng to end, and uses the first definition of multiply-defined symbols. Also, if module A occurs in a library before module B, and B gets bound and refers to an entry in A, then A must also get bound. Since B is unable to cause A to be bound, another mechanism (such as .FORCE in A, or another previous reference to A) must be used to cause A to be bound.

Force-Bind Flag

When the force-bind flag for an object module in a library file is set, the binder automatically binds that module, whether or not it contains symbols referenced elswhere in the program. You may use the library file editor to set the force-bind flag for each module in a library.

System Library

Data General supplies the library OSL.LB with each MP/AOS operating system. This library contains routines that are needed to support certain system functions. Unless you use the /N command line switch, the binder automatically checks this library after the other files in the command line have been processed.

Executable Output Files

The binder can produce two classes of executable output files: program/overlay files and system files.

Program/Overlay Files

When you do not use the /SUPER or /KERNEL switch, the binder produces a program file (.PR) and, when you use overlay designators, an overlay file. The program file can be executed by MP/AOS systems; overlay files are used with program files.

To specify overlays in the command line, use the designators listed in Table 3.2. These symbols must be separated from all other arguments by spaces or a comma.

Designator	Description
!*	Marks beginning of list of module names that use the same overlay node.
!	Separates lists of module names within an overlay node.
*!	Marks end of list of module names that use the same node.

Table 3.2 Overlay designators

The following command line shows how to use overlay designators in a command line.

```
) XEQ BIND/L=MAIN.MAP MAIN S1 !* S2 ! S3 *! !* S4 S5 ! S6 *! ↲
```

The inputs to the binder are a main program MAIN.OB and six subroutines S1.OB to S6.OB. The binder creates the program file MAIN.PR and the overlay file MAIN.OL. MAIN.PR contains subroutine S1 and two overlay nodes. The first overlay node has two overlays, which contain S2 and S3, respectively. The other node also has two overlays: one contains both S4 and S5, and the other contains S6. The binder also produces a load map in file MAIN.MAP.

System Files

The binder provides two switches that enable you to build the MP/AOS system. The /KERNEL switch builds the kernel of the system, and the /SUPER switch builds the supervisor. These elements of the operating system are described in *MP/AOS System Programmer's Reference*, DGC No. 093-400051.

In most cases, however, system files can be more easily created with the SYSGEN utility described in *System Generation and Related Utilities*, DGC No. 069-400206.

See Appendix C for an explanation of how to use the binder to generate an MP/OS system.

Program Parameter Symbols

In addition to detecting and tabulating global symbols in the input modules, the binder also keeps track of parameters that describe your program. You may reference these values using the symbols listed in Table 3.3.

Symbol Name	Value
?AOS	0 = MP/OS or MP/AOS program file 1 = AOS program file
?CLOC	Starting address of an unnamed common block
?CSZE	Size of an unnamed common block
?NMAX	One plus the highest impure memory address used by the program
?NTAS	Maximum number of tasks in the program
?OVTB	Starting address of the overlay table
?REV	Program revision number
STYP?	0 = MP/OS 1 = AOS 2 = MP/AOS

Table 3.3 Program parameter symbols

Binder Listings

Listings include the load map, the symbols list and the error listing. Each is described briefly below.

Load Map

When you include the /L=*filename* switch on the binder command line, the binder produces a load map and places it in *filename.* The listing is in two sections.

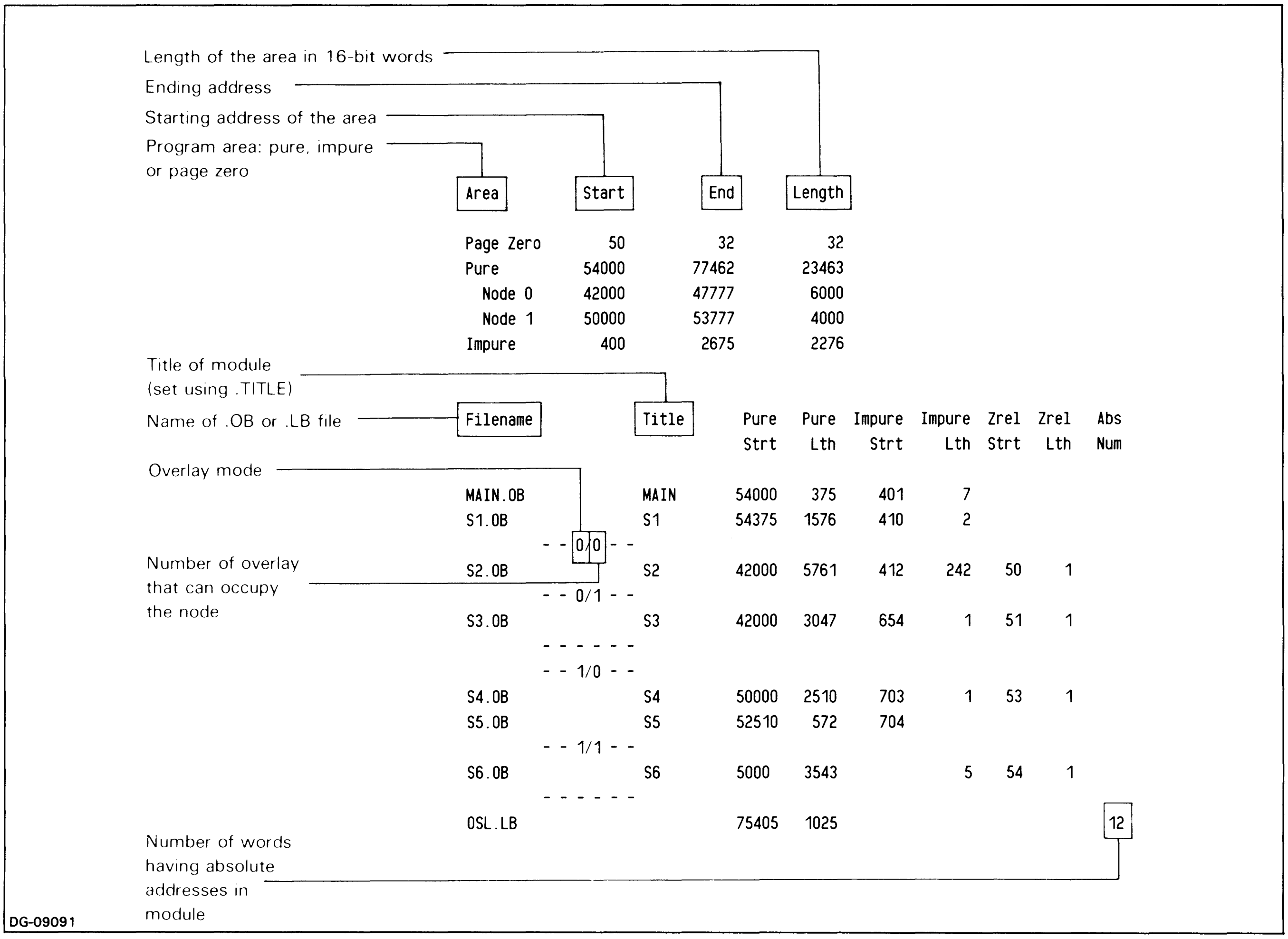
The first section lists the starting address, ending address and length (in words) of the page zero, impure, and pure areas of the program and each overlay node.

The second section lists the filename, title, starting addresses and lengths of pure, impure, and lower page zero areas for each object file listed separately on the command line (i.e., not those modules taken from libraries). It also lists, for each of these, the number of words whose absolute addresses were assigned by an assembler or compiler.

The second section of the load map also lists the same information for each overlay node.

When you use the /LIBLIST switch, the load map lists the same information for each module bound from a library as it does for modules not in libraries. When you do not use the /LIBLIST switch, the load map lists summaries of this information for each library.

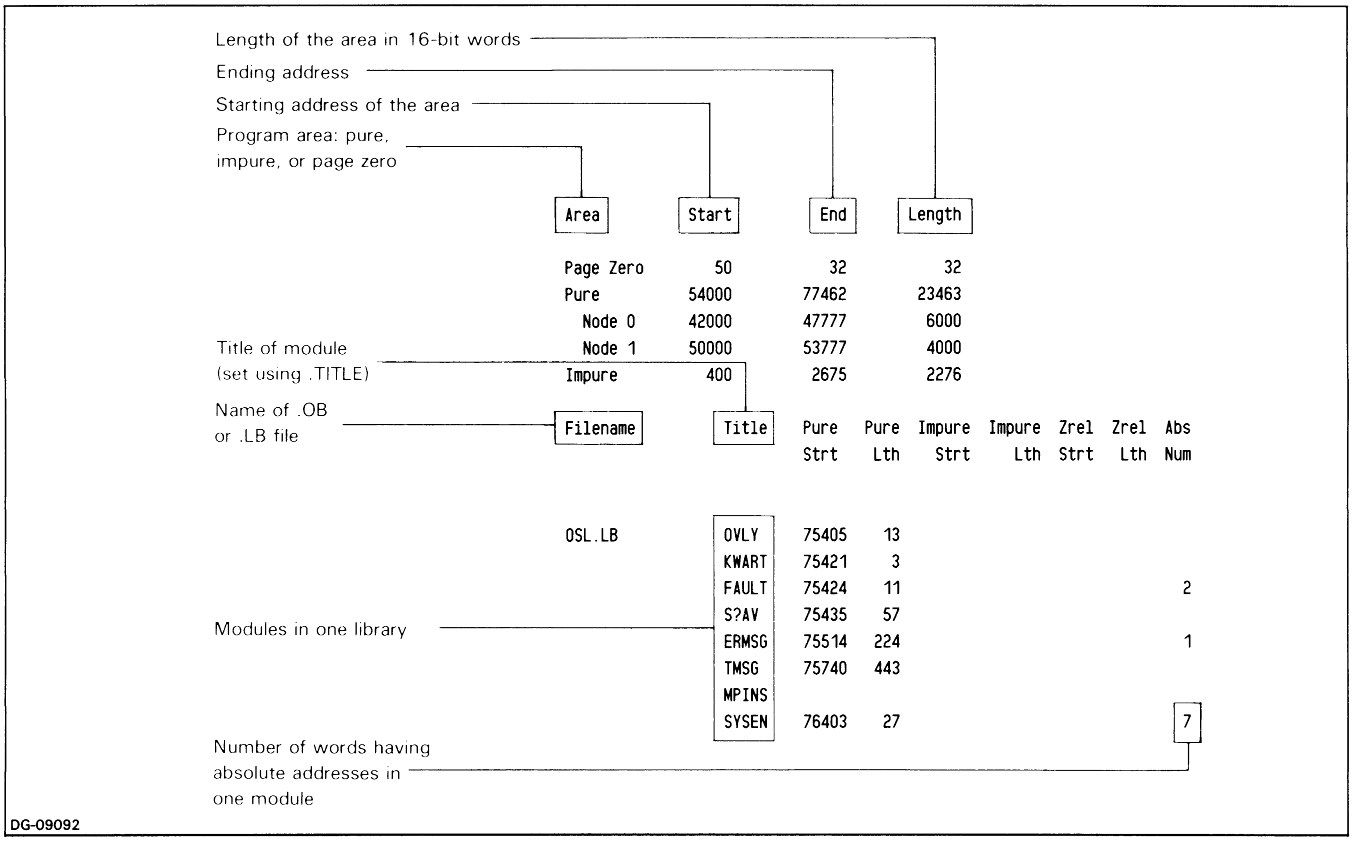
Figure 3.2 shows a binder load map produced using the /L = switch alone. Figure 3.3 shows a binder load map produced using the /LIBLIST switch with the /L = switch.



```
Area         Start     End      Length

Page Zero      50        32        32
Pure        54000     77462     23463
  Node 0    42000     47777      6000
  Node 1    50000     53777      4000
Impure        400      2675      2276

Filename            Title    Pure   Pure  Impure  Impure  Zrel  Zrel   Abs
                             Strt    Lth    Strt     Lth  Strt   Lth   Num

MAIN.OB             MAIN    54000    375     401       7
S1.OB               S1      54375   1576     410       2
           - - 0/0 - -
S2.OB               S2      42000   5761     412     242    50     1
           - - 0/1 - -
S3.OB               S3      42000   3047     654       1    51     1
           - - - - - -
           - - 1/0 - -
S4.OB               S4      50000   2510     703       1    53     1
S5.OB               S5      52510    572     704
           - - 1/1 - -
S6.OB               S6       5000   3543               5    54     1
           - - - - - -
OSL.LB                      75405   1025                                12
```

Length of the area in 16-bit words
Ending address
Starting address of the area
Program area: pure, impure or page zero
Title of module (set using .TITLE)
Name of .OB or .LB file
Overlay mode
Number of overlay that can occupy the node
Number of words having absolute addresses in module
DG-09091

Figure 3.2 Partial binder listing (without /LIBLIST)



Length of the area in 16-bit words
Ending address
Starting address of the area
Program area: pure, impure, or page zero
Title of module (set using .TITLE)
Name of .OB or .LB file
Modules in one library
Number of words having absolute addresses in one module
DG-09092

```
Area         Start      End      Length

Page Zero       50       32          32
Pure         54000    77462       23463
   Node 0    42000    47777        6000
   Node 1    50000    53777        4000
Impure         400     2675        2276

Filename         Title    Pure   Pure  Impure  Impure  Zrel  Zrel  Abs
                          Strt   Lth    Strt     Lth   Strt   Lth  Num

OSL.LB           OVLY     75405   13
                 KWART    75421    3
                 FAULT    75424   11                                 2
                 S?AV     75435   57
                 ERMSG    75514  224                                 1
                 TMSG     75740  443
                 MPINS
                 SYSEN    76403   27                                 7
```

Figure 3.3 Partial binder listing (with /LIBLIST)

Symbols Listing

When you include the /ALPHA switch with the /L= switch, the binder appends an alphabetic list of global symbols to the load map; when you include the /NUMERIC switch, the binder appends a list of global symbols sorted by numeric value.

Error Listing

The binder sends a listing of warnings and errors to the console, or to the file you specify using the /E= switch. Errors prevent the binder from producing an executable file. Warnings are caused by conditions that must be corrected, but that do not prevent the binder from creating an executable file.

Object Module Formats

This chapter describes the organization of object modules. You do not need the information in this chapter unless you are writing a system utility such as an assembler or compiler, but it may give you an insight into the workings of the binder.

Contents of Object Modules

Object modules contain the instructions and data of your source program, translated into 16-bit binary words, and the relocation information for each program component (symbols, addresses and data words). Addresses can have relocation properties that the binder uses to compute the order of words in the program file.

In addition to a binary translation and relocation properties of each program component, object modules also contain summary and index information that the binder uses to construct the program file. This information includes the names and locations of local, entry, and external symbols; the number of tasks in the program; and the names and lengths of common areas.

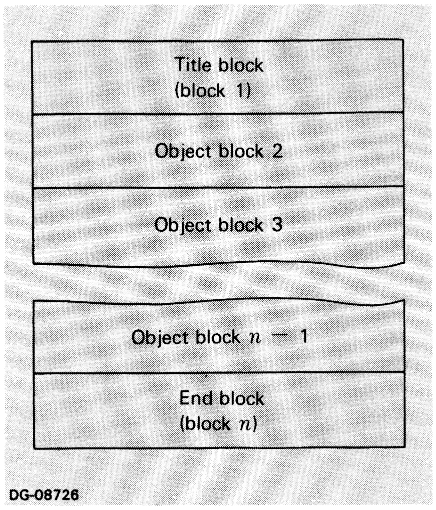


Figure 4.1 Object module format

Each object module is organized into *object blocks*. These blocks identify the module, mark its beginning and end, identify the type and relocation property of every relocatable entity, and contain the summary information just described. Object modules conform to the format illustrated in Figure 4.1. The first block is a title block and the last block is an end block; there may be only one of each of these per module. Between the title block and end block are other object blocks that contain the program components. Within an object module, there may be any number of object blocks. External symbol blocks must precede the data blocks or entry blocks that contain data or values relocated from the external symbols. Object block types are listed in Table 4.1.

Object Block Format

Object blocks are made up of a *block header* and a *block body*. The block header has the same format for all block types; the format of the block body differs for each block type.

Block Header

The first three words of every block comprise the block header. The rightmost, low-order byte of the first word contains a number that indicates the type of the block, as summarized in Table 4.1. The second word in the block is the sequence number, which is used as a validity check. (The title block must be number 1, and all subsequent blocks must be consecutively numbered.) The third word, *length*, specifies the total number of words in the block, including the header.

One bit of the leftmost, high-order byte of the first word is used only in the title block, which is described below. The rest of the byte is reserved otherwise.

Number (octal)	Block Type
0	Data
1	Title
2	End
3	Unlabelled common
4	External symbols
5	Entry symbols
6	Local symbols
7	Library start
10	Reserved (address information)
11	Reserved (shared library)
12	Task
13	Reserved (limit)
14	Named common
15	Reserved (accumulating symbol)
16	Reserved (debugger symbols)
17	Reserved (debugger lines)
20	Reserved (lines title)
21	Library end
22	Reserved (statistics)
23	Reserved (partition definition)
24	Reserved (conditional load block)
25	Reserved (overlay definition)
26	Reserved (binder revision)
27	Reserved (filler)
100	Library load

Table 4.1 Object block types

Block header format is shown at the top of Figure 4.2.

Block Body

This section describes formats of object block bodies according to the numerical order of block types shown in Table 4.1. All object blocks begin with the three-word block header described above; block bodies therefore begin with the fourth word.

Relocation Information

The relocation information for each program component is embedded in the block body. This information consists of a *relocation base* and a *relocation operation*.

The relocation base is a number that indicates whether the program component is absolute, is to be relocated in the pure or impure area of this program, or is to be relocated relative to an externally defined symbol.

The relocation operation is the number of the formula that the binder should use to calculate the absolute address or value of the program component. These operations are described in Table 4.2.

Operation Number (Octal)	**Operation Name**	**Description**
0	Absolute	No operation performed on data word
1	Word	Adds base to data word.
2	Byte	Adds (base * 2) to data word.
3	Displacement	If bits 1 to 15 of data word are 0: adds base to data word If bits 6 and 7 (MRI index bits) are 0: adds base to data word and issues displacement overflow error if any bits in high byte change from original data word. If bits 6 and 7 are not zero: takes signed 8-bit value in low byte of data word and adds relocation base. Issues displacement overflow error if signed result cannot be represented in 8 bits. The 8-bit result is placed in the low byte of the original data word (leaving the high byte unchanged).
4	Subtraction	Subtracts data word from base.
5	Overlay entry	Specifies overlay descriptors.
6	Multiplication	Multiplies data by base.
7 to 10	-----	Reserved for future use.
11	GREF	Adds low-order 15 bits of base to low-order 15 bits of data. Bit 0 of resulting data word has same value as bit 0 of original data word.
12-17	-----	Reserved for future use.

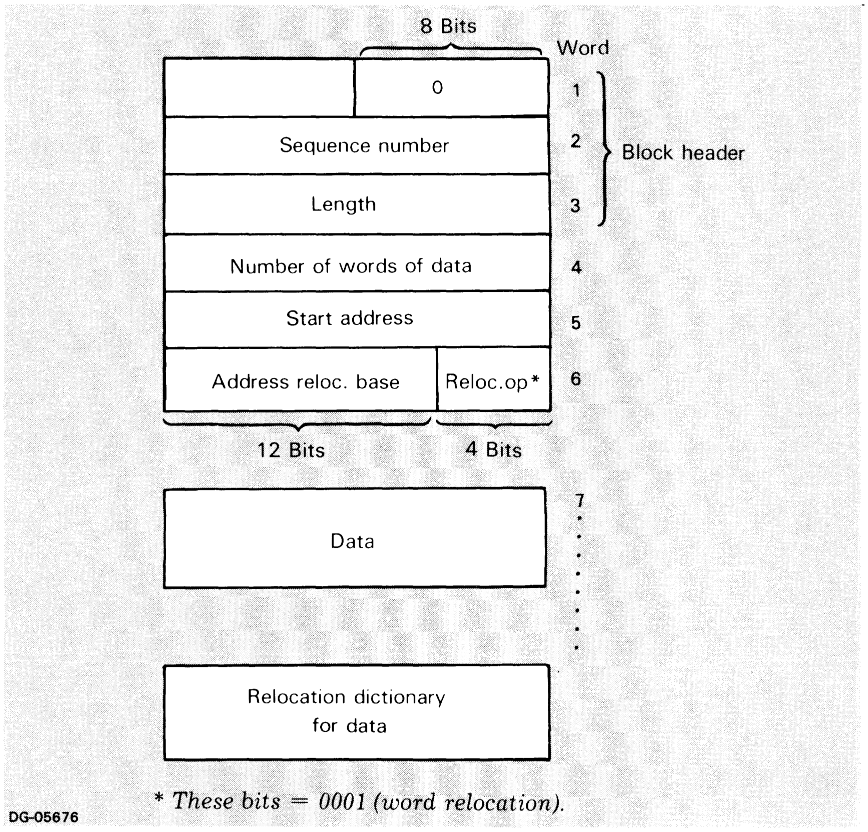
Table 4.2 Relocation operations

Data Block

Data blocks (type no. 0) contain the code and data of the source program being bound. Word 4 contains the number of data words in the block body. Word 5 contains the start address, relative to the relocation base, to which data words are to be relocated. The relocation base is specified in bits 0-11 of word 6. This formula implies that all words in a data block are placed in the same partition.

The data words begin at word 7 of the block. Following these words are relocation dictionary entries. These are two-word entries that contain an offset and another relocation base and operation.

Figure 4.2 illustrates the format of a data block. Figure 4.3 illustrates the structure of relocation dictionary entries.



* *These bits = 0001 (word relocation).*

Figure 4.2 Data block structure

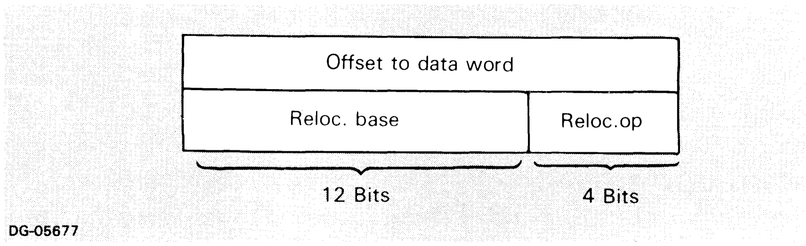


Figure 4.3 Structure of relocation dictionary entries

Title Block

The title block (type no. 1) is the first in every object module.

The fourth bit in the first word of the header (not the body) is the force bind flag for the module. When this bit is 1, the flag is set (see Chapter 3).

Word 4 contains the revision number of the program, word five contains the title length in bytes, and word 6 is a byte pointer (relative to first word of block) to the title string, which begins in word 7.

Figure 4.4 illustrates the structure of the title block.

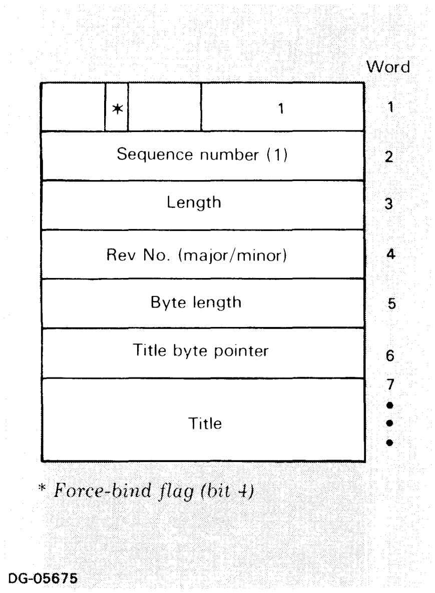


Figure 4.4 Title block structure

End Block

The structure of the end block (type no. 2) is illustrated in Figure 4.5. Word 4 of the block specifies the starting address of the program. Word 5 specifies its relocation base and operation.

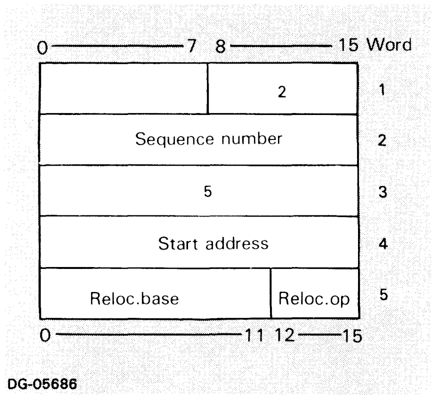
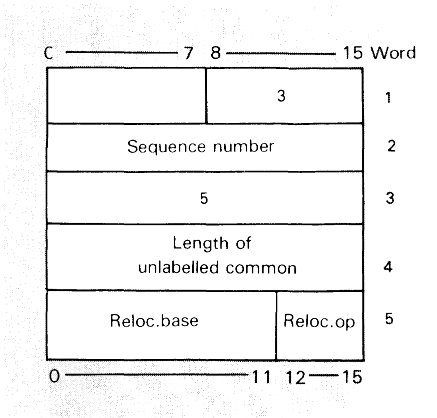


Figure 4.5 End block structure

Unlabelled Common Block

Unlabelled common blocks (type no. 3) specify an area of the logical address space without a user-defined name (used by more than one module).

The structure of an unlabelled common block is illustrated in Figure 4.6. Word 4 specifies the length of the area. If there are several blocks that specify different lengths, the binder uses the largest area specified. Word 5 contains the relocation base and operation of the area length. Both relocation values must be 0, to indicate an absolute value with no relocation.



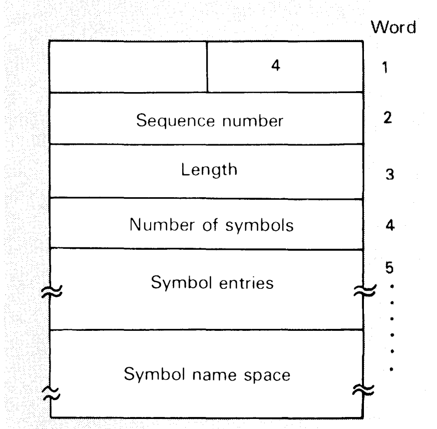
DG-05684

Figure 4.6 Unlabelled common block structure

External Symbols Block

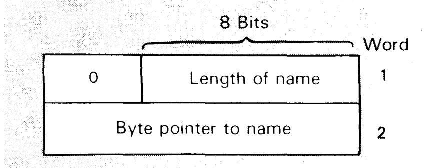
External symbols are global symbols defined in another object module.

Figure 4.7 illustrates the structure of an external symbols block. Figure 4.8 illustrates a symbol entry in the external symbols block.



DG-05681

Figure 4.7 External symbols block structure



DG-05682

Figure 4.8 Symbol entry in external symbols block

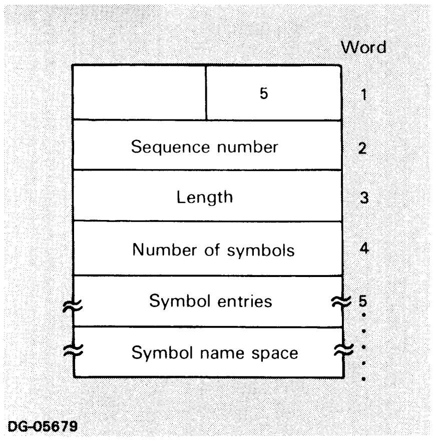


Figure 4.9 Entry symbols block structure

Entry Symbols Block

Entry symbols are global symbols that are defined in one object module and referenced in others.

Figure 4.9 illustrates the structure of an entry symbols block (type no. 5).

Word 4 of the block specifies the number of symbols it contains. The remainder of block consists of a four-word entry for each symbol, and a symbol name space. Each symbol entry contains

- type,
- byte length of the symbol's name,
- a byte pointer into the name space,
- value for the symbol,
- relocation base and operation for the symbol's value.

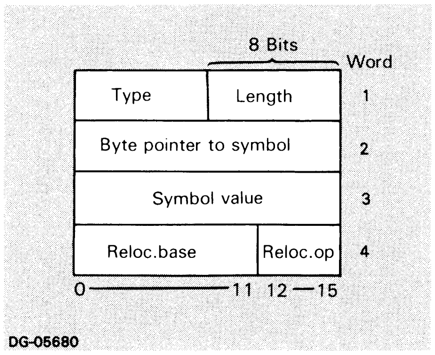


Figure 4.10 Symbol entry in entry symbols block

The symbol name space is a string of bytes in which the names of the symbols are spelled out. The symbol names may be concatenated or overlapped since each symbol name has its own byte pointer and length. For example, the single string CAB could be used to contain the names of symbols C, A, B, CA, AB, and CAB.

The symbol's type is 4 for overlay entries, and 0 for all other entries. If the type is 4, the relocation operation must be 5 to indicate overlay relocation.

Figure 4.10 illustrates the structure of a symbol entry in an entry symbols block.

Local Symbols Block

Local symbols are those used only within the object module that defines them. The binder ignores local symbols blocks; they are generated by Data General assemblers and compilers for use by debuggers on other systems.

A local symbols block (type no. 6) has the same structure as an entry symbols block (see Figures 4.9 and 4.10). The only difference is the block type number in the first word of the block. Word 4 of the block specifies the number of symbols it contains. The remainder of the block consists of a four-word entry for each symbol, and a symbol name space. Each local symbol entry contains

- type,
- byte length of the symbol's name,
- a byte pointer into the name space,
- value for the symbol,
- relocation base and operation for the symbols's value.

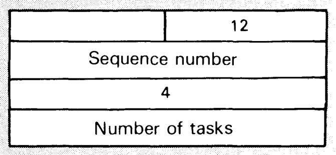
The symbol name space is a string of bytes in which the names of the symbols are spelled out. The symbol names may be concatenated or overlapped since each symbol name has its own byte-pointer and length. For example, the single string CAB could be used to contain the names of symbols C, A, B, CA, AB, and CAB.

Task Block

Task blocks (type no. 12) are used in multitasked programs to tell the binder how many tasks the program requires.

Figure 4.11 illustrates task block structure.

After the block header, a single word specifies the number of tasks.



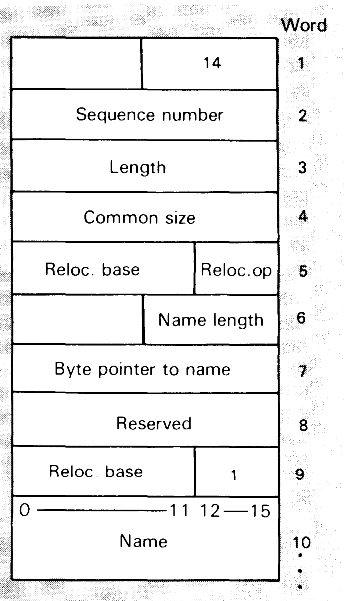
DG-05685

Figure 4.11 Task block structure

Named Common Block

A named common block (type no. 14) describes a common area with a user-defined name. If the binder finds several blocks with the same name, it assigns one area for all the blocks. If the area is specified with several different sizes, the binder allocates the largest area requested.

The structure of a named common block is illustrated in Figure 4.12. Word 4 specifies the size of the common area. Word 5 must be zero. This indicates absolute relocation for the value in word 4. Word six specifies the length of the area's name (in bytes), and word 7 is a byte-pointer to the name. The leftmost 12 bits of word 9 are ignored by the binder; the rightmost 4 bits represent 1, since word relocation is always used. The remaining words contain the block's name.



DG-05683

Figure 4.12 Named common block structure

Library File Format

Libraries consist of a library start block, one or more object modules, and a library end block. The format of a library file is shown in Figure 4.13. The MP/AOS libary file editor constructs library files (including start and end blocks) from a list of object modules (see Part 3).

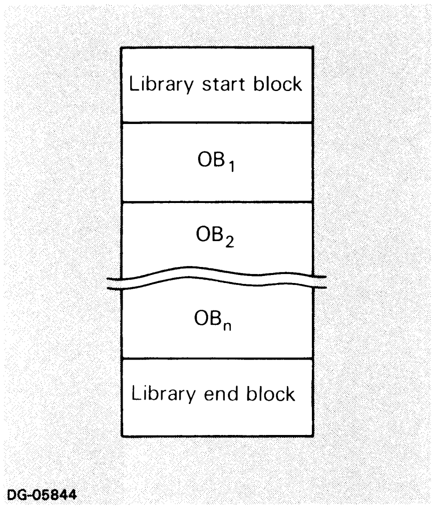


Figure 4.13 Library format

Library Start Block

Figure 4.14 shows the structure of a library start block (type no. 7).

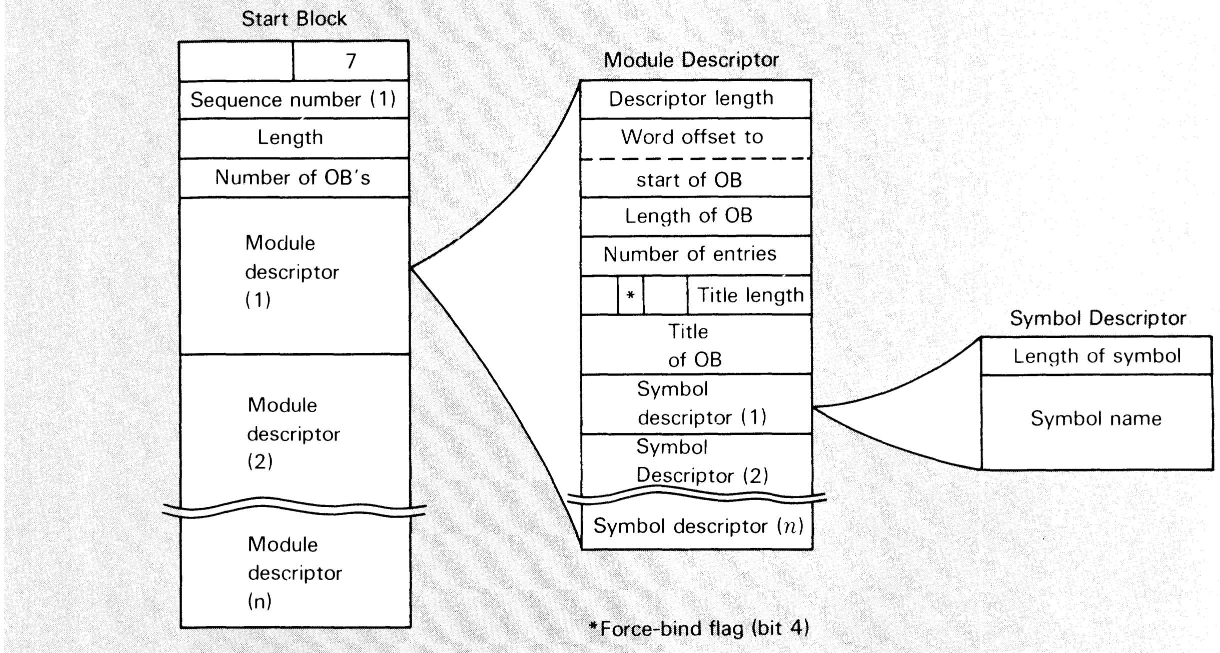
The library start block begins with the standard three-word block header; the sequence number is always 1.

The next word after the header contains the number of object modules in the library.

Beginning with word 5 are *module descriptors* for every module in the library. Each module descriptor consists of a six-word header, the object module title string, and a *symbol descriptor* for each symbol that is declared as an entry to the module.

The first word of a module descriptor gives the descriptor's word length. The next two words are a 32-bit number giving the word offset from the start of the first module in the library to the start of the object module being described. (The offset value in the first module descriptor is 0). The fourth word gives the word length of the OB module and the fifth word gives the number of entry symbols within the module.

The left byte of the sixth word contains the force-bind flag (bit 4) for the module (see Chapter 2). Other bits in this byte are reserved. The right byte in this word gives the length (in bytes) of the module's title. The sixth word is followed by the title string. After the title string, the remainder of the module descriptor consists of the symbol descriptors. Each symbol descriptor contains one word giving the symbol's length, and a byte string containing the symbol's name.



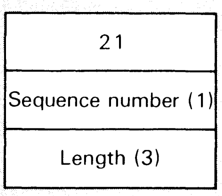
DG-05846

Figure 4.14 Format of library start block and module descriptor

Library End Block

Figure 4.15 shows the structure of a library end block (type No. 21).

All libraries end with this block, which consists of only the three-word header. The sequence number is always 1 and the length is always 3.



DG-05845

Figure 4.15 Format of library end block

Library File Editor and Symbol Cross-Reference Analyzer

Part 3

Library File Editor and Symbol Cross-Reference Analyzer

This chapter describes two utility programs that maintain libraries of object modules. The library editor (LED) enables you to build library files from object files and from other libraries. The symbol cross-reference analyzer (SCAN) enables you to inspect the use of global symbol names in your program and to identify mutliply-defined and undefined symbols.

Library File Editor

The library file editor is a utility program that builds libraries from object files or libraries that you specify as input. To build the library, the editor abstracts the information it needs from the modules and constructs the library start block, strips the files of the MP/AOS file header, concatenates the object modules, and appends the library end block. Library structure is explained in greater detail in Chapter 4.

Using the library file editor, you may set or clear the library start block force-bind flag for each module. A description of the force-bind flag appears in this chapter.

LED Command Line

The format of the LED command line is

XEQ LED {*/function-switch*} *filename*{*/arg-switch*}...

where:

funtion-switch is one or more of the function switches listed in Table 1.1.

filename is the filename or pathname of an object file or library file.

arg-switch is the /F or /C argument switch.

Switch	Description
/LB=*filename*	Specifies *filename*.LB as the name of the library being created. If you do not use this switch, the library name is derived from the first filename on the command line.
/L	Sends a listing of module titles and sizes to the console.
/L=*filename*	Places a listing of module titles and sizes in *filename.*
/E	Sends a listing of any error messages to the console.
/E=*filename*	Places a list of error messages in *filename.*

Table 1.1 LED switches

Creating a New Library File

To create a new library file, invoke the library editor using the /LB switch (this switch is optional but recommended). To build the library, the editor abstracts the information it needs from the object files and library files listed on the command line, constructs the library start block, concatenates the object modules, and appends the library end block.

Filename Arguments to Library Editor

The editor looks in your working directory and then through your searchlist for each filename you list as an argument to the command line. The .OB and .LB extensions are optional; if you do not type an extension, the binder looks first for each filename with the .OB extension, then with the .LB extention, and finally with no extension.

If you do specify one of these extensions, the editor looks only for the file as you specified it.

Order of Filenames in Command Line

The editor places modules in the library in the order that they are listed in the command line. As explained in Chapter 3, the order of modules in the library is important, since the binder scans a library only once, from beginning to end. Before building a library, determine the order of modules to ensure that all symbols are defined when the library is bound.

When you do not use the /LB=*filename* switch, the name of the first file in the command line determines the name of the library file produced. To name the new library, the editor strips off whatever extension is on the name of the first file and appends .LB.

Listing the Library

To obtain a list of the titles and sizes of the modules in the library you are creating, use the /L=*filename* switch. If you use the switch without a filename, the list is displayed on your console. When you bind the library, you may obtain a list of the modules used by including the /LIBLIST switch on the binder command line.

Force-Bind Flag

As explained in Chapter 2, each object module in a library has two force-bind flags associated with it. One flag is in the module title block, and the other is in the library start block. When the program is bound, the flag in the library start block determines whether the module will be automatically bound.

The force-bind flag of the title block can be set in assembly language modules by using the .FORCE pseudo-op (see Part 1, Chapter 7).

The force-bind flag in the library start block can be set in two ways:

- by using a switch on an object file name in the LED command line. The /F switch sets the flag; the /C switch clears it;
- by using a switch on a library filename in the LED command line. The /F switch sets the flag for every module in the program, and the /C switch clears it.

If neither switch is used, the flag is copied unchanged from the module title block to the library start block.

For example, assume that you have these files in your working directory:

- an object file PRO.OB with the force-bind flag set;
- an object file NEST.OB with the force-bind flag not set;
- a libary file KLV.LB in which some of the title-block force-bind flags are set and some are not.

If you now type

```
XEQ LED PRO.OB NEST.OB/F KLV.LB/C
```

the editor will construct a new library called PRO.LB. In it, the library start block force-bind flag for PRO.OB and NEST.OB will be set, and the libary start block flags for all the other modules will be cleared. (For simplicity, this example does not show the use of the /LB= switch. Remember, the use of /LB= is recommended.)

None of the force-bind flags in the title blocks for these modules will be affected.

Symbol Cross-Reference Analyzer

The symbol cross-reference analyzer (SCAN) is a utility program that produces information about the use of global symbol names in libraries and object modules. SCAN provides an alphabetical list of all symbol names in the modules and, for each symbol name, an alphabetical list of the modules in which the symbol is used. It identifies symbols that are undefined within the set of scanned modules, as well as those that are multiply-defined.

You can use this program to check for symbols that are multiply defined and to help you look for symbols that may be inconsistently defined. You can also use it to determine that a symbol name you are considering is unique, and to find all affected modules when you change a symbol definition.

SCAN Command Line

To execute SCAN, type the command line given below. Using function and argument switches, you can modify the criteria for including symbol names in the listing (or for including a symbol count and no symbol names) and for suppressing certain information on the listing.

The format of the SCAN command line is

XEQ SCAN{*/function-switch*} *filename*{*/arg-switch*}...

where:

funtion-switch is one of the function switches listed in Table 1.2. The minimal unique abbreviation is acceptable. (Note that the mnemonics associated with the /OPTIONS= switch contain a letter and minus sign.)

filename is the name of an obect file or library file.

arg-switch is the /OMIT argument switch described under "Arguments to SCAN," below.

Switch	Description
/COUNT	Provides a count of entry and external symbols. Neither creates nor allocates space for cross reference list.
/L	Sends the listing to the lineprinter, @LPT.
/L=*filename*	Places listing in *filename*.
/OPTIONS=	Suppresses output, as follows:
W—	All warning messages,
U—	Undefined-symbol flags,
M—	Multiply-defined symbol flags,
Z—	Zero-reference flags,
H—	Page headings and formfeeds on output,
S—	Summary listing.
/SYSTEM	Includes symbols containing a question mark or starting with a period (except virtual symbols). Otherwise these symbols are not listed.
/VIRTUAL	Includes symbols beginning V?, P? and Q?. Otherwise these symbols are not listed.

Table 1.2 SCAN function switches

Arguments to SCAN

The arguments to the SCAN are the names of libraries or object files in your working directory or on your searchlist. The filename arguments may be in any order. If a filename does not have an extension, the SCAN looks in turn for a filename with the .OB extension, the .LB extension, and finally no extension.

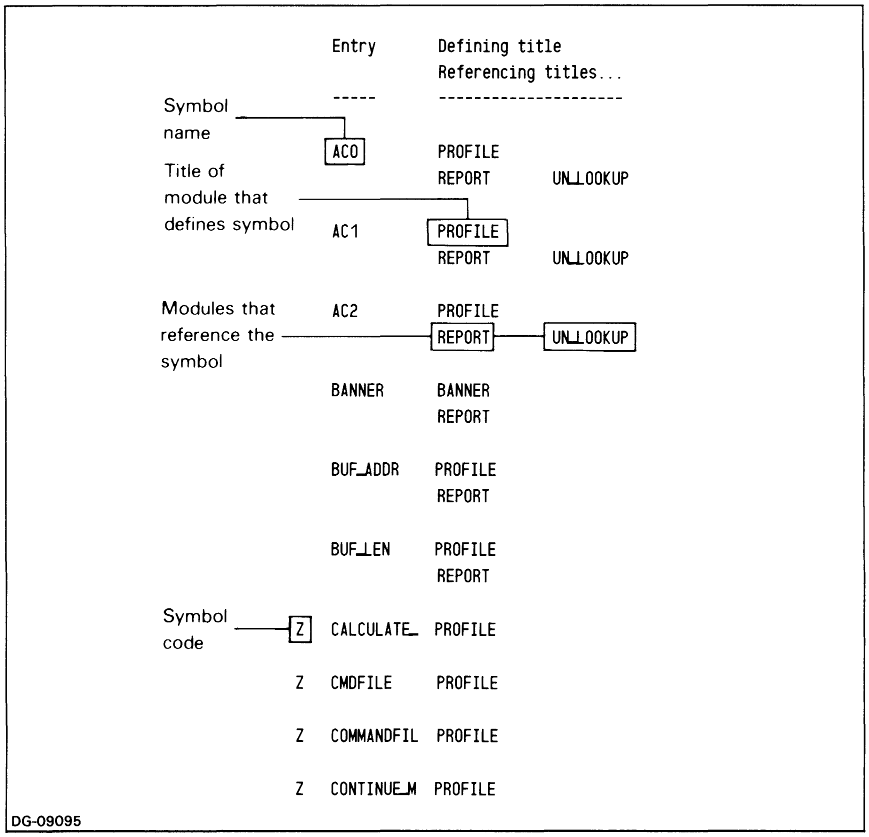
When you append the switch /OMIT to an argument, the listing does not include that title in the cross-reference listing. For example, if a file has an object module entitled BOB, containing an external reference to FRED, you can use /OMIT to cause SCAN to omit BOB from the list of modules referring to FRED.

SCAN Output

SCAN produces a single listing with four fields, as illustrated in Figure 1.1.

Starting in the fourth column is a vertical list of symbol names, alphabetically ordered. On the same line as the symbol name is a horizontal list of any modules in which the symbol is declared as an entry symbol. Beneath this line are one or more horizontal lines that alphabetically list modules in which the symbol is declared as an external symbol. Modules that contain symbol definitions are listed one per line.

When a symbol is declared as an entry symbol in more than one module, as an entry symbol but not as an external symbol, or as an external symbol but not an entry symbol, the SCAN listing includes a single-letter code to the left of the symbol name. The meaning of these letters is explained in Table 1.3.



```
      Entry       Defining title
                  Referencing titles...
      -----       ---------------------

      AC0         PROFILE
                  REPORT        UN_LOOKUP

      AC1         PROFILE
                  REPORT        UN_LOOKUP

      AC2         PROFILE
                  REPORT        UN_LOOKUP

      BANNER      BANNER
                  REPORT

      BUF_ADDR    PROFILE
                  REPORT

      BUF_LEN     PROFILE
                  REPORT

    Z CALCULATE_  PROFILE

    Z CMDFILE     PROFILE

    Z COMMANDFIL  PROFILE

    Z CONTINUE_M  PROFILE
```

Symbol name
Title of module that defines symbol
Modules that reference the symbol
Symbol code
DG-09095

Figure 1.1 Partial SCAN listing

Code Letter	Meaning
M	Multiply defined. The symbol is declared as an entry symbol in more than one module.
U	Undefined. The symbol is declared as an external but none of the modules scanned defines it as external.
Z	Zero reference. The symbol is declared as an entry but none of the modules scanned defines it as an entry.

Table 1.3 SCAN listing codes

ASCII Character Set

A

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT. TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑—	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ∧
95	137	5F	← OR —
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

Macroassembler Error Codes

B

Macroassembler error messages appear as single letter flags in the first three character positions of a listing line. If a line of code contains one error, the error flag appears in character position three of that line. If there is a second error in a line, the second error flag appears in character position two. A third error in a line causes an error flag to appear in character position one. A fourth or subsequent error does not cause a flag to appear in the listing, but it is included in the total error count.

The macroassembler writes lines containing errors to the specified error file and as part of your assembly listing. You can usually suppress output of errors to the error file. If you suppress the program listing, the error listing is written to the error file. If you suppress both the assembly and the error listings, then the error listing is sent to the console. Table B.1 lists the error codes and their symbols.

The following pages provide explanations and examples of each error code. However, the examples do not show all possible causes of assembly errors.

Symbol	Error Code
A	Address error
B	Bad character, bad line
C	Macro error
D	Radix error
E	Equivalence error
F	Format error
G	Global reference error
K	Repetitive or conditional assembly error
L	Location counter error
M	Multiply-defined symbol error
N	Number error
O	Field overflow error or stack error
P	Phase error
Q	Questionable line error
R	Relocation error
U	Undefined symbol error
V	Variable label error
X	Text error
Z	Illegal use of external

Table B.1 Error codes and their symbols

A Addressing Error

Indicates an addressing error in a memory reference instruction.

Example 1

In this example a page zero relocatable instruction tries to reference a normal relocatable (.NREL) address.

```
                        .NREL   0
00003'000010     G:     10
                        .ZREL
A00000-044000           STA     0,G
```

Example 2

In this example an .NREL location tries to reference an address outside the program location counter's address range.

```
                                        ;(.-200< = disp < = .+177).
              .NREL 0
A00004'020000'   LDA   0,Y              ;Y is outside the
       004423'   .LOC  .+416            ;instruction's
00423' 000002 Y:2                       ;address range.
```

Bad Character B

Indicates an illegal character in some context. This type of error often leads to other errors.

Example

In this example the label contains an illegal character: %.

```
                        .NREL
  B00000'024023  .A%:   LDA         1,23
```

Macro Error C

Occurs under the following circumstances:

- You specify more than 63_{10} arguments.
- You attempted to continue the definition of a macro which was not the last one you defined.

The example illustrates the last circumstance.

Example

```
        .MACRO  A   ;This defines macro A.
            .
            .
    %
            .       ; other code
            .
        .MACRO  A   ;This is a legal continuation of macro
            .       ;A's definition.
            .
    %
        .MACRO  B   ;This is a new macro.
            .
            .
    %
            .       ;Other code
            .
     C  .MACRO  A   ;Since you have begun a new macro, B,
                    ;you cannot continue to define macro A.
```

D Radix Error

Occurs in three instances:

- The argument in a .RDX command is not within the range of 2-20.
- The argument in a .RDXO command is not within the range of 8-20.
- You use a digit not within the current input radix.

Example 1

```
D 000030       .RDX    4*6
```

Example 2

```
           000002            .RDX    2
   D00000  000013     B:      35
```

E Equivalence Error

Occurs when an equivalence line contains an undefined symbol to the right of the equal sign. This error may occur on pass 1 before the symbol has been defined or on pass 2 if the symbol was never defined.

Example

```
EE                    A=B      ;Pass 1 - B is unidentified.
EUU      000000       A=B      ;Pass 2 - B is unidentified.
```

F Format Error

Occurs when you try to use a format that is illegal for the current line. This error often occurs in conjunction with other errors.

Example 1

```
F00000 143000     ADD   2     ;Not enough arguments.
```

Example 2

```
F00000 041410    STA    0,10,3,SNC    ;Too many arguments and wrong
                                      ;operand for instruction type.
```

Example 3

```
         060512    .DUSR   C=DIAS
   F00000 060512    C       1              ;This symbol does not accept
                                           ;arguments.
```

Global Symbol Error G

Occurs when there is an error in the declaration of an external or entry symbol.

Example 1

In this example HH is never defined.

```
       GU     .ENT    HH
              .END
```

Example 2

In this example, AA is an entry in a program which declares AA external.

```
      G     AA:
                  .EXTN   AA
                  .END
```

Conditional Assembly Error K

Occurs when an .ENDC pseudo op is not preceded by a .DO or .IF*x* psuedo op.

Example

```
       000002    .DO      2
                 .
                 .
                 .
                 .ENDC
        K        .ENDC
```

L Location Error

Occurs when errors are detected in lines affecting the location counter.

If the expression in a .LOC or a .BLK statement evaluates to less than zero, then the macroassembler flags the line with an L. An L also flags such a line if the expression in a .LOC or a .BLK statement cannot be evaluated on the first pass of the assembler. In either case, the macroassembler ignores the .LOC or the .BLK and leaves the location counter unchanged.

Example 1

```
        L  177777    .LOC    -1
```

Example 2

```
      77711'000000   A:    0
     L      100012'         .BLK    .+100
```

Example 3

```
    LU  000000     .BLK   B       ;B undefined.
```

M Multiple Definition Error

Flags an illegal attempt to redefine a symbol. Within an assembly program, labels may be defined only once. Also, when IM is set, semipermanent symbols may not be redefined. Each time a multiply-defined symbol appears, the macroassembler flags it with an M.

Multiple occurrences of the .OB pseudo-op also cause M errors.

Example

```
        M00000'000000    ALPH:   0
        PM00001'000001   ALPH:   1
```

NOTE: *The second definition of ALPH is also flagged as a phase error (P) on the second pass. See **P**, the Phase Error entry.*

Number Error

N

Occurs if a single-precision integer is greater than or equal to 2^{16}, or if a double-precision number is greater than or equal to 2^{32}. The macroassembler truncates the former number to 16 bits, and the latter to 32 bits. This error also occurs if a floating-point number is too large.

Example

```
000012   .RDX    10
N000140  65539
000003
```

Field Overflow Error

O

Occurs in the following cases:

- You exceed the size of the stack.
- You did not issue a PUSH before issuing a .POP or a .TOP.
- You code an instruction operand too large to fit the corresponding field.
- You supply a value for a field which already contains a value.

When overflow occurs in an instruction field the field remains unchanged.

Example

```
000000 020775   LDA    5,.-3        ;AC field is too large.
       070400  .DIAC   R=DIA 2,0
000001 070400   R      1            ;AC field already has value.
000002 000000   .POP                ;Stack is empty.
000003 000000   .TOP                ;Stack is empty.
```

P Phase Error

Occurs during pass 2 when the macroassembler detects some unexpected difference from the source program scan on pass 1. For example, a label defined on the first pass which has a different value on the second pass causes a phase error. If you multiply-define a label, the M error flags each statement containing the symbol; the P error flags the second and later statements containing the symbol.

Example

```
 M00001 000000  B:   0
PM00002 000000  B:   1
```

Example

```
 00000 000001      .BLK      .PASS
P00001 000000   C:  0
```

Q Questionable Line

Occurs under the following conditions:

- You used a # or @ atom improperly.
- You used a ZREL displacement field in a memory reference instruction indexed by AC2 or AC3.
- You used a conditional skip instruction immediately before a two-word instruction.
- You wrote an illogical ALC instruction.

Example

```
 Q00002 113000          ADD    0,@2      ;Incorrect use of @.
                        .ZREL
 00000-000010    FLD:  .BLK    10
       000001          .NREL   1
Q00000!000000          LDA     0,FLD,2  ;FLD may get bound into
                                        ;a location greater than 177.
 00001!125015          MOV#    1,1,SNR  ;MOV instruction
Q00002!000000          ELDA    0,SYMB   ;precedes a two-word
                                        ;instruction.
Q00003!105010          MOV#    0,1      ;No-load bit is
                                        ;set here, but no
                                        ;skip specified.
```

Relocation Error

R

Indicates one of the following:

- The macroassembler cannot evaluate an expression to a legal relocation type (absolute, word-relocatable, or byte-relocatable).
- The expression mixes .ZREL and pure .NREL symbols.
- The expression mixes impure .NREL and .ZREL symbols.
- The expression mixes pure and impure .NREL symbols.

Example 1

```
       000000       .NREL 0
  00000'000010   E:  10
  00001'000000“      E+E          ;Contents are .NREL
                                  ;byte-reloctable.
 R00002'000000'      E+E+E        ;Contents not absolute,
                                  ;word-relocatable, or
                                  ;byte-relocatable.
```

Example 2

```
                     .ZREL
  00000-000000   A:   0
       000001       .NREL 1
  00000!000000        0
 R00001!000000!  B:   A+B         ;A and B are of different
                                  ;relocation types.
```

Undefined Symbol Error

U

Occurs during pass 2 when the macroassembler encounters a symbol whose value was not defined after pass 1. Occurs during pass 1 when a symbol definition depends on another symbol whose value is unknown.

NOTE: *A symbol does not have to be defined in a source module if it is already defined in MASM.PS.*

Example

```
    U00002 030000   LDA   2,B   ;B is as yet unknown.
```

Also see the example given in the entry for Equivalence Error (**E**).

V Variable Label Error

Occurs if anything other than a symbol follows the .GOTO or .ENDC pseudo ops.

Example 1

```
FV                    .GOTO    14
                      .GOTO    AUG
           [AUG]

    000001            .IFE     0
    000001            .IFE     0

FV                    .ENDC    14
                      .ENDC    HQF

           [HQF]
```

X Text Error

Occurs if the two expression delimiters < and > within a text string do not enclose a recognizable arithmetic or logical expression. You cannot use relational operators within text strings.

Example

```
                         .NREL   0
  00000'00001    X:     1
  00001'00002    Y:     2
 X00002'054476           .TXT    #<X+ Y>#      ;Spaces not allowed in
        000000                                ;expressions.
 X00004'000000           .TXT    #<+>#         ;Lacks operands.
 X00005'000076           .TXT    #<X=>Y>#      ;Illegal relational operator.
        054476                                ;Macroassembler
        000000                                ;sees < X = > as the
                                              ;expression, which is not a
                                              ;legal expression
                          .END
```

Cross Development

C

The three tables in this appendix describe switches you can use to bind programs under one operating system (the host system) to be run under another operating system (the target system).

For information on compatibility among operating systems or how to tailor a program to run under more than one system, refer to the system programmer's references for the host and target systems.

Switch	Function
/D	Includes the Debugger and places the symbol table in the program immediately above the impure area.
/DN	Includes the Debugger but not the symbol table.
/DS	Includes the Debugger and places the symbol table at the top of memory.
/MPOS	Generates an MP/OS-format program (and overlay) file. The binder includes MSL.LB unless you use the /N switch.
/MTOP=*addr*	Generates a program file for a system where the highest available memory address is *addr* (octal). The default value is 77776_8.
/N	Binder does not automatically search the system library file MSL.LB
/RS=*filename*	Generates a stand-alone system. The binder looks first for *filename*.RS, then for *filename*.
/SA	Generates a stand-alone system for RAM. Does not put the usual MP/OS header data in the program file, and assumes that the highest memory address (MTOP) is 077777_8.
/SP	Generates a stand-alone system for PROM. Loads the impure area at octal location 400 and the pure area at the top of memory. Also, does not put the usual MP/OS header data in the program file, and assumes that the highest memory address (MTOP) is 077777_8.
/ST	Builds an AOS format symbol table file.
/SYS	Generates an MP/OS system file.

Table C.1 Binder switches, MP/OS target

Switch	Function
/MPAOS	Builds an MP/AOS-format program (and overlay) file. Searches OSL.LB by default.
/N	Does not search the standard library file OSL.LB.

Table C.2 Binder switches, MP/AOS target

Switch	Function
/AOS	Generates an AOS-format program and overlay file. Includes MICREM.OB, MMSL.LB and URT.LB.
/N	Does not automatically include MICREM.OB or search the library file MMSL.LB. The library file URT.LB is searched whether or not this switch is used.

Table C.3 Binder switches, AOS target

Index

DG OFFICES

NORTH AMERICAN OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: Anaheim, El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Riverside, Sacramento, San Diego, San Francisco, Santa Barbara, Sunnyvale, Van Nuys
Colorado: Colorado Springs, Denver
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Bettendorf, Des Moines
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland, Westbrook
Maryland: Baltimore
Massachusetts: Cambridge, Framingham, Southboro, Waltham, Wellesley, Westboro, West Springfield, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Reno
New Hampshire: Bedford, Portsmouth
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Buffalo, Lake Success, Latham, Liverpool, Melville, New York City, Rochester, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh, Research Triangle Park
Ohio: Brooklyn Heights, Cincinnati, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego
Pennsylvania: Blue Bell, Lancaster, Philadelphia, Pittsburgh
Rhode Island: Providence
South Carolina: Columbia
Tennessee: Knoxville, Memphis, Nashville
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston, San Antonio
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Bellevue, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Grand Chute, Madison

INTERNATIONAL OFFICES

Argentina: Buenos Aires
Australia: Adelaide, Brisbane, Hobart, Melbourne, Newcastle, Perth, Sydney
Austria: Vienna
Belgium: Brussels
Bolivia: La Paz
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
Chile: Santiago
Columbia: Bogata
Costa Rica: San Jose
Denmark: Copenhagen
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
France: Le Plessis-Robinson, Lille, Lyon, Nantes, Paris, Saint Denis, Strasbourg
Guatemala: Guatemala City
Hong Kong
India: Bombay
Indonesia: Jakarta, Pusat
Ireland: Dublin
Israel: Tel Aviv
Italy: Bologna, Florence, Milan, Padua, Rome, Tourin
Japan: Fukuoka, Hiroshima, Nagoya, Osaka, Tokyo, Tsukuba
Jordan: Amman
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Monterrey
Morocco: Casablanca
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jeddah, Riyadh
Singapore
South Africa: Cape Town, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid
Sweden: Gothenburg, Malmo, Stockholm
Switzerland: Lausanne, Zurich
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
United Kingdom: Birmingham, Bristol, Glasgow, Hounslow, London, Manchester
Uruguay: Montevideo
USSR: Espoo
Venezuela: Maracaibo
West Germany: Dusseldorf, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart

Data General

Ordering Technical Publications

TIPS is the Technical Information and Publications Service—a new support system for DGC customers that makes ordering technical manuals simple and fast. Simple, because TIPS is a central supplier of literature about DGC products. And fast, because TIPS specializes in handling publications.

TIPS was designed by DG's Educational Services people to follow through on your order as soon as it's received. To offer discounts on bulk orders. To let you choose the method of shipment you prefer. And to deliver within a schedule you can live with.

How to Get in Touch with TIPS

Contact your local DGC education center for brochures, prices, and order forms. Or get in touch with a TIPS administrator directly by calling (617) 366-8911, extension 4086, or writing to

Data General Corporation
Attn: Educational Services, TIPS Administrator
MS F019
4400 Computer Drive
Westborough, MA 01580

TIPS. For the technical manuals you need, when you need them.

DGC Education Centers

Boston Education Center
Route 9
Southboro, Massachusetts 01772
(617) 485-7270

Washington, D.C. Education Center
7927 Jones Branch Drive, Suite 200
McLean, Virginia 22102
(703) 827-9666

Atlanta Education Center
6855 Jimmy Carter Boulevard, Suite 1790
Norcross, Georgia 30071
(404) 448-9224

Los Angeles Education Center
5250 West Century Boulevard
Los Angeles, California 90045
(213) 670-4011

Chicago Education Center
703 West Algonquin Road
Arlington Heights, Illinois 60005
(312) 364-3045

Technical Products Publications
Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title: ________________________

Document No. 069-400210-00

Yes	No			
□	□	Is this manual easy to read?	○ You (can, cannot) find things easily. ○ Language (is, is not) appropriate. ○ Technical terms (are, are not) defined as needed.	○ Other:
		In what ways do you find this manual useful?	○ Learning to use the equipment ○ As a reference ○ As an introduction to the product	○ To instruct a class. ○ Other:
□	□	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:	
□	□	Does the manual tell you all you need to know? What additional information would you like?		
□	□	Is the information accurate? (If not please specify with page number and paragraph.)		

Name: ____________________ Title: ________________

Company: ____________________ Division: ________________

Address: ____________________ City: ________________

State: ________ Zip: ________ Telephone: ____________ Date: ________

FOLD *FOLD*

TAPE *TAPE*

FOLD *FOLD*

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

DataGeneral
ATTN: Technical Products Publications (C-138)
4400 Computer Drive
Westboro, MA 01581

CUT ALONG DOTTED LINE

Data General

users group

Installation Membership Form

Name ______________________ Position ______________________ Date __________

Company, Organization or School ____

Address ______________________ City ______________ State ______ Zip ________

Telephone: Area Code __________ No. ______________ Ext. ______________

1. Account Category

□ OEM
□ End User
□ System House
□ Government
□ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) ________		
________		
________		

3. Software

□ AOS
□ DOS
□ MP/OS
□ RDOS
□ Other

Specify ______________
____

4. Languages

□ Algol
□ DG/L
□ Cobol
□ PASCAL
□ Business BASIC
□ BASIC
□ Assembler
□ Fortran
□ RPG II
□ PL/1
□ Other

Specify ______________
____

5. Mode of Operation

□ Batch (Central)
□ Batch (Via RJE)
□ On-Line Interactive

6. Communications

□ HASP
□ RJE80
□ RCX 70
□ CAM
□ XODIAC
□ Other

Specify ______________
____

7. Application Description

○ ______________
____
____
____
____
____

8. Purchase

From whom was your machine(s) purchased?

□ **Data General Corp.**
□ Other
Specify ______________
____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ ______________
____

Data General

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General
ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

Data General Corporation, Westboro, Massachusetts 01580