COP8™

Assembler/Linker/Librarian
User's Manual

Literature Number 620896-003
November 1996

National Semiconductor

READER'S COMMENT FORM

In the interest of improving our documentation, National Semiconductor invites your comments on this manual.

Please restrict your comments to the documentation. Technical Support may be contacted at:

(800) 272-9959 (U.S.)	U.S., Canada and South America
49 0-180-532 78 32 (Germany)	Europe and Israel
852 737-1800 (Hong Kong)	South East Asia, Australia, New Zealand and India
81-043-299-2309	Japan
(55-11) 212-5066	Brazil

Please rate this document according to the following categories. Include your comments below.

	EXCELLENT	GOOD	ADEQUATE	FAIR	POOR
Readability (style)	❒	❒	❒	❒	❒
Technical Accuracy	❒	❒	❒	❒	❒
Fulfills Needs	❒	❒	❒	❒	❒
Organization	❒	❒	❒	❒	❒
Presentation (format)	❒	❒	❒	❒	❒
Depth of Coverage	❒	❒	❒	❒	❒
Overall Quality	❒	❒	❒	❒	❒

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

Do you require a response? ❒Yes ❒No PHONE _____

Comments: _____

_____

_____

_____

_____

Assembler/Linker/Librarian User's Manual

FOLD, STAPLE, AND MAIL November 1996

Please return this to:

In U.S.A.
(Including Canada and South America):

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Attn: Technical Publications, M/S E215

In South East Asia
(Including Australia, New Zealand and India):

National Semiconductor H.K. Ltd.
15th Floor, Straight Block
Ocean Centre
5 Canton Road
Tsim Sha Tsui East
Kowloon, Hong Kong
Attn: Business Center Marketing

In Europe
(Including Israel):

National Semiconductor GmbH
Livry-Garge-Stra. #10
D-82256 Fuerstenfeldbruck
Germany
Attn: Business Center Marketing

In Japan

National Semiconductor Japan Ltd.
Sumitomo Chemical Engineering Building, 7F
1-7-1, Naskase, Mihama-Ku
Chiba-City, Ciba Prefecture 261
Japan
Attn: Business Center Marketing

(fold)

Place
Stamp
Here

National Semiconductor

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	09/92	First Release. COP800™ Assembler/Linker/Librarian User's Manual Publication Number 424421632-001A
B	08/93	Cleaned up and edited entire manual.
C	10/94	Added new parts to Appendix B.
-001	05/95	Added new parts to Appendix B.
-002	01/96	Added new parts to Appendix B.
-003	11/96	Added new parts to Appendix B.

PREFACE

This manual provides information on system programs that support the development of COP8™ microcontroller applications. It is written for application developers who are using a chip in the COP8 microcontroller family in their embedded system.

Chapter 1 contains an overview of the COP8 development programs, ASMCOP, LNCOP, and LIBCOP. It also describes the documentation conventions used in this manual.

Chapter 2 gives a detailed description of the COP8 cross-assembler, ASMCOP, including inputs, instruction formats, directives, controls, and outputs.

Chapter 3 describes the COP8 Cross-Linker, LNCOP. Chapter 4 describes the COP8 Cross-Librarian, LIBCOP. Chapters 5, 6, and 7 describe four utility programs, DUMPCOFF, PROMCOP, HEXLM, and LMHEX. Appendix A contains the ASCII character set with its hexadecimal equivalent codes. Appendix B describes the supported COP8 chips and default memory ranges.

ASMCOP is a cross-assembler for the National Semiconductor COP8 microcontrollers. This manual describes the instruction formats, features, and directives of the ASMCOP assembler. For a description of the instructions, see the *COP8 Basic Family User's Manual*, Literature Number 620895 and the *COP8 Feature Family User's Manual*, Literature Number 620897.

LNCOP is a cross-linker that links object files created by *ASMCOP*, to create an absolute object file that can be down-loaded to a COP8 emulator.

LIBCOP is a cross-librarian that reads object modules produced by *ASMCOP* and combines them into one file called a library for later use in other COP8 programs.

DUMPCOFF is a utility program used to display the COFF object files (generated by LNCOP) in a readable form.

PROMCOP is a utility program used to convert the COFF object file into one or more output files for the purpose of burning PROMS.

HEXLM and LMHEX utilities convert -LNCOP-hex files to National LM format, or LM format to Intel-hex.

This manual assumes you are already familiar with the host operating system. For example, you need to know how files are named and used under the operating system. You also need to be able to use an editor to produce symbolic files.

The information contained in this manual is for reference only and is subject to change without notice.

COP8 is a trademark of National Semiconductor Corporation.
MS-DOS is a trademark of Microsoft Corporation.

CONTENTS

Figures

Tables

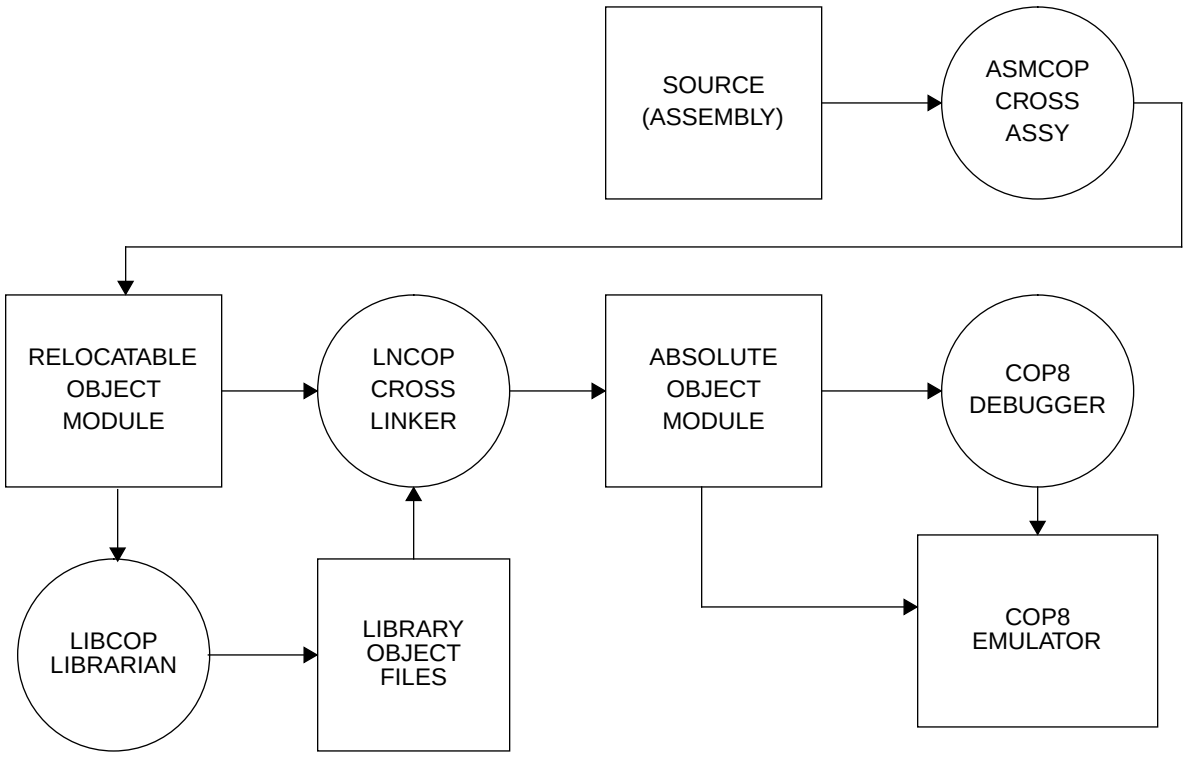
Chapter 1

INTRODUCTION

1.1 OVERVIEW

This manual provides information on system programs that support the development of COP8™ microcontroller applications on COP8 development systems. This manual does not describe all necessary software; for example, it does not include information on a text file editor, but such a program is necessary to develop input files for the COP8 cross-assembler, ASMCOP.

The COP8 cross-assembler, ASMCOP, translates assembly source files into relocatable object modules that contain instructions in binary machine language. These object modules are linked using the COP8 cross-linker, LNCOP, to generate an absolute object module; the absolute object module can be loaded into a COP8 emulator. The ASMCOP object modules can also be combined into a library by using the COP8 cross-librarian, LIBCOP. Figure 1-1 illustrates the software development process.



ASM-01

Figure 1-1 COP8 Software Development Process

The following utilities are also provided:

- DUMPCOFF is a utility program used to display COFF object files (generated by LNCOP) in a readable form.
- PROMCOP is a utility program used to convert a COFF object file into an Intel hex output file for the purpose of burning PROMs.
- HEXLM and LMHEX utilities convert Intel-hex files to NSC LM format, or LM format to Intel-hex.

1.2 COP8 CROSS-ASSEMBLER (ASMCOP)

The COP8 cross-assembler (ASMCOP) is a cross-assembler for the National Semiconductor COP8 family of microcontrollers. ASMCOP translates symbolic input files into object modules and generates an output listing of the source statements, machine code, memory locations, error messages, and other information useful in debugging and verifying programs.

Chapter 2 describes the format, features, and directives of the COP8 cross-assembler. See the *COP8 Basic Family User's Manual* and the *COP888 Feature Family User's Manual* for a description of the instructions.

1.3 COP8 CROSS-LINKER (LNCOP)

The COP8 cross-linker (LNCOP) links object files generated by *ASMCOP*. The result is an absolute load module in one of various formats, such as Intel-hex, National "lm" format, or COFF (Common Object File Format). The COFF file is accepted by COP8 debuggers.

1.4 COP8 CROSS-LIBRARIAN (LIBCOP)

The COP8 cross-librarian (LIBCOP) reads object modules produced by ASMCOP and combines them into one file called a library. When linking other programs, the linker can then search the library for modules defining any undefined external symbols.

1.5 COP8 COFF DISPLAY UTILITY (DUMPCOFF)

The COP8 COFF display utility (DUMPCOFF) displays the COFF object file in readable form. Code, symbols, sections, and line information can be displayed optionally.

1.6 COP8 PROM UTILITY (PROMCOP)

The COP8 PROM utility, PROMCOP, converts the COFF object file into an Intel hex file for the purpose of burning PROMs.

1.7 HEX LM UTILITIES (HEXLM, LMHEX)

These utilities convert Intel-hex into NSC LM format or LM format into Intel-hex.

1.8 DOCUMENTATION CONVENTIONS

The follow documentation conventions are used in the text, syntax descriptions, and examples when describing commands and parameters.

1.8.1 General Conventions

Non-printing characters are indicated by enclosing a name for the character in angle brackets <>. For example, <CR> indicates the RETURN key, <CTRL>B indicates the character input by simultaneously holding down the control key(Ctrl) and pressing the "B" key.

1.8.2 Conventions in Syntax Descriptions

The following conventions are used in syntax descriptions:

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ } Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign "|."

[ ] Large brackets enclose optional items.

| Logical OR sign separates items of which one and only one may be used.

... Three consecutive periods indicate optional repetition of the preceding item. If a group of items can be repeated, the group is enclosed in large parentheses "( )."

,,, Three consecutive commas indicate optional repetition of the preceding item. Items must be separated by commas. If a group of items can be repeated, the group is enclosed in large parentheses "**( )**."

( ) Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces that must be entered are smaller than the symbols used to describe the syntax. (Compare user-entered [ ], with **[ ]** which show optional items.)

1.8.3 Example Conventions

In interactive examples where both user input and system responses are shown, the machine output is shown in `typewriter` type; user-entered input is shown in **boldface** type.

Chapter 2

CROSS-ASSEMBLER (ASMCOP)

2.1 INTRODUCTION

This chapter describes the inputs, format, directives, and outputs of the ASMCOP cross-assembler. See the *COP8 Basic Family User's Manual* and the *COP8 Feature Family User's Manual* for lists of instruction mnemonics and functions.

ASMCOP translates source files into object modules that contain instructions in binary machine language. These object modules are linked with LNCOP to generate an absolute object module; the absolute object module can be loaded into the COP8 emulator for program debugging and emulation. The ASMCOP object modules may also be combined into a library for later use in other programs by using LIBCOP.

To aid the programmer, the assembler optionally generates an output listing of the source statements, machine code, memory locations, error messages, and other information useful in debugging and verifying programs.

The assembler has a number of directives that control the operation of the assembler. For example, the .TITLE directive controls the identifying title that the assembler prints on the pages of the output listings; the .BYTE directive instructs the assembler to generate data in memory bytes.

2.2 INVOCATION AND OPERATION

This section discusses the invocation of ASMCOP, assembler options, search order for include and help files, and the default filenames and extensions. See the release letter for installation instructions.

2.2.1 Invocation

For the MS-DOS operating system, the invocation line is as follows:

ASMCOP (gives help information)

ASMCOP [*options***]** *asmfile***[**,*asmfile***[**,...**]] [***options***]**

where: *asmfile* is the name of a program to assemble. Multiple files may be assembled by joining them with a "," for MS-DOS systems. Multiple source files are designed to allow the user to include standard definition files without having to specify them in each program. The default extension is .asm.

options is a list of assembler options. If no options are specified the assembler defaults are used. Section 2.2.2 describes the option syntax.

NOTE: MS-DOS supports *@cmdfile*, where *cmdfile* contains additional invocation line source filenames and/or options. This file has a default extension of .cmd. For example, if the file *a.cmd* contains:

```
test /a
/o
```

and file *b.cmd* contains

```
/l=file
```

then the command

ASMCOP @a /e @b

is equivalent to

ASMCOP test /a /o /e /l=file

Any error detected on the invocation line causes the assembler to stop execution and display the part in error with an appropriate message.

The following are example invocation lines for MS-DOS systems:

```
ASMCOP TEST /L=CON /NOOBJ/NOTABLE
```

The first command line assembles the file *test.asm* and outputs a listing to the console. No object module or symbol table is produced.

```
ASMCOP JACK.SRC /CROSS /L
```

The second command line assembles the file *jack.src* in the current directory and outputs a listing with cross-reference to the default file *jack.lis*. In addition, the object module is written to the file *jack.obj*.

```
ASMCOP \DEMO\SAMPLE /D COUNT=6 /PW=120
```

The next example assembles the file *sample.asm* which resides in the directory *demo*. It produces *sample.obj*, which is in the current directory. By default, it produces *sample.lis* (errors only) in the current directory. The symbol COUNT is defined and given the value 6. Also, the page width is set to 120 characters.

```
ASMCOP \DEMO\SAMPLE /L=\DEMO\
```

The fourth example assembles the file *sample.asm* which resides in the directory *demo*. It produces *sample.obj*, which is in the current directory. It produces *sample.lis* in the directory *demo*.

```
ASMCOP MATH.DEF,MATH
```

The last example shows two files being assembled as one assembly file. The program first assembles *math.def*. When the end-of-file is reached, it then continues the assembly with *math.asm*. By default, file *math.obj* is produced.

2.2.2 Assembler Options

An assembler invocation line option is an assembler control that is specified in a manner consistent with the operating system.

The invocation line options for an MS-DOS operating system start with a slash(/), which may be preceded or followed by a space. Assembler options are not case sensitive and may be abbreviated to the minimum number of characters as specified in the control descriptions. For example:

/CROSSREF / CNDL

See Section 2.10 for a description of all the assembler controls.

2.2.3 Default Filenames and Extensions

For those options that require a filename, the name may include a directory path. If it consists of just a directory path, the default filename is used with that directory. The default filename is the name of the last source file specified with any extension removed.

Default extensions depend on the operating system and how the file is specified on the invocation line. For the MS-DOS operating system, a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists only of a directory, it should always be terminated by a "\" on the MS-DOS operating system. If not, it is treated as a filename. Thus, /L=txt\ outputs the listing to file *txt\cat.lis* (assuming *cat.asm* is input file). /L=txt outputs the listing to *txt.lis*.

2.2.4 Include File Search Order

When searching for a file specified on the .INCLD directive, directories are searched in the following order:

1. Current directory
2. Directories specified by the /I option
3. Default directories
    a. Directory specified by environment variable ASMCOP, if it exists
    b. Directory specified by environment variable COP, if it exists
    c. Directory \COP

If the /X option is specified, only the directories specified by the /I option are used.

Any filename that specifies an explicit directory is checked for only in that directory. No other directories are searched.

2.2.5 Help File Search Order

When searching for the help file *ASMCOP.hlp*, directories are searched in the following order:

1. Current directory
2. Default directories (as noted in Section 2.2.4)

2.2.6 Temporary File Directory

Temporary files are generated in the current directory, unless the environment variable TMP specifies another directory, e.g., DOS command: set TMP=d:\. It is recommended to specify TMP as a directory on a RAM drive, of size 256K.

Recommendations concerning the use of a RAM drive are contained in the ASMREAD.ME file in your installation directory.

2.2.7 Error Level Return

If no errors occur, an error level of zero is returned. If errors occur, a nonzero error level is returned (warnings are not considered errors).

2.3 ASSEMBLY LANGUAGE ELEMENTS

This section discusses the format used to write the following assembler statements.

- Character set
- Location counter
- Symbol and labels
- Expressions
- The four fields of assembly language statements:

— label field

— operation field

— operand field

— comment field

The statement fields appear in the following order:

label field operation field operand field comment field

Since the assembler accepts free-form statements, the user may disregard specific field boundaries provided that the appropriate delimiters for each field are used (see individual field descriptions). However, for clarity and readability, the use of field boundaries is highly recommended.

2.3.1 Character Set

Each statement is written using the following characters:

Letters — A through Z (a through z)

Numbers — 0 through 9

Special Characters — ! $ % ' ( ) * + , - . / ; : < = > & # ? _ b

NOTE: Upper- and lowercase are distinct, and **b** indicates a blank.

2.3.2 Location Counter

Each program section has a separate location counter, and the counter is relative to the start of that section. The assembler uses the location counter in determining where in the current program section the current statement goes. For example, if the location counter has the value X'24 (i.e., 24 hex) and the assembler encounters a 1-byte instruction, the assembler assigns the instruction machine code to section address X'24 and increments the location counter by one, since the statement requires one byte of memory. If the program section is relocatable, the linker assigns an absolute address to the instruction.

The location counter symbol is a single dot (.). If the location counter symbol is used on the right side of an assignment, the left symbol is assigned the current value of the location counter. If the location counter symbol is on the left side of an assignment, the value of the location counter is changed to the value of the right side of the assignment statement.

2.3.3 Symbol and Label Construction

The following are the rules for symbol construction:

1. The first character of a symbol must be either a letter, a question mark (?), an underline (_), a dollar sign ($), or a period (.).
2. All other characters in the symbol may be any alphanumeric character, dollar sign ($), question mark (?), or underline (_).

   Examples: LOOPloop$_? legal symbols

   ?1

   _1

   $1

3. The first 64 characters are used by the assembler (the SIZESYMBOL control may reduce the number).
4. Symbols that start with a dollar sign are local symbols and are only defined in a local region. (See Section 2.9.23.)
5. Symbols are case sensitive.

Symbols and labels are used to provide a convenient name for values and addresses. The rules for constructing symbol names and the rules for constructing label names are the same; only use distinguishes a symbol from a label. Section 2.3.6 describes how values are assigned to labels. Sections 2.5 and 2.9.33 describe how values are assigned to symbols.

2.3.4 Operand Expression Evaluation

The expression evaluator in the assembler evaluates an expression in the operand field of a source program. The expressions are composed of combinations of terms and operators.

Terms

Terms in an expression are:

- numbers in decimal, hexadecimal, octal, or binary
- string constants
- labels and symbols
- location counter symbol

Each term isdescribed by four attributes: value, relocation type, memory type and size. The relocation type is either absolute or relocatable. An absolute term is one in which the value is completely known during assembly. A relocatable term is defined as a label within a relocatable section (see Section 2.9.32), a symbol equated to another relocatable expression, or a symbol declared with the .EXTRN directive. The value of a relocatable term is the offset of the label from the start of the section or the external value. Both of these values must be determined by the linker.

The memory type of a term indicates whether the term represents a BASE, RAM, EERAM, REG, SEG, SEGB or ROM address. This is also specified by use of the .SECT or .EXTRN directives. In addition, the memory type of a term may be null in the case of an absolute term. The size attribute of a term is null, byte or word. A term has the byte attribute if it is the label on a .DB, .DSB, or .FB directive, or it is specified as byte on a .EXTRN or .SET directive or = (assignment). A term has the word attribute if it is the label on a .DW, .DSW, or .FW directive, or it is specified as word on the .EXTRN or .SET directive or = (assignment).

An expression has the same attributes as a term. These attributes are taken from the terms that comprise the expression, and only certain combinations of terms are valid in an expression. The relocation type of an expression is derived from its terms as follows (aterm is absolute, rterm is relocatable, op represents any operator, cop represents any conditional operator, e.g., >=):

rterm = rterm + aterm

rterm = rterm - aterm

rterm = aterm + rterm

aterm = aterm op aterm

aterm = rterm - rterm

aterm = rterm cop rterm

In the last two cases, the terms must be relocatable within the same section. Any other expression is considered to have a complex relocation type and must be resolved by the linker.

The memory or size type of an expression is derived from its terms in a manner similar to the relocation type. If type is an expression with memory or size type and number is an absolute (non-label) value, then the following rules apply:

type = type + number

type = type - number

type = number + type

Any other combinations of memory or size types are considered null. Also, a complex expression is considered to have a null size type.

The notation for the various types of terms is detailed later in this section.

Operators

Operators in an expression are:

- arithmetic operators
- logical operators

- relational operators
- upper- and lower-byte extraction operators
- untype operator

The *arithmetic* operators are the usual **+**, **-**, *****, **/**, MOD, SHL, SHR, ROL, and ROR. The *logical* operators are NOT, AND, OR and XOR. The available *relational* operators are EQ, NE, GT, LT, GE and LE. The ampersand (**&**) is the untype operator. The upper- and lower-byte *extraction* operators are HIGH and LOW. Table 2-1 lists the operators, function and whether the operator is unary or binary. Some operators have optional syntax for compatibility with older assemblers. Table 2-2 lists the precedence order for the evaluation of the operators; a higher precedence operator is evaluated before a lower precedence operator.

Parentheses are permitted in expressions. Parentheses in expressions override the normal order of evaluation; the expression(s) within the parentheses are evaluated before the outer expressions.

The assembler recognizes twelve types of terms. They are listed with their notations in the following sections.

Decimal Constant Terms

A decimal constant term is a decimal number that optionally begins with “D'” or “d'.” Leading zero is not permitted for decimal, except for simple case of constant 0.

Examples: 3, 234, -10, D'3.

Hexadecimal Constant Terms

A hexadecimal constant term is a hexadecimal number that starts with “X'” or “x'” or “H'” or “h'” or “0X” or “0x” or 0. An optional “H” or “h” is permitted at the end.

Examples: X'23A, H'23A, 0x23A, 023A, 023AH.

Octal Constant Terms

An octal constant term is an octal number that starts with “O'” or “o'” or “Q'” or “q'.”

Examples: O'27, Q'27.

Binary Constant Terms

A binary constant term is a binary number that starts with “B'” or “b'.”

Examples: B'011, B'0111011.

Table 2-1 Arithmetic, Logical, and Relational Operators

Operator	Optional	Function	Type
+		Addition	Unary or Binary
-		Subtraction	Unary or Binary
*		Multiplication	Binary
/		Division	Binary
MOD		Modulo	Binary
SHL		Shift Left	Binary
SHR		Shift Right	Binary
ROL		Rotate Left	Binary
ROR		Rotate Right	Binary
NOT	%	Logical NOT	Unary
AND	&	Logical AND	Binary
OR	!	Logical OR	Binary
XOR		Logical XOR	Binary
LT	<	“Less Than”	Binary
EQ	=	“Equal To”	Binary
GT	>	“Greater Than”	Binary
LE	<=	“Less Than or Equal To”	Binary
GE	>=	“Greater Than or Equal To”	Binary
NE	<>	“Not Equal To”	Binary
&		Untype	Unary
LOW	L	Lower 8 bits	Unary
HIGH	H	High 8 bits	Unary
B_SECT		Beginning of section	Unary
E_SECT		End of section	Unary

Table 2-2 Operator Precedence Value

Operator	Precedence Value
)	0 (lowest)
OR, !	1
XOR	1
AND, &	2
NOT, %	3
LT, <	4
GT, >	4
EQ, =	4
NE, <>	4
LE, <=	4
GE, >=	4
+	5
-	5
/	6
*	6
MOD	6
SHL	6
SHR	6
ROL	6
ROR	6
LOW, L	7
HIGH, H	7
(	8
UNARY -	9 (highest)
UNARY +	9

String Constant Terms

A string constant term is a one or two character string enclosed in single quotation marks.

Examples: 'Z', '$', '23', '', '"', '"'.

The null string '' is evaluated as 0. Within a string, single quotation marks are indicated by two quotation marks. The string '"' is the single quotation mark string, and '"' is the double quotation mark string.

String constants are represented internally by the appropriate 8-bit ASCII code (the most significant bit is zero).

Examples:
''	is replaced by 0
'A'	is replaced by 041
'AB'	is replaced by 04142

The following escape codes may be used in a string constant to represent the value shown:

\a	0x7	bell
\b	0x8	backspace
\f	0xC	formfeed
\n	0xA	linefeed
\r	0xD	carriage return
\t	0x9	horizontal tab
\v	0xB	vertical tab
\0	0	null
'	0x27	quote
\"	0x22	double quote
\\	0x5C	reverse slash

Any lowercase character may also be specified as uppercase (e.g., \b and \B are the same).

Label Terms

A label term is described in Section 2.3.6 under the label field description.

Symbol Terms

A symbol term consists of a single symbol. The symbol has been given a value by either an assignment statement (Section 2.5) or by the .SET directive (Section 2.9.33).

Location Counter Terms

The location counter term is a single dot (.). The dot represents the location counter and, if it appears within an expression, it is replaced by the current value of the location counter.

Example: JP .

Lower Half and Upper Half Terms

A lower-half term is represented by LOW *expression*. An upper-half term is represented by HIGH *expression*. When the assembler encounters these terms in an expression, it replaces it with either the lower or the upper eight bits of the value of the expression.

Examples: HIGH X'172F is replaced by X'17

LOW X'172F is replaced by X'2F

Size Type

The size type may be removed from a term with the **&** operator. For example, if

BYT = 7:BYTE

then &BYT has the value 7 and its size type is null.

B_SECT and E_SECT Operators

The B_SECT and E_SECT operators are used to obtain the beginning address of a relocatable section and the ending address of a relocatable section plus one. This section must be declared in this module, if it exists externally. (See the .SECT directive Section 2.9.32.) Each has a format of

B_SECT (*section name*)

E_SECT (*section name*)

Example: LD A,#E_SECT(ONE) - B_SECT(ONE)

loads register A with the length of section ONE.

Numbers

Numbers are represented internally in the assembler in 16-bit two's complement notation. The range for numbers in this representation is -32768 (X'8000) to +32767 (X'7FFF) for signed numbers and 0 to 65535 for unsigned numbers.

Expressions

An expression may consist of a single term, as shown in the following:

Examples:
```
5
X'3C
'Q'
SUB
.
HIGH(X'3CF)
LOW(SUB)
```

Alternatively, an expression may consist of two or more terms combined using the operators shown in Table 2-1.

Examples:
```
36 + SUB
X'3F0-10
X'7F AND 'Q'
3*5 OR XYZ
(NOT SUB)/2
```

NOTE: All expression evaluations treat the terms as unsigned numbers; for example, -1 is treated as value X'FFFF.

The magnitude of the expression must be compatible with the memory storage available for the expression. For example, if the expression is to be stored in an 8-bit memory word, then the value of the expression must not exceed X'FF.

2.3.5 Addressing

This section shows the syntax of the various instruction addressing modes. The following labels are used in the examples in this section:

```
.SECT example, RAM

WRD1: .DSW 1

BYT1: .DSB 3

NUM = 6
```

NOTE: WRD1 has the word attribute and BYT1 has the byte attribute; NUM is of type null. These attributes only have meaning in a debugger; the assembler treats word, byte, and null attributes the same.

Direct Addressing

A direct operand is specified.

```
Example:    LD   A,WRD1+2      ; load contents of byte at address WRD1+2
            LD   A,BYT1+1      ; load contents of byte at address BYT1+1
```

Immediate Addressing

An immediate operand is specified with a #. Values must be in range –256 to 255; –256 is treated as 0, –1 as 255.

```
Example:    LD   A,#BYT1+2     ; load A immediate
            ADD  A,#1          ; add immediate to A
```

Register Indirect Addressing

A register indirect operand is specified by [*reg*], where *reg* is X or B.

```
Example:    LD   A,[B]         ; B indirect
            LD   A,[X]         ; X indirect
```

Register Indirect Addressing, Auto Increment/Decrement

A register indirect operand with auto increment is specified by [B+]; [X+]. Auto decrement uses – instead of +.

```
Example:    LD   A,[X+]        ; reg indirect, auto increment
            LD   A,[B-]        ; reg indirect, auto decrement
```

Branch Addressing

The operand of one of the branch type instructions (JP, JMP, JMPL, JSR or JSRL) must be a null type expression, and the operand is normally defined in a section that has the ROM attribute. Assume you have the following program fragment:

```
        .SECT   ONE,RAM
DAT:    .DSB    1
        .SECT   TWO,ROM
BYT:    .DB     2
LBL:    LD      A,DAT
```

Then:

```
JMP     LBL     ; is valid instruction
JMP     BYT     ; is an error, operand type must be null
JMP     DAT     ; is a error, operand is RAM type
JMP     &BYT    ; valid now, byte type removed by &
```

Operand Size

The assembler normally generates the minimal instruction size possible for each instruction. There may be cases in which you want the instruction to be the maximum size for ease of debugging. In these cases, the > operator is provided to force the maximum size for each operand. This operator can appear only at the start of the operand.

```
Example:    JP    >label        ; 3 bytes
            LD    >B,#0         ; 3 bytes
```

2.3.6 Label Field

The label field is optional and has two uses. Most frequently, the field contains a symbol used to identify a statement referenced by other statements. Symbols used in this way are labels. Alternatively, the field contains a symbol whose value is set by the assignment operator. For more information on the second use, see Section 2.5.

When the assembler encounters a label, it assigns the current value of the location counter to the label. A colon (:) is used to delimit (terminate) each label in the label field. (When the label is used in the operand field of an instruction, the colon is omitted.)

The rules for label name construction are the same as the rules as for any symbol. Refer to Section 2.3.3.

A label referencing an instruction need not be on the same line as the instruction. This allows the programmer, when writing source code, to devote a separate line with comments to labels, providing clearer documentation of the program and allowing for easier editing of the source code.

```
Example:            .=01000H        ; set location counter to 01000H
                                    ; label SUB is assigned 01000H
                SUB:
                    NOP
                    .
                    .
                    .
                    .
```

CAUTION

Read the following before using labels on a blank line.

The assembler always processes labels on a line after it processes any following fields. Therefore, when a label appears on the same line as an assignment statement which alters the location counter, the label is assigned the location counter value after the location counter is altered. A label on a preceding line in this case is not the same value.

```
Example:            .=01001H        ; set location counter to 01001H
                SUB:                ; label SUB is 01001H

                SUB4: .=01010       ; label SUB4 is assigned 01010H
```

2.3.7 Operation Field

The operation field contains an identifier that indicates what type of statement is on the line. The identifier may be an instruction mnemonic or an assembler directive. The operation field is required, except in lines that consist of only a label and/or comment.

In an instruction statement, the operation field contains the mnemonic name of the desired instruction.

```
Example:        label  operation
                SUB:   NOP
```

In a directive statement, the operation field contains a period (.) immediately followed by the name of the desired directive.

```
Example:        .END
```

See Section 2.9 for the valid directive names.

In an assignment statement, the operation field contains an equal sign (=). See Section 2.5.

One or more blanks terminate the operation field.

2.3.8 Operand Field

The operand field contains entries that identify data to be acted upon by the operation defined in the operation field, e.g., source or target address for the movement of data, or immediate data for storage or adding to another value.

Some statements do not require use of the operand field. For those that do, the operand field usually consists of one or two expressions (the second expression is separated from the first by a comma.) Expressions can be composed of numbers, string constants, labels and symbols combined with arithmetic, logical and relational operators. Section 2.3.4 describes in detail the types of terms used in expressions, the permitted operators, and the order of evaluation used for expressions with more than one operator.

2.3.9 Comment Field

Comments are optional descriptive notes printed on the assembler output listing for programmer reference and documentation. Comments should be included throughout the source program to explain subroutine linkages, data formats, algorithms used, formats of inputs processed, and so forth. A comment may follow a statement on the same line, or the comment may be entered on one or more separate statement lines. The comment has no effect on the assembled object module file.

The following conventions apply to comments:

- A comment must be preceded by a semicolon (;).
- All ASCII characters, including blanks, may be used in comments.

2.4 ASSEMBLY PROCESS

The assembler performs its functions by reading through the assembly language statements sequentially from top to bottom, generating the machine code and a program listing as it proceeds. Since it reads statements sequentially, a special problem occurs in most assemblers which must be overcome. If the assembler encounters the statement

```
JMP CLEAR
```

before it encounters the label CLEAR, it is unable to generate machine code for that instruction. This problem is solved by making the assembler perform two “passes” through the assembly language statements.

Pass 1 of the assembler does not generate an object module or a listing. Its purpose is to assign values to labels and symbols. The assembler assigns labels by using an internal counter called the location counter. Each program section has a separate location counter,

which indicates where the current instruction/data is in relation to the start of the section. Each time the assembler encounters an instruction, the location counter is incremented by the size of the instruction. As the assembler encounters program labels, the labels are assigned the current value of the location counter. As each symbol is encountered, the symbol is saved with its value.

During pass 2, the assembler generates the object module and/or listing, as specified by the invocation line. It uses the table of label/symbols to generate machine code values for instructions. It also uses the location counter to determine the section address that each instruction should occupy. The linker does the final job of assigning where each program section will go and generating the absolute object file.

Although two passes solve the problem previously mentioned, they do not solve it in the optimal fashion. If a microprocessor has two forms of a jump instruction, one that has an 8-bit range and one that has a 16-bit range, it is up to you to choose the appropriate form of the instruction. However, if the program changes, then some short jumps must be made into long ones and vice versa. Also, in many assemblers, instructions that reference data have only one opcode, and in the case of forward references there is no choice but to use the long form of the instruction. This is because when the assembler first sees the operand it does not know its size.

Unlike most assemblers, this one has the ability to perform more than two passes and, therefore, can optimize any instructions that use a forward reference. The PASS control allows you to set the number of passes. In this case, the assembler may go through the standard pass-1 phase many times to determine the final value of each label.

Throughout this manual, two-pass mode refers to the assembler performing two passes. Optimize mode refers to the assembler performing three or more passes. By default, the assembler operates in optimize mode.

2.5 ASSIGNMENT STATEMENTS

Syntax: **[***label:***]** *symbol* = *expression***[**; *comments***]**

[*label:***]** *symbol* = *expression*:BYTE **[**; *comments***]**

[*label:***]** *symbol* = *expression*:WORD **[**; *comments***]**

The assignment statement assigns the value and all other attributes of the *expression* on the right of the equal sign to the *symbol* on the left of the equal sign. The assignment statement does not generate machine code. It simply assigns the *expression* to a *symbol*. When the *symbol* is used in an instruction statement operand field, the assigned value is used in code generation. The size attribute is passed to the debugger through the COFF file. :BYTE or :WORD may be optionally used to assign a size to the symbol. If used, it will override the size of the expression. The **&** operator may be used to force a null expression. :BYTE may also be specified as :B and :WORD as :W.

```
Example:    Y=5              ; assign the value 5 to Y.
            B1=6:BYTE        ; B1 is a byte expression with a value of 6
```

The assignment statement may also refer to the current value of the location counter. The location counter symbol (“.”) may appear on both sides of the assignment statement equal sign. If it appears on the left, it is assigned the value of the *expression* to the right side of the equal sign. In that case, the *expression* on the right must be defined during the first pass so that the pass 1 label assignments may be made.

```
Examples:    .=X'1020          ; set location counter to address X'1020
                               ; this is same as .ORG X'1020
             LOC = .           ; save current location counter value in “LOC”
```

A symbol may be assigned only one value during an assembly with an assignment statement. Attempting to redefine the value of the symbol will result in an error message. The .SET directive, however, allows symbol values to be redefined during an assembly (see Section 2.9.33.)

Some memory reference symbols are predefined, and may not be redefined. These are the registers SP, A, B, and X.

2.6 MACROS

Macros simplify the assembly process. Duplicative or similar sequences of assembly language statements can be inserted into the program source code instead of manually entering them into the program each time they are required. Once defined, a macro will automatically, during assembly time, place repetitive code or similar code with changed parameters into the assembler source code when called by its macro name. The following sections explain the process of defining and calling macros, with and without parameters, and describe how to use assembler directives related to macro generation.

Using macros, a programmer can gradually build a library of basic routines. Variables unique to particular programming applications can be defined in and passed to a particular macro when called by main programs. Such macros can be automatically included in the assembly source code using the .INCLD directive (Section 2.9.21).

2.6.1 Defining a Macro

The process of defining a macro involves preparing statements that perform the following functions:

- Assign a name to the macro
- Declare any parameters to be used
- Write the assembler statements it contains
- Establish its boundaries

Macros must be defined before they are used in a program. Macro definitions within an assembly do not generate code. Code is generated only when macros are called by the main program. Macro definitions are formed as follows:

```
.MACRO mname[,parameters]
    .
    .
    .
macro body
    .
    .
    .
.ENDM
```

where:

.MACRO is the directive mnemonic that initiates the macro definition. It must be terminated by a blank.

The *mname* is the name of the macro. It is legal to define a macro with the same name as an already existing macro, in which case the latest definition is used. Previous definitions are, however, retained in the macro definition table; if the existing macro is deleted by the .MDEL directive, the previous definition becomes active. If *mname* is the same name as an instruction mnemonic, the macro definition is used in place of the normal instruction assembly.

The main program calls the macro using the macro name. The name must adhere to the rules given for symbols in Section 2.3.3.

Parameters is the optional list of parameters used in the macro definition. Each parameter must adhere to symbol rules in Section 2.3.3. Parameters are delimited from *mname* and successive parameters by commas.

The following are examples of legal and illegal .MACRO directives:

Legal	Illegal	Reason Illegal
.MACRO MAC,A,B	.MACRO SUB,*AB	Special character used
.MACRO $ADD,OP1,OP2	.MACRO 1MAC	First character is numeric

The macro body is a sequence of assembly language statements and may consist of simple text, text with parameters, and/or macro-time operators.

The .ENDM signifies the end of the macro and must be used to terminate macro definitions.

Simple Macros

The simplest form of macro definition is one with no parameters or macro operators. The macro body is simply a sequence of assembly language statements which are substituted

for each macro call. These identical macro calls are inefficient if called repetitively within the same assembly program; a repeatedly used series of assembly language statements within a program should be coded as a subroutine. However, simple macros with no variables are useful in compiling a library of basic routines to be used within different programs. They allow the programmer to simply call the macro within the program rather then to repeatedly code all the macro body statements into each program when needed.

```
Example:                        ; MACRO "IND1" stores one indirectly into
                                ; memory using B register
            .MACRO IND1         ; begin macro definition
            LD    A,#1
            X     A,[B]
            .ENDM
```

Macros with Parameters

The previous macro could be made more flexible by adding parameters to the macro definition. Parameters allow the programmer to specify what is being loaded and stored.

```
Example:                                ; MACRO "LOAD" loads DEST with
                                        ; SOURCE
            .MACRO LOAD,DEST,SOURCE     ; DEST is destination,
                                        ; SOURCE is source
            LD    A,SOURCE
            X     A,DEST
            .ENDM
```

2.6.2 Calling a Macro

Once a macro is defined, it may be called by a program to generate code. A macro is called by placing the macro name in the operation field of the assembly language statement, followed by the actual values of the parameters to be used (if any). The following form is used for a macro call:

mname **[***parameters***]**

where: *mname* is the name previously assigned in the macro definition

parameters refer to the optional list of input parameters. When a macro is defined without parameters, the parameter list is omitted from the call.

A call to the simple IND1 macro, previously defined, is expanded as follows:

Source Program		Assembled Program
.		.
IND1	generates	IND1
.		LD A,#1
.		X A,[B]

NOTE: The macro call (IND1), the expanded macro opcodes, and source code will appear on the assembler listing if the appropriate controls are enabled. The macro call statement (IND1) itself does not generate code.

2.6.3 Using Parameters

The power of a macro can be increased by the use of optional parameters. The parameters allow variable values to be declared when the macro is called. For example, the parameter version of IND1 is LOAD, which can be used to load memory utilizing various addressing modes:

Source Program	Assembled Program
.	.
LOAD [B],#1	LOAD [B],#1
.	LD A,#1
.	X A,[B]
.	.
.	.

When parameters are included in a macro call, the following rules apply to the parameter list:

1. One comma and/or one or more blanks delimit parameters.
2. A semicolon terminates the parameter list and starts the comment field.
3. Single quotes (') may be included as part of a parameter except as the first character.
4. A parameter may be enclosed in single quotation marks ('), in which case the quotes are removed and the string is used as the parameter. This function allows blanks, commas, or semicolons to be included in the parameter. To include a quotation mark in a quoted parameter, include two quotation marks ('').
5. Missing or null parameters are treated as strings of zero length.

Parameters Referenced by Number

The macro operator **@** references the parameter list in the macro call. When used in an expression, it is replaced by the number of parameters in the macro call. For example, the following .IF directive causes the conditional code to be expanded if there are more than ten parameters in the macro call:

```
.IF @>10
```

When used with a constant or symbol (not a macro definition parameter), the **@** operator references the individual parameters in the parameter list. The following example demonstrates how this function may define and call a macro to establish a program memory table:

```
        .MACRO X
        .WORD  @1,@2,@3          ; first, second, third arguments
        .WORD  @Q                ; Qth argument
.ENDM
```

Macro call	**Generated Code**	
Q=3	Q=3	
. .		
X 3,4,5	X 3,4,5	
.	.WORD 3,4,5	; first, second, third arguments
.	.WORD 5	; Qth argument

This technique eliminates the need for naming each parameter in the macro definition, which is particularly useful to deal with long parameter lists. With the **@** parameter count operator, it is possible to create macros that have a variable number of parameters.

NOTES: 1. The **@** operator is replaced during macro expansion in comments; ordinary macro parameters are not.

2. A .DOPARM loop acts as a macro, and the above description of the **@** operator also applies.

2.6.4 Concatenation Operator

The "^" macro operator is used for concatenation. The "^" is removed and the strings on each side of the operator are compressed together after parameter substitution. If the

right string is a defined absolute null symbol (not a macro definition parameter), the decimal value of the symbol is used; if "^^" is used, the hex value of the symbol is used.

```
Example:                .MACRO LABEL,X
          R^X:          .WORD X   ; X is macro parameter
          R^Q:          .WORD Q   ; Q is defined symbol
          R^^Q:         .WORD Q
                        .ENDM

           Macro call   Generated code(without comments)
           Q=11         Q=11
            .           .
           LABEL 0      LABEL 0
            .           R0:  .WORD 0
            .           R11:  .WORD Q
                        R0x000B:  .WORD Q
```

Another example of this operation is shown in Section 2.6.7.

2.6.5 Macro Local Symbols

When a label is defined within a macro, a duplicate definition results from the second and each subsequent call of the macro. This problem can be avoided by using the .MLOC directive to declare labels local to the macro definition. The .MLOC directive may occur at any point in a macro definition, but it must precede the first occurrence of the symbol(s) it declares local. Any symbol used before the .MLOC will not be recognized as local. Local macro labels (symbols) appear as ZZ*dddd*, where *dddd* is a particular decimal number.

```
Example: ; BLOCK MOVE
         ; SOURCE is source, DEST is destination,
         ; DESTEND is last dest addr
                              .MACRO MOVE,SOURCE,DEST,DESTEND
                              LD      X,#SOURCE
                              LD      B,#DEST
                              .MLOC   BMV
         BMV:
                              LD      A,[X+]
                              X       A,[B+]
                              IFBNE   #DESTEND+1
                              JMP     BMV
         .ENDM
         Source Program       Generated Code
          .                    .
         MOVE  4000,40,47     MOVE    4000,40,47
          .                   LD      X,#4000
```

```
.                       LD      B,#40
.                       ZZ0000:
.                       LD      A,[X+]
.                       X       A,[B+]
                        IFBNE   #47+1
.                       JMP     ZZ0000
.                        .
MOVE  5000,50,57        MOVE    5000,50,57
.                       LD      X,#5000
.                       LD      B,#50
.                       ZZ0001:
.                       LD      A,[X+]
.                       X       A,[B+]
                        IFBNE   #57+1
.                       JMP     ZZ0001
```

2.6.6 Conditional Expansion

The conditional assembly directives allow the user to generate different lines of code from the same macro simply by varying the parameter values used in the macro calls. These directives are described in Section 2.9.20.

```
Example:    ; if add flag <>0, add X to A; else subtract X from A
            .MACRO ADDSUB ADDFLG,X
            .IF ADDFLG
            ADD A,#X
            .ELSE
            ADD A,# -X
            .ENDIF
            .ENDM
```

2.6.7 Macro-Time Looping

The following examples show the use of the .DO, .ENDDO, and .EXIT directives. The macro CTAB generates a constant table from 0 to MAX where MAX is a parameter of the macro call. Each word has DY: label, where Y is the decimal value of the data word:

```
            .MACRO      CTAB,MAX
            .SET        Y,0
            .DO         MAX+1
D^Y:        .WORD       Y
            .SET        Y,Y+1
            .ENDDO
            .ENDM
```

Source assembly Assembled code

```
.                       .
CTAB 2                              CTAB 2
.                                   .SET Y,0
.                       D0:         .WORD Y
.                                   .SET Y,Y+1
.                       D1:         .WORD Y
.                                   .SET Y,Y+1
.                       D2:         .WORD Y
.                                   .SET Y,Y+1
```

2.6.8 Nested Macro Calls

Nested macro calls are allowed; that is, a macro definition may contain a call to another macro. When a macro call is encountered during macro expansion, the state of the macro currently being expanded is saved and expansion begins on the nested macro. Upon completing expansion of the nested macro, expansion of the original macro resumes. The allowed number of levels of nesting depends on the sizes of the parameter lists, but at least ten is typical.

A logical extension of a nested macro call is a recursive macro call; that is, a macro that calls itself. This is allowed, but care must be taken not to generate an infinite loop.

2.6.9 Nested Macro Definitions

A macro definition can be nested within another macro. Such a macro is not defined until the outer macro is expanded and the nested .MACRO statement is executed. This allows the creation of special-purpose macros based on the outer macro parameters and, when used with the .MDEL directive, allows a macro to be defined only within the range of the macro that uses it.

2.6.10 Macro Comments

Normally all lines within a macro definition are stored with the macro. However, any text following “;;” is removed before being stored. A line that starts with “;;” is completely removed from the macro definition. These lines appear on the listing of the macro definition; they do not appear on an expansion.

2.7 ERROR AND WARNING MESSAGES

Assembler errors are divided into command line errors and assembly time errors.

2.7.1 Command Line Errors

For a command line error, the message is displayed after the invocation line. Some command line errors appear with an error number; these are described in Section 2.7.2. The other command line errors are:

Error on File	System file error.
File Conflict	The filename shown is being used multiple times as an output file, or an output filename is the same as the source filename.
File Not Found	The filename shown cannot be found. Possibly the wrong extension has been assumed or it resides in a different directory.
Disk or Directory is full	No more room exists to create an output file.
No Source File	The command line must contain at least the source filename.
Source File can't be a Device	The source file must be a disk file. It cannot be a device such as the console (CON).
Expected an option	An option must start with a /; filenames must be separated by commas.
Can't nest indirect files	An indirect file (@file) cannot be nested inside another indirect file.

2.7.2 Assembly Time Errors

When an error or warning message occurs, the "^" symbol points at, or just after, the place where the error occurred.

```
Example:    LD          A,#258
                           ^
            ERROR 12,   Value Out of Range
```

Each assembly time error is shown with its number, message, and the conditions that cause the error (see Table 2-3). Errors are formatted as follows:

```
#    message
```

where: # is the error number.

message is the error message that is displayed on the output listing.

Table 2-3 Assembler Errors

Message Number	Message and Causes
1	Invalid or Missing Opcode number is next token after label delimiter after label is not comment or EOL opcode has bad terminator
2	Undefined Opcode opcode token not in opcode tables
3	Symbol error delimiter is first character on line invalid local symbol
5	Duplicate Label/Symbol duplicate label symbol already defined .SET symbol already defined as a non .SET symbol external symbol in .PUBLIC external symbol already defined duplicate formal parameter
7	Undefined Symbol undefined symbol .PUBLIC symbol not defined macro not defined in .MDEL
9	Syntax error bad operator illegal op combination
11	Invalid Numeric number not valid for radix
12	Value Out of Range byte value out of range or relocatable .DS value too big alignment > section maximum address .SECT directive option value out of range
13	Invalid Register

Table 2-3 Assembler Errors

Message Number	Message and Causes
14	Missing or Bad Symbol .PUBLIC, .EXTRN, not a symbol option in .SECT not a symbol or after = not a symbol formal parameter not a symbol bad operand or .MDEL
15	Missing Operand missing instruction or directive operand
16	Missing or Bad Separator .IFSTR, error in string bad actual parameter macro string operator error missing separator between formal parameters
17	Missing or Bad Delimiter
18	Invalid Operand .EXTRN type is invalid bad .SECT option or hit terminator looking for option .MDEL directive operand is bad missing EQ or NE in .IFSTR
19	Multiple Externals two externals in a non-complex expression
20	Two Operands in Sequence expression has two operands without an intervening operator
22	Missing String delimiter string type operand missing the terminating delimiter
23	Invalid Keyword Usage
24	Nesting error too many conditional levels multiple .ELSE's in conditional block conditional block still open at end of program too many relocatable sections too many macro calls open conditional block on macro exit too many .INCLD levels
25	Questionable Operand Combination combination of operands not valid for the instruction

Table 2-3 Assembler Errors

Message Number	Message and Causes
26	Forward Reference
28	Relocation Usage error operand that must be absolute is relocatable relocatable operand combination invalid for operator
29	Value requires LOW/HIGH to be treated as an 8-bit value. 16 bit relocatable value
30	Invalid External Usage .= or .ORG contains external = operand is external but not the only token .END is external
31	Branch Out of Range JP, JMP, or JSR operand is out of range.
32	Trailing Characters Extra characters have been found at the end of this line; check whether it should be a comment.
33	Phase error The label had a different value during pass 1. The location counter is set to the label value. Some instructions must have varied in size between passes; use VERIFY control to check.
34	String in Expression or too Large String should not appear in expression.
35	Unrecognized Control
36	"NO" not Valid for Control This control is not allowed to use NO.
37	Can't have Primary Control A primary control can be used only on the command line or at beginning of program.
38	Bad File Name The filename in this control is invalid.
39	Can only be on Command Line This control is valid only on the command line.
40	Not used
41	Default section size specified by .Chip Directive exceeded Chip table specifies default ranges.

Table 2-3 Assembler Errors

Message Number	Message and Causes
42	Source File can't be a Device The file specified in an .INCLD must be a file.
43	File Not found
44	Invalid Local symbol usage The local symbol has the wrong format or a bad numeric.
45	Section error .= or .ORG has a relocatable operand that doesn't match the current program section. JP, JMP, or JSR address in different section
46	Options don't match previous usage An option used on the .SECT directive doesn't match the options from a previous usage. These options are ignored.
47	Opcode Usage error .ELSE, .ENDIF not in conditional .MLOC, .EXIT, .ENDM used outside of macro
48	Invalid Character in Expression An expression has a character that can't be part of an expression.
49	Attribute Conflict The attributes specified on the .SECT directive conflict. For example, both ABS and REL can't be specified for section.
50	User Error User specified via .ERROR directive.
51	User Warning User specified via .WARNING directive.
52	Invalid Processor Usage This instruction is invalid for the processor.
53	Invalid Index Register The index register is invalid for instruction.
54	Expression too Big The complex expression is too large for the assembler to process.
55	Invalid Complex Usage This operand may not be complex.

Table 2-3 Assembler Errors

Message Number	Message and Causes
56	Too Many Parameters Only 125 formal parameters can be defined; there are more actual parameters than formals for this macro.
58	Instruction or Directive not valid in Section Object code may not be generated in a section with this section type.
59	Absolute Value Required An operand of this instruction or directive must be absolute.
60	Ambiguous Control An invocation line or control line option (control) is not unique. Specify additional letters for the option.
62	Expression outside of Section. Expression outside section type range.
63	JP .+1 converted to NOP. These constructions are equal.
65	Duplicate .OPT Duplicate .OPTs for same number give error.
66	Branch out of 4K Block. Can't use use jumps >4k for same section for SMALL or MEDIUM memory models.
67	Undefined Macro. First, define macro; then call it.
68	Section size cannot be greater 4K for SMALL or MEDIUM memory model Don't violate 4K block size.
69	Bit 7 of PORTGD set. Microcontroller will enter HALT mode if bit 7 in the GPORT (0xD4) is set.
71	Branch past 0x7FFF Can't use jumps to address which is >0x7FFF.
72	Missing Section Type. Don't miss section type.
73	Start Address must be placed at zero. Label in the.END directive must be at address zero.

Table 2-3 Assembler Errors

Message Number	Message and Causes
74	Symbol name is directive Directive used as symbol name.
200	Invalid Alignment for Section or Directive The directive can't be in a section of this alignment.
204	Invalid Alignment for Section or Directive Same as 200 but this is a warning.
205	Section not Specified A .SECT directive must be used before any object code or storage can be allocated.
206	Divide by 0.
208	Missing Options, Defaults will be used On a .SECT directive, at least the section name and ROM/RAM/EERAM/REG/SEG/SEGB/BASE must be specified.
214	DEBUGGER DIRECTIVE: Too Many Dimensions Too many dimensions have been specified on the .DIM directive. (For future use only.)
215	DEBUGGER DIRECTIVE:Invalid Storage Class The value on the .SCL directive is invalid. (For future use only.)
216	DEBUGGER DIRECTIVE:Definition not in effect or didn't finish last one A debugging directive has been specified, but a .DEF directive has not been specified. Another .DEF directive has been read but a .ENDEF did not finish the last one. (For future use only.)
221	Evaluation limit exceeded For evaluation software only. (See evaluation release letter.)
222	Directive already specified Directive may be specified only once.
223	String is truncated String truncated to maximum length.
224	An absolute section starts at zero If no address given on absolute section, defaults to zero. Use abs=address to change.

Table 2-3 Assembler Errors

Message Number	Message and Causes
225	Constant Truncated Constant is truncated to 16 bits.
226	Program Counter backed up A error indicating that there is a possibility of overlapping code.
227	.DS directive used in ROM section A warning indicating that uninitialized data space was defined in ROM.
238	Chip must appear before any .sect The .chip directive should precede .sect directive.
239	Invalid Chip type The .chip directive has illegal chip name.
241	Data cannot be accessed as code Accessing non-ROM as code space.
242	Code cannot be accessed as data Accessing ROM as data space.
244	Rom address greater 0x7FFF Can't exceed maximum ROM address.
251	Section type not valid for this chip Using memory type not valid for this chip.
253	Use Range command in linker The .range and .maxrom directives do not exist for ASMCOP.
254	Section Inpage overflow Inpage section size should be <255.
255	Symbol may not be reserved name Do not use the names of predefined registers (e.g., X, SP, B) and operators (e.g., XOR) as a symbol name.
	Memory full, Type x Too many symbols, macros, etc.

2.8 THE ASSEMBLY LISTING

The listing contains program assembly language statements together with line numbers and page numbers, error messages, and a list of the symbols used in the program.

The listing of assembly language statements that generate machine code includes the hexadecimal address of memory locations used for the statement and the contents of these locations. Relocatable addresses are shown as offsets from the start of the section. To the left of the instruction, an “R” indicates a relocatable argument in this instruction, “X” indicates an external argument, “C” indicates a complex argument and “+” indicates macro expansion.

The assembler listing optionally includes an alphabetical listing of all symbols used in the program together with their values, absolute or relocatable type, word or byte or null type, section memory type, and public or external type.

Optionally, a cross-reference of all symbol usage by source line number is given; the defining line number is preceded by a “-” dash.

The total number of errors and warnings, if any, is printed with the listing. Errors and warnings associated with assembly language statements are flagged with descriptive messages on the appropriate statement lines.

The opcode checksum, opcode byte count, input file, output file, chip type (see directive .chip, Section 2.9.4), and memory model (see control model, Section 2.10.27) are shown at end of listing. Note that the opcode checksum does not reflect the final opcode checksum, if any relocatable opcodes exist.

2.9 DIRECTIVES

Directive statements control the assembly process and may generate data in the object program. The directive name may be preceded by one label and may be followed by a comment. The directive’s name occupies the operation field. Some directives require an operand field *expression*.

Assembler directive statements and their functions are summarized in Table 2-4. All directive statements begin with a period.

Table 2-4 Summary of Assembler Directives

Directive	Function	
.ADDR	8-bit address generation	Section 2.9.1
.ADDRW	16-bit address generation	Section 2.9.2
.BYTE	8-bit data generation	Section 2.9.3
.CHIP	Specify member of COP8 family	Section 2.9.4
.CONTRL	Automatic code alteration control	Section 2.9.5
.DB	8-bit data generation	Section 2.9.3
.DO	Macro loop directive	Section 2.9.6
.DOPARM	Macro loop directive	Section 2.9.7
.DSB	Reserve 8-bit data	Section 2.9.8
.DSW	Reserve 16-bit data	Section 2.9.8
.DW	16-bit data generation	Section 2.9.36
.ELSE	Conditional assembly directive	Section 2.9.20
.END	End of source program; reset address	Section 2.9.10
.ENDDO	Macro loop end	Section 2.9.6
.ENDIF	Conditional assembly directive	Section 2.9.20
.ENDM	End macro	Section 2.9.13
.ENDSECT	End program section	Section 2.9.32
.ERROR	User error message	Section 2.9.15
.EXIT	Macro loop exit	Section 2.9.6
.EXITM	Macro loop termination	Section 2.9.6
.EXTRN	Externally defined symbols	Section 2.9.17
.FB	Fill bytes	Section 2.9.18
.FW	Fill words	Section 2.9.18
.FORM	Output listing top-of-form	Section 2.9.19
.IF	Conditional assembly directive	Section 2.9.20
.IFB	Conditional assembly directive	Section 2.9.20
.IFC	Conditional assembly directive	Section 2.9.20

Table 2-4 Summary of Assembler Directives

Directive	Function	
.IFDEF	Conditional assembly directive	Section 2.9.20
.IFNB	Conditional assembly directive	Section 2.9.20
.IFNDEF	Conditional assembly directive	Section 2.9.20
.IFSTR	Conditional assembly directive	Section 2.9.20
.INCLD	Include disk file source code	Section 2.9.21
.LIST	Listing output control	Section 2.9.22
.LOCAL	Establish a new local symbol region	Section 2.9.23
.MACRO	Macro directive	Section 2.9.24
.MDEL	Macro delete directive	Section 2.9.25
.MLOC	Macro local directive	Section 2.9.26
.OPDEF	Define opcode	Section 2.9.27
.OPT	Define chip options	Section 2.9.28
.ORG	Set location counter	Section 2.9.29
.OUT	Message to console	Section 2.9.30
.OUT1	Message to console	Section 2.9.30
.OUT2	Message to console	Section 2.9.30
.OUTALL	Message to console	Section 2.9.30
.PUBLIC	Public symbols	Section 2.9.31
.SECT	Define program section	Section 2.9.32
.SET	Assign values to symbols	Section 2.9.33
.SPACE	Space n lines on output listing	Section 2.9.34
.TITLE	Identification of program	Section 2.9.35
.WARNING	User warning message	Section 2.9.15
.WORD	16-bit data generation	Section 2.9.36

2.9.1 .addr

Syntax: **[***label:***]** .ADDR *expression* **[**,*expression***]...[**; *comments***]**

Description: The .ADDR directive generates eight bits as specified by one or more *expressions* in the operand field of this directive and places them in successive memory locations. These *expressions* are usually *labels* and are used as address pointers by the COP8 JID (Jump Indirect) instruction which transfers program control to the contents of the address generated by the .ADDR directive. The lower 8 bits of each expression is stored in memory; the JID, .ADDR, and expression must all be in the same page. Otherwise, an error message will be generated.

It is highly recommended that the JID, associated .ADDR, and labels all be placed in a section with the inpage attribute (see .SECT directive, Section 2.9.32). This will guarantee that proper error checking is done. .ADDR may be used in an absolute or PAGE aligned relocatable section without the inpage attribute; however, the assembler cannot check that the JID is in the proper page.

Whenever an .ADDR expression evaluates to an address which is not in the same page as the .ADDR, an error is generated.

NOTE: If the JID is at the last byte of a 256-byte ROM block, the ROM accessed is in the next 256-byte block.

Example: Create an address pointer table to be used by the JID instruction.

```
            .SECT EXAMPLE, ROM, INPAGE
JMPI:       ADD A, # LOW (OFFSET)     ; add offset to table
            JID
OFF-        .ADDR TBL1, TBL2, TBL3
SET:
TBL1:                                 ; TBL1 code
...
TBL2:                                 ; TBL2 code
...
TBL3:                                 ; TBL3 code
            .ENDSECT
```

2.9.2 .addrw

Syntax: **[***label:***]** .ADDRW *expression* **[***,expression***]...[***; *comments***]**

Description: The .ADDRW directive generates consecutive 16-bit words of address data for each given expression. It is used for VIS address tables. If the directive has a label, it refers to the address of the first data word.

Example:
```
.SECT     EXAMPLE, ROM, ABS=0FF
VIS                                   ; place vis at 0ff
.ENDSECT
.SECT     INTERRUPT, ROM, ABS=01E0  ;vector table at 01e0
.ADDRW    LABEL
.ADDRW    LABEL1
```

NOTE: The .ADDRW directive stores words in byte order, high byte to low byte. This is the correct order for VIS address tables and is the opposite byte order of the .WORD directive.

2.9.3 .byte, .db

Syntax: **[***label*:**]** .BYTE *expression***[**,*expression*...**][**; *comments***]**

[*label*:**]** .DB *expression***[**,*expression*...**][**; *comments***]**

Description: The .BYTE and .DB directives generate consecutive 8-bit bytes of data for each given *expression*. If the directive has a *label*, it refers to the address of the first *byte*. The value of each *expression* must be in the range –256 to +255 where –256 is treated as 0, –1 as 255. The value of the expression may be interpreted either as signed or unsigned. The .BYTE and .DB directives are valid only in a ROM type section. Any label will be assigned the byte type.

The hexadecimal value of ASCII characters may be stored in memory using the .BYTE (and .DB) directive and an operand *expression* specifying character strings or their hexadecimal equivalents. (See Appendix A.)

NOTE: A single quotation mark in a string is represented by two quotation marks. An ASCII character may also be specified using the escape characters described in Section 2.3.4.

Examples:

```
1.     .BYTE  X'FF
2.  T: .BYTE  MPR-10, X'FF
3.     .BYTE  'DON''T'
4.     .DB    X'44,X'4F,X'4E,X'27,X'54
```

Example 1 stores the hexadecimal number FF in a byte of memory.

Example 2 stores two hexadecimal numbers in consecutive bytes in memory.

Examples 3 and 4 store the ASCII string (DON'T) in consecutive bytes of memory.

2.9.4 .chip

Syntax: [*label*:] .CHIP *string* [; *comments*]

Description: The .CHIP directive specifies the member of the COP8 family. Valid string arguments are the chip name with or without leading "COP"; ANYCOP is a special generic chip name which permits a common instruction subset of all chips to be assembled.

Only one .CHIP directive is allowed and must appear before any code.

Example:
```
.CHIP 820           ; specify 820
.CHIP COP888cg      ; specify 888cg
.CHIP ANYCOP        ; specify instruction subset
```

NOTE: The .chip directive informs the assembler as to instruction set and ROM/RAM ranges. Appropriate error checking is done. The chip name is also passed to linker who provides final range checking. See Appendix B for valid chip names and default ranges.

Note that the CHIP control (Section 2.10.2) will override any .CHIP directive.

The linker disallows linking of object modules or libraries with conflicting chip types except that type ANYCOP will link with any other type.

2.9.5 .contrl

Syntax: [*label*:] .CONTRL *expression* [; *comments*]

Description: The .CONTRL directive controls automatic code alteration by the assembler on instructions. This directive either decreases the number of bytes of the instruction (to optimize code), or increases the number of bytes of the JP, JMP and JSR instructions to avoid a range error. Note that in two-pass mode, the assembler can alter only those instructions which have their operand defined during pass 1, which excludes forward-reference operands. This is because the assembler must know the size of the instruction on pass 1. In optimize mode, the assembler can always choose the optimal size for these instructions.

The .CONTRL directive also allows maximum code size for all instructions to be selected for ease of patching code during debugging. The instructions all have maximum size operand fields, so any valid operand can be patched-in during debugging.

Control of the various options depends upon the three least-significant bits of the evaluated expression in the operand field (the expression must be defined during pass 1). Table 2-5 shows the options available, their associated bit weights and assembler default values. Table 2-6 shows the possible code alterations that may occur for the jump instructions.

The .CONTRL directive may be used throughout the program to enable or disable the alteration of code. Normally, it is desirable to disable code alteration of the instructions only during a block of code which must remain a fixed size (for example, a critical timing loop).

This directive takes precedence over any multi-pass optimization. Thus, if code reduction is disabled, no optimization will take place regardless of the number of passes.

Example:

```
.CONTRL 0  ; disable all code alteration
        .  ; fixed size code block
        .
        .
.CONTRL 3  ; re-enable code alteration
```

For further code optimization notes, see Section 2.4.

Table 2-5 .CONTRL Options

Control Function	Bit Positions	Binary Value	3-bit Hex Value	Description
Reduce Code	0	0	0	Suppress reduce code
(Optimize)		1	1	*Enable reduce code
Increase Code	1	0	0	Suppress increase code
(Prevent range errors)		1	2	*Enable increase code
Maximum Code	2	0	0	*Enable bit 0,1 values
(For debug patching)		1	4	Force maximum code
* indicates default.				

Table 2-6 Code Alteration Based for JP, JMP, JMPL, JSR, and JSRL

Coded Instruction	Operand In JP Range (1-byte Opcode Result)	Operand In JMP Range (2-byte Opcode Result)	Operand In JMPL Range (3-byte Opcode Result)
JP	X	+	+
JMP	–	X	+
JMPL	–	–	X
		Operand In JSR Range	**Operand In JSRL Range**
JSR	N/A	X	+
JSRL	N/A	–	X

NOTE: Bit 2 of .CONTRL must = 0

– is possible result if reduce code (bit 0) = 1

+ is possible result if increase code (bit 1) = 1

X is unchanged result, possible whether bit 0, bit 1 = 1 or 0

```
Example:                                ; default .CONTRL in effect
                                        ; (code alteration enabled)
            BCKWRD:                     ; label
                        RET
                                        ; note that following references to
                                        ; BCKWRD are not forward-references
                        JMP  BCKWRD     ; reduced to single byte (JP)
                        JP   BCKWRD     ; no change (JP)
                                        ; note that following references to
                                        ; FORWRD are forward references
                                        ; assembler cannot alter these in two-
                                        ; pass mode but can in optimize mode
                        JMP  FORWRD     ; no change (JMP) in two-pass, else (JP)
                        JP   FORWRD     ; no change (JP)
            FORWRD:
```

2.9.6 .do, .enddo, .exit, exitm

Syntax: [*label*:] .DO *count* [; *comments*]

[*label*:] .ENDDO [; *comments*]

[*label*:] .EXIT [; *comments*]

[*label*:] .EXITM [; *comments*]

Description: These directives are used to delimit a block of statements that are repeatedly assembled. The number of times the block will be assembled is specified by the .DO directive *count* value. The following is the format of a .DO – .ENDDO block:

Example:
```
.DO count
.
.
.
source
.
.
.
.ENDDO
```

The .EXIT directive is used to terminate a .DO – .ENDDO block before the count is exhausted. This directive allows the current pass through the loop to finish and then terminates looping. The .EXIT directive is commonly used in conjunction with a conditional test within a macro loop. The test will exit from the loop if a variable is equal to a particular value. In such cases, the .DO *count* value is not crucial, provided it exceeds the maximum number of times the .DO loop will be required or expected to be executed for a particular macro definition or for possible macro calls.

.EXITM is similar to .EXIT, except .EXIT allows the macro expansion to continue until the end of the macro is reached, while .EXITM terminates the macro expansion immediately.

Example:
```
.DO count
  .
  .
  .
.IF cond
.EXIT
.ENDIF
.ENDDO
```

2.9.7 .doparm

Syntax: [*label*:] .DOPARM *formal,list* [; *comments*]

Description: The .DOPARM directive repeats a macro block a number of times depending upon the number of parameters in *list*. During each expansion, the formal parameter is replaced by the next actual parameter in the *list*. The *list* may be empty, in which case one expansion takes place with a null actual parameter. Parameters in the list are treated like macro actual arguments and may be enclosed in quotation marks.

Example:
```
.DOPARM  P1,FLAG1,FLAG2,FLAG3
LD       P1,#0
.ENDDO
```

This expands to:

```
LD       FLAG1,#0
LD       FLAG2,#0
LD       FLAG3,#0
```

2.9.8 .dsb, .dsw

Syntax: **[***label:***]** .DSB *size* **[**; *comments***]**

[*label:***]** .DSW *size* **[**; *comments***]**

Description: These directives allocate a block of storage whose contents is undefined. *Size* is the size in bytes for .DSB, in words for .DSW. *Size* must be defined during pass 1. A label on a .DSB is given the byte attribute. A label on a .DSW is given the word attribute.

Example:
```
BYT: .DSB 5    ; 5-bytes
WRD: .DSW 5  ; 5 words (10-bytes)
```

NOTE: These directives give a warning for ROM sections.

2.9.9 .else

See .IF, .IFB, .IFC, .IFDEF, .IFNB, .IFNDEF, .IFSTR, .ELSE, and .ENDIF: (Conditional Assembly) directives, Section 2.9.20.

2.9.10 .end

Syntax: [*label*:] .END [*reset label*] [; *comments*]

Description: The .END directive marks the physical end of the source program. Any assembly source statement appearing after this directive is ignored. The optional *reset label* following .END indicates the start address for the program; the label must be at a start of a section (which the linker places at address zero). The *reset label* must be in the current module and must be of type ROM. If the program consists of multiple modules, only one can contain a reset label.

Example:

```
.sect code, rom
START: .
       .            ; source code
       .            ; START is starting address of program
       .END START   ; end of program
```

An included source file (see .INCLD directive) may optionally contain a .END directive, which is treated as an end-of-file but does not end the assembly. A *reset label* must *not* appear on the .END in an include file.

2.9.11 .enddo

See .DO, .ENDDO, .EXIT, and .EXITM Directives — Macro Time Looping, Section 2.9.6.

2.9.12 .endif

See .IF, .IFC, .IFB, .IFDEF, .IFNB, .IFNDEF, .IFSTR, .ELSE, and .ENDIF: (Conditional Assembly) directives, Section 2.9.20.

2.9.13 .endm

Syntax: [*label*:] .ENDM [; *comments*]

Description: The .ENDM directive marks the end of a macro definition. All macros must end with the .ENDM directive.

Example:

```
.MACRO EXMP
 .              ; macro definition code
 source
 .
.ENDM           ; end of macro
```

The optional label is in the macro definition, but the comment is not.

See Section 2.6.1 for more examples.

2.9.14 .endsect

See .SECT and .ENDSECT Directives — Program Section Directives, Section 2.9.32.

2.9.15 .error, .warning

Syntax: **[***label*:**]** .ERROR **[**'*string*'**]** **[**; *comments***]**

[*label*:**]** .WARNING **[**'*string*'**]** **[**; *comments***]**

Description: The .ERROR directive generates an error message and an assembly error that is included in the count at the end of the program. The .WARNING directive generates a warning message that is included in the warning count. These directives are useful for parameter checking.

Example:
```
.IF       VALUE<16  ; test value to see <16
LD        A,#VALUE  ; if so, generate instruction
.ELSE
.ERROR    'value>=16'  ; else generate error
.ENDIF
```

2.9.16 .exit, .exitm

See .DO, .ENDDO, .EXIT, and .EXITM Directives — Macro Time Looping, Section 2.9.6.

2.9.17 .extrn

Syntax: **[***label*:**]** .EXTRN *symbol* **[**:*type***]** **[**:*secttype***]** **[**,*symbol***[**:*type***]** **[**:s*ecttype***]** ...**]**

Description: The .EXTRN specifies symbols that are defined public in other modules, with the .PUBLIC directive, but used in this module. The external is given the byte or word attribute by specifying :BYTE (or :B) or :WORD (or :W). If no type is specified, the size attribute is null. The default section type of an external is the same as the section in which it is defined. This section type may be overridden by specifying :BASE, :RAM, :EERAM, :REG, :SEG; :SEGB, and :ROM; these section types are explained for .SECT directives.

For best error checking and most efficient code, externals should be given the same byte/word type and section type as the public symbol. Local symbols (symbols which start with **$**) may not be used with .EXTRN.

Example: Module 1

```
; word type, ROM type
.EXTRN Z:ROM:WORD
.SECT CODE,ROM
; byte type, ROM type
.EXTRN Q:BYTE
; no size type, ROM type (label)
.EXTRN LABEL
LD A, Q
JMP LABEL
```

Module 2

```
.PUBLIC Z, LABEL, Q
.SECT CODE,ROM
; word type, ROM section type
Z: .DW 2
; byte type, ROM section type
Q: .DB 3
; no size type, ROM section type (label)
LABEL: NOP
```

NOTE: It is highly recommended to place absolute value symbols (e.g., Q=2:WORD or Z=3) in an include file and include the file in an assembly rather than pass the symbols externally using .EXTRN. If .EXTRN is used with an absolute value, the appropriate sect type should be used with it, e.g., :BASE for value 0–0f, :REG for value f0–ff, no sect type for other values. Even so, the .EXTRN value may generate less efficient code than an absolute value.

2.9.18 .fb, .fw

Syntax: [*label*:] .FB *size* , *fill*

[*label*:] .FW *size* , *fill*

Description: These directives allocate a block of memory which is *size* of bytes or words in length. *Size* must be absolute and defined during pass 1.

Fill specifies the value to which each byte or word in the block is set. This value may be absolute or relocatable, but may not be complex. An error is indicated if the value does not fit within a byte for .FB (range -256 to 255).

The label refers to the address of the first byte of data. For .FB, the label is assigned byte type. For .FW, the label is assigned word type.

.FB and .FW are valid only in ROM type section.

Only one line of data will appear on the output listing.

Examples:
```
INIT: .FB  0x100,0  ; set 0x100 bytes to zero
TOP:  .FW  20,0xfff  ; 20 words filled with 0xfff
```

2.9.19 .form

Syntax: **[***label*:**]** .FORM **[**'*string*'**][**; *comments***]**

Description: The .FORM directive spaces forward to the top of the next page of the output listing (it performs a form feed). The optional *string* is printed as the third line of the page header on each page until a .FORM directive containing a new *string* is encountered.

If the assembler generates a top-of-form because the listing page is full and then immediately encounters a .FORM directive, the assembler does not generate a top-of-form for the directive.

Example: .FORM 'BCD ARITHMETIC ROUTINES'

2.9.20 .if, .ifb .ifc, .ifdef, .ifnb, .ifndef, .ifstr, .else, .endif

Syntax:

[*label*:] .IF *expression*[; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

OR

[*label*:] .IFSTR *string1 operator string2* [; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

OR

[*label*:] .IFB argument[; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

OR

[*label*:] .IFNB argument [; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

OR

[*label*:] .IFDEF *symbol* [; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

OR

[*label*:] .IFNDEF *symbol* [; *comments*]

[*label*:] .ELSE [; *comments*]

[*label*:] .ENDIF [; *comments*]

Description: The conditional assembly directives selectively assemble portions of a source program based on the operand field of the directive statement. All source statements between a .IF, .IFC, .IFSTR, .IFB, .IFNB, .IFDEF or .IFNDEF directive and its associated .ENDIF are defined as a .IF-.ENDIF block. These blocks may be nested 99 levels deep.

The .ELSE directive can be optionally included in a .IF-.ENDIF block. The .ELSE directive divides the block into two parts. The first part of the source-statements block is assembled if the .IF *expression* is not equal to zero; otherwise, the second part is assembled. When the .ELSE directive is not included in a block, the block is assembled only if the .IF *expression* is not equal to zero. If an error is detected in the *expression*, the assembler assumes a false value (zero). The *expression* must be defined during pass 1 (not containing a forward referenced value.)

The .IFSTR directive is optionally called .IFC. It allows conditional assembly based on character strings rather than the value of an expression. The *string1* and *string2* are the character strings to be compared. The *operator* is the relational operator between strings. Two operators are allowed: EQ (equal) and NE (not equal). If the relational operator is satisfied, the lines following the .IFSTR are assembled until a .ELSE or .ENDIF directive.

The primary application of the .IFSTR is to compare a macro parameter value with a specific string, for example:

```
.IFSTR @3 NE INT     ; see if third macro argument is string “INT”.
```

String1 or *string* may be enclosed within quotation marks, if the string contains special characters such as a blank. For example:

```
.IFSTR 'ADD X' EQ '@1'
```

A quotation mark within this type of string is specified by two quotation marks. For example:

```
.IFSTR @2 NE 'don"t'
```

The .IFB and .IFNB directives allow conditional assembly based on whether the operand is blank (.IFB) or non-blank (.IFNB). A comment field, as the only item following the directive, is considered a blank operand.

The .IFDEF and .IFNDEF directives allow conditional assembly based on whether the symbol is defined (.IFDEF) or undefined (.IFNDEF). The operand must consist of a single symbol, and the symbol is considered undefined if it is a forward reference.

Listing of conditional assembly code is controlled by the .LIST directive.

Labels may optionally appear on a conditional directive line. Section 2.6.6 describes the use of conditionals in macros.

Two examples of the use of the conditional assembly directives are:

Examples:

1. Two-part conditional assembly:

```
.IF       COMPR
.                     ; Assembled if COMPR non-zero.
.
.
.ELSE
.
.                     ; Assembled if COMPR is equal to zero.
.
.ENDIF
```

2. Nested .IF-.ENDIF block:

```
.IF       SMT
.                     ; Assembled if SMT is not zero.
.
.
.ELSE
.                     ; Assembled if SMT is equal to
.                     ; zero.
.
.IF       OBR
.                     ; Assembled if OBR is not zero
.                     ; and SMT is equal to zero.
.
.ENDIF
.                     ; Assembled if SMT is
.                     ; equal to zero.
.
.ENDIF
```

2.9.21 .incld

Syntax: [*label*:] .INCLD *filename* [; *comments*]

Description: The .INCLD directive includes the assembler statements in the file *filename* as part of the code being assembled.

The file must be a symbolic file; the default file extension is determined for the source file on the invocation line.

If the included lines are to be listed, the proper control line must be in the source code. See .LIST directive, Section 2.9.22.

More exactly, the .INCLD directive causes the assembler to read source code from the specified file until an end-of-file mark (or .END directive) is reached. Then it again starts reading source code from the assembly input file.

.INCLD directives may be nested 100 deep.

The directory search order is described in Section 2.2.4.

Example:

```
.INCLD    BCDADD     ; include BCDADD.ASM file.
```

2.9.22 .list

Syntax: [*label*:] .LIST *expression* [; *comments*]

Description: The .LIST directive controls listing of the source program. This includes a listing of assembled code in general, a listing of unassembled source lines contained in a .IF-.ENDIF block, a listing of code generated by the .INCLD directive, a listing of macro code, and a listing of warning messages.

The .LIST directive is equivalent to specifying several controls at once. The directive is maintained for compatibility with older assemblers.

Control of the various list options depends upon the seven least-significant bits of the evaluated expression in the operand field (bits 6 through 0). Table 2-7 shows the options available, their associated bit weights, and equivalent controls. See Section 2.10 for a description of these control functions.

Options may be combined to produce the desired listing.

Examples:

1. Full Master listing:

```
.LIST 1
```

2. Suppress listing:

```
.LIST 0
```

Table 2-7 List Options

Control Function	Bit		7-bit Hex Value	Equivalent Controls (Section 2.10)
	Positions	Binary Value		
Master List	0	0	00	#NOMASTERLIST
		1	01	#MASTERLIST
.IF-.ENDIF Block List	1	0	00	#NOCNDLINES #NOCNDDIRECTIVES
		1	02	#CNDLINES #CNDDIRECTIVES
Macro List 1. Macro calls only	3,2	00	00	#MCALLS #NOMEXPANSIONS #MOBJECT
2. Macro calls and expanded code		10	08	#MCALLS #MEXPANSIONS #MOBJECT
3. Macro calls and all expansion		11	0C	#MCALLS #MEXPANSIONS #NOMOBJECT
Full Data List	4	0	00	#NODATADIRECTIVES
		1	10	#DATADIRECTIVES
Include File List	5	0	00	#NOILINES
		1	20	#ILINES
Warning List	6	0	00	#NOWARNINGS
		1	40	#WARNINGS

2.9.23 .local

Syntax: [*label*:] .LOCAL [; *comments*]

Description: The .LOCAL directive establishes a new program section for local labels (labels beginning with a dollar sign [$]). All local labels between two .LOCAL directive statements have their values assigned to them only within that particular section of the program. Note that a .LOCAL directive is assumed at the beginning and the end of a program; thus, one .LOCAL directive within a program divides the program into two local sections. Up to 255 .LOCAL directives may appear in one assembly.

Local symbols may not be used as publics or externals.

Example:

```
$X: .WORD 1  ; first label $X
    .LOCAL   ; establish new local symbol section
$X: .WORD 1  ; second label $X, no confusion since
             ; they are in different “LOCAL” blocks
```

2.9.24 .macro

Syntax: [*label*:] .MACRO *mname*[,*parameters*] [; *comments*]

Description: The .MACRO directive names a macro and signifies the start of the macro definition.

The *mname* is the name of the macro. The name must conform to the definition of a symbol; see Section 2.3.3. The *parameters* are used in the macro definition. Each parameter must also conform to the symbol definition rules.

See Section 2.6.1 for a description of macro definition. Note that the optional label and comment on the directive line are not included in the macro definition.

2.9.25 .mdel

Syntax: **[***label*:**]** .MDEL *mname***[**,*mname***]**... **[**; *comments***]**

Description: The .MDEL directive deletes the macro definition and restores the previous macro definition of the same name (if any).

Example:
```
.MACRO INC  ; first macro def
LD A,[X+]
.ENDM
.MACRO INC  ; second macro def of same name
LD A,[B+]
.ENDM
.MDEL INC   ; INC is now first defined macro
```

2.9.26 .mloc

Syntax: **[***label*:**]** .MLOC *symbol***[**,*symbol***]**... **[**; *comments***]**

Description: When a label is defined within a macro, a duplicate definition results with the second and each subsequent call of the macro. This problem can be avoided by using the .MLOC directive to declare labels local to the macro definition.

Refer to Section 2.6.5 for an example of the .MLOC directive.

2.9.27 .opdef

Syntax: **[***label***]** .OPDEF *mname, opcode*

Description: Opdef assigns another name, *mname*, to the specified *opcode*. This name can then be used just like the original opcode. Of course, the original opcode is still available. *Opcode* may be a standard opcode, a previously defined macro, or a symbol defined in a previous .OPDEF.

Examples:
```
.OPDEF  IRP,.DOPARM ; note that even though .DOPARM is a
                    ; directive, IRP doesn't
                    ; require a leading period.
```

2.9.28 .opt

Syntax: [*label*:] .OPT *expression*, *expression* [; *comments*]

Description: The .OPT directive is for future use. It specifies which mask-programmable options were selected for the COP8. The first *expression* indicates the option number; the second *expression* indicates the value to be assigned to the specified option number. Values for the first *expression* (option numbers) must be within the range 1 through 68; values for the second *expression* (option values) must be within the range 0 through 255.

The assembler places the .OPT directive information in the load module output file.

Example:
```
.OPT   1,3     ; Specify option 1=3
```

2.9.29 .org

Syntax: [*label*:] .ORG *expression* [; *comments*]

Description: The .ORG directive is used to set the program counter to a new value. Any bytes skipped have an indeterminate value.

The *expression* must not contain any forward references and must be either absolute, or relocatable in the same section as the current program counter. The optional label is set to the value of the new program counter.

A .ORG expression is equivalent to a .= expression.

In an absolute section (see Section 2.9.32, .SECT directive), the program counter may not be set to a value lower than that specified on the .SECT directive.

Example:
```
.ORG 0100          ; section program counter = 0100
```

2.9.30 .outall, .out1, .out2, .out

Syntax: **[***label*:**]** .OUTALL '*string*' **[**; *comments***]**

[*label*:**]** .OUT1 '*string*' **[**; *comments***]**

[*label*:**]** .OUT2 '*string*' **[**; *comments***]**

[*label*:**]** .OUT '*string*' **[**; *comments***]**

Description: These directives write a message to the console. The .OUT1 is performed only during pass 1 of the assembly, .OUT2 during pass 2, and .OUT on both passes. The .OUTALL may be used to output a message on all passes of the assembler, including all optimization passes.

The operand field contains the message to be output.

Example:
```
.OUT1  'Pass 1'
.OUT   'ARG contains bad value'
```

2.9.31 .public

Syntax: **[***label*:**]** .PUBLIC *symbol***[**,*symbol*...**]**

Description: This directive specifies which symbols to make available to other modules. In the other module, a .EXTRN directive is used. A symbol may be any value defined in this module. Local symbols (symbols which start with **$**) may not be used with .PUBLIC.

Example:
```
.PUBLIC DOG
DOG:  .DW 2
```

2.9.32 .sect, .endsect

Syntax: **[***label*:**]** .SECT *name*, **[***memory* **[**,REL**]** **[**,*align*****][**,INPAGE**]]** **[**;*comments***]**

[*label*:**]** .SECT *name*, **[***memory* ,COMMON **[**,*align*****][**,INPAGE**]]** **[**;*comments***]**

[*label*:**]** .SECT *name*, **[***memory*, ABS=*addr***[**,INPAGE**]][**;*comments***]**

[*label*:**]** .ENDSECT **[**;*comments***]**

where:

name Specifies section name. This can be any valid symbol name.

memory

- ROM is read-only memory and holds code/constant data.
- RAM is read-write memory and holds data.
- BASE is RAM, range 0 to f.
- EERAM is RAM range 080 to 0bf.
- REG is RAM range 0f0 to 0ff.
- SEG is RAM above 0ff.
- SEGB is base area of SEG, address xx00 to xx0f.

REL REL is relocatable (default).

align Specifies the align type and, if relocatable, the boundary on which the loader places the relocatable section. This may be:

- byte any address (default)
- page address divisible by 0x100
- block address divisible by 0x1000

COMMON COMMON is same as REL, except that sections of the same name in two or more modules will overlay each other. The size of the section is size of the largest.

ABS ABS is absolute. An address is specified, e.g., ABS=0x1000. This sets the program counter to this address and also indicates the lowest value the program counter may have in this section.

INPAGE INPAGE is used to guarantee that the entire section falls within a page, exclusive of the last byte of the page, i.e., the range xx00 to xxfe. This guarantees that a JID or LAID instruction along with associated data falls within a page. See .ADDR directive (Section 2.9.1) for an example.

Description: This directive defines a program section. It specifies a section name and attributes. If this is the first use of the section, its location counter is set to zero. If the section has already been used, its location counter is set to its previous value. The section names and attributes are used by the linker to combine similarly named sections from other modules.

If only the section name is given, the attributes from the first definition of that section name are used. A section name may not be defined again with different attributes.

The .ENDSECT is optionally used to end a program section and restore the previous section (otherwise, the next .SECT ends a section and defines a new one).

Example:

```
.SECT CSECT,ROM          ; define CSECT section
.DB 2
.SECT DSECT,RAM          ; define DSECT section
.DSB 1
.ENDSECT                 ; end DSECT section
.DW 2                    ; back to CSECT section
```

NOTES: 1. A .SECT directive must appear at the start of a module before first data or code usage.

2. The assembler generates a general instruction addressing mode for any BASE or REG address whose offset is outside the BASE (0 to F) or REG (F0 to FF) range. This is the only way correct code is assured. For example:

```
    .SECT  B1,BASE
L1: .DSB   1
L2: .DSB   2
    .SECT  CODE,ROM
    LD     B,#L1          ; 1 byte instruction
    LD     B,#L1-1        ; 2 byte instruction
    LD     B,#L2-1        ; 1 byte instruction
```

For the first LD, the offset of L1 from the start of section B1 is 0 so a one byte instruction is generated. The second LD has an offset of –1; after linking, L1-1 could actually have the value –1 thus a two-byte instruction is generated. The final LD has an offset of 0 so a one-byte instruction is generated.

2.9.33 .set

Syntax: **[***label*:**]** .SET *symbol*, *expression***[**:*type***][**; *comments***]**

Description: The .SET directive is used to assign values to *symbols*. In contrast to an assignment statement, a *symbol* assigned a value with the .SET directive can be assigned a new value at any place within an assembly language program.

The optional type may be specified as byte (:BYTE or :B) or word (:WORD or :W) and assigns this type to the symbol. Use of type overrides the type of the expression.

Example:
```
.SET A1,0           ; set A1=0
.SET A1,100         ; set A1=100
.SET B1,50          ; set B1=50
.SET C1,A1-25*B1/4  ; set C1=A1-25*B1/4
.SET D1,A1:BYTE     ; set D1= to value 100 with byte type
```

NOTE: A .SET symbol may be set to a forward-referenced operand only the first time the .SET symbol is defined. Following definitions with forward-referenced operands generate errors.

2.9.34 .space

Syntax: [*label*:] .SPACE *expression* [; *comments*]

Description: The .SPACE directive inserts a number of blank lines into the output listing. The number of lines inserted is specified by the *expression* in the operand field. If the HEADINGS control is on, the number of blank lines inserted will not exceed the end of the current page.

Example:
```
.SPACE 20                ; skip 20 lines.
```

2.9.35 .title

Syntax: **[***label*:**]** .TITLE **[***symbol***][**,'*string*'**][**; *comments***]**

Description: The .TITLE directive identifies the output listing in which it appears with an optional *symbolic name* and an optional title *string*. If more than one .TITLE directive is used, the last one encountered specifies the *symbolic name* and *string*. The *symbolic name* consists of any character up to the comma. Single quotation marks (') must appear at the beginning and end of the character *string*. A single quotation mark in the string is represented by two quotation marks ('').

Example: .TITLE TBLKP, 'TABLE LOOKUP'

uses TBLKP on the first header line. 'TABLE LOOKUP' appears as the second header line.

.TITLE ,'DATA TABLE'

uses default module name. 'DATA TABLE' appears as the second header line.

NOTE: .TITLE must appear as the first source line if the header of the first page is to contain symbol and string.

2.9.36 .word, .dw

Syntax: **[***label*:**]** .WORD *expression***[**,*expression*...**]** **[**; *comments***]**

[*label*:**]** .DW *expression***[**,*expression*...**]** **[**; *comments***]**

Description: The .WORD and .DW directives generate consecutive 16-bit words of data for each given expression. If the directive has a label, it refers to the address of the first word. The program section must be ROM type. Any label is assigned the word type.

Examples:
```
1.      .WORD  0FF
2.  T: .DW  MPR-10,0FFFF,'AB'
```

Example 1 stores the hexadecimal number FF in a word of memory.

Example 2 stores three 16-bit values in consecutive words in memory.

To store ASCII strings in consecutive bytes, use the .BYTE directive, Section 2.9.3.

NOTE: The .WORD, .DW directive stores words in byte order, low byte to high byte. This is **not** correct for VIS address table; use .ADDRW directive which stores in opposite order.

2.10 ASSEMBLER CONTROLS

This section describes the controls that may be used in the source program on a control line or used on the invocation line as an option. The syntax of their usage as an option is described in Section 2.2.2.

A control line is indicated by a # in column 1 of the source line, followed by any of the following controls separated by white space. Comments may be included on a control line by using a semicolon followed by the comment. The ";" terminates the control line.

A control name may be abbreviated to the number of letters shown in capital letters in the descriptions. For example, MASTERLIST may be also specified as MASTER, MA, or even M. However, MC gives an error, since it is uncertain whether MCALLS or MCOMMENTS is wanted. Many of the controls may also have the prefix NO. In this case, the rest of the name may be abbreviated as normal. Control names are not case-sensitive.

A control may be classified as primary, general or invocation line only. Primary controls are those that can be set only on the invocation line or at the beginning of the program before any other statements, except for other control lines or comments. General controls may be specified anywhere and can be respecified at any time. Thus, the usual source program structure is:

```
#MASTERLIST        ;general is okay up here

; comment okay between primary controls

#HEADINGS          ;turn on headings

; instruction follows

        NOP

; at this point only general controls will be valid
```

A few of the controls are shown as "invocation line only." This, of course, means they are valid only on the program invocation line and cannot be used within the program.

A general control may be saved and restored by the SAVE and RESTORE controls.

Invocation line controls are masters and override the same control in the program source. Thus, NOMASTERLIST on the invocation line overrides #MASTERLIST in the source.

Assembler controls and functions, described in this section, are summarized in Table 2-8.

Table 2-8 Summary of Assembler Controls

Control	Function	
[NO]Aserrorfile[=*file*]	[NO]Create a source file with errors	Section 2.10.1
CHip=string	Specify member of COP8 family	Section 2.10.2
[NO]CNDDirectives	[NO]List of conditional directives	Section 2.10.3
[NO]CNDLines	[NO]List lines of conditional code	Section 2.10.4
[NO]COMMentlines	[NO]List comment lines	Section 2.10.5
[NO]COMPlexrel	[NO]Complex relocation	Section 2.10.6
[NO]CONstants	[NO]List constants in symbol table	Section 2.10.7
[NO]Crossref	[NO]Cross-reference table in listing file	Section 2.10.8
[NO]DAtadirectives	[NO]List all lines of object code	Section 2.10.9
Define=*symbol*[=*value*]	Define a symbol	Section 2.10.10
[NO]ECho	[NO]Echo command file to the console	Section 2.10.11
[NO]Errorfile[=*file*]	[NO]Error file	Section 2.10.12
[NO]Formfeed	[NO]Form feeds	Section 2.10.13
[NO]Headings	[NO]Heading on each list page	Section 2.10.14
[NO]ILines	[NO]List include file	Section 2.10.15
Include=*directory*	Additional include search directory	Section 2.10.16
[NO]Listfile[=*file*]	[NO]List file	Section 2.10.17
[NO]LOcalsymbols	[NO]Local symbols in object file	Section 2.10.18
[NO]Masterlist	[NO]List of source lines in list file	Section 2.10.19
[NO]MCAlls	[NO]List of macro call statements	Section 2.10.20
[NO]MCOmments	[NO]Macro comments saved in definition	Section 2.10.21
[NO]MDefinitions	[NO]List of macro definition	Section 2.10.22
[NO]MEMory	Use memory for optimization	Section 2.10.23
[NO]MExpansions	[NO]List macro expansion lines	Section 2.10.24
[NO]MLocal	[NO]Macro local symbols in symbol table	Section 2.10.25
[NO]MObject	[NO]List macro object lines only	Section 2.10.26

Table 2-8 Summary of Assembler Controls

Control	Function	
MODel=*model*	Memory Model	Section 2.10.27
[NO]Numberlines	[NO]Number list lines by source file	Section 2.10.28
[NO]Objectfile[=*file*]	[NO]Object file	Section 2.10.29
PAss=*number*	Number of passes assembler performs	Section 2.10.30
PLength=*number*	Number of lines per page	Section 2.10.31
PWidth=*number*	Number of characters per line	Section 2.10.32
Quick	Fast list only	Section 2.10.33
[NO]Remove	[NO]Remove source file error lines	Section 2.10.34
REStore	Restore state of controls that were saved	Section 2.10.35
SAve	Saves state of general controls	Section 2.10.36
[NO]SIGnedcompare	[NO]Comparisons using signed arithmetic	Section 2.10.37
SIZesymbol=*number*	Specifies maximum symbol size	Section 2.10.38
SYm_debug	Generate source debugging information	Section 2.10.39
[NO]Tablesymbols	[NO]Symbol table in list file	Section 2.10.40
[NO]TABS	[NO]Tabs in list file	Section 2.10.41
Undefine=*symbol*	Undefine a symbol	Section 2.10.42
[NO]UPpercase	[NO]Convert symbols to upper-case	Section 2.10.43
[NO]Verify	[NO]List all passes of assembler	Section 2.10.44
[NO]Warnings	[NO]List warning messages	Section 2.10.45
[NO]Xdirectory	[NO]Search only specified Include directories	Section 2.10.46

2.10.1 Aserrorfile

Syntax: [NO]Aserrorfile [=*filename*]

Description: This control causes the program to create a source error file. This file contains all the lines that are in the source program plus error messages for those lines containing errors. You can then edit this file and quickly fix the errors, as both the source statements and errors are combined. Normally, the user then runs this file through the assembler using the REMOVE control with ASERRORFILE specified again. An error-free source file can then be created. The class is invocation line only. The default is NOASERRORFILE. If *filename* is not specified, the source filename is used with the default extension. The default filename extension is .ase.

Example:
```
A                       ; uses file sourcefile.ase
ASE=MAIN.NEW
```

NOTE: The generated error lines start with ;!*** if the line in error is found in the source file. (The source file is the last input file specified on the invocation line.) The error lines start with ;!+++ if the line is not in the source file (e.g., in include file or part of macro expansion); the line in error is also shown preceded by ;!+++.

ERRORFILE (Section 2.10.12) is used to generate a file containing only errors.

2.10.2 CHip

Syntax: CHip=*string*

Description: The CHIP control specifies the member of the COP8 family. Valid string arguments are the same as for the .CHIP directive (see Section 2.9.4). The class is invocation line only.

Example:
```
CHIP=820          ;specify 820 instruction set
CHIP=COP888cg     ;specify COP888cg instruction set
CH=ANYCOP         ;specify instruction subset
```

NOTE: See notes for .CHIP directive (Section 2.9.4).

2.10.3 Cnddirectives

Syntax: [NO]CNDDirectives

Description: This control enables/disables the listing of conditional directives (e.g., .IF, .ELSE, and .ENDIF). It is overridden by the NOCNDLINES control. Thus, while NOCNDDIRECTIVES can be used to disable listing all conditional directives, CNDD does not list those lines suppressed by NOCNDLINES. The default is NOCNDDIRECTIVES. The class is general.

2.10.4 Cndlines

Syntax: [NO]CNDLines

Description: CNDLINES lists lines of code that are not assembled because of conditional assembly. NOCNDLINES inhibits the listing of these lines. The default is NOCNDLINES. The class is general.

2.10.5 Commentlines

Syntax: [NO]COMMentlines

Description: This control allows you to include comment lines or not in the output listing. It is provided to allow the user to get a quick listing. Note that a blank line is considered a comment. The default is COMMENTLINES. The class is general.

2.10.6 Complexrel

Syntax: [NO]COMPlexrel

Description: This control enables/disables complex relocation(see Section 2.3.4). In most cases, the user isn't concerned about this control. However, in programs in which it is known that there are no complex expressions, it is useful for error-checking purposes to use NOCOMPLEXREL. It is the unusual program that requires complex expressions. The default is COMPLEXREL. The class is general.

NOTE: Complex relocation is any relocation expression, other than rterm + constant, rterm – constant, or rterm – rterm where rterm is relocatable. For example, rterm * 2 is complex.

2.10.7 Constants

Syntax: [NO]CONstants

Description: Many programs define hundreds of constant symbols (e.g., CR=13), and typically a .INCLD file is used to include the definitions of these symbols. Normally all symbols, including these, are listed in a symbol or cross-reference table. NOCONSTANTS can be used to ignore these constant symbols in the symbol/cross-reference table, effectively "cleaning up" the listing. If LOCALSYMBOLS is in effect, NOCONSTANTS also inhibits constant symbols from being placed into the object module. Note that a constant symbol is different from a label defined in an absolute section. The default is CONSTANTS. The class is primary.

2.10.8 Crossref

Syntax: [NO]Crossref

Description: CROSSREF causes a cross-reference table to be included in the output listing. This table consists of each symbol and its value along with the line numbers of where it is defined, and used. The line number where it is defined is preceded by a "-" on the listing. CROSSREF overrides TABLESYMBOLS. The default is NOCROSSREF. The class is primary.

2.10.9 Datadirectives

Syntax: [NO]DAtadirectives

Description: Some data generation directives display more than one line of object code in the listing file. DATADIRECTIVES allows all of these lines to be listed while NODATADIRECTIVES causes only the first line to be listed. The default is DATADIRECTIVES. The class is general.

NOTE: The .FW and .FB directives display at most one line of object code.

2.10.10 Define

Syntax: Define=*symbol*[=*value*]

Description: This control is used to define a symbol from the invocation line. This could then be used within the program to create different versions, etc. The conditional assembly directive, .IFDEF or .IFNDEF, can also detect if the symbol has been defined. If no value is specified for the symbol, it is assigned a value of 1. The symbol name is case-sensitive. The class is invocation line only.

Example:
```
DEF=LOOPCOUNT=10  ; set LOOPCOUNT to 10
D=VERSION2        ; set VERSION2 to default of 1
```

2.10.11 Echo

Syntax: [NO]ECho

Description: This control specifies whether an MS-DOS command file (i.e., @file) is echoed to the console. ECHO only applies to commands that follow it. The default is NOECHO. The class is invocation line only.

2.10.12 Errorfile

Syntax: [NO]Errorfile [=*filename*]

Description: This control specifies the name of a file to which errors are written. ERRORFILE without the filename option uses the source filename with the default extension .err. NOERRORFILE indicates that no error file is to be used. The default is ERRORFILE to the console (unless the listing file is also the console). The class is invocation line only.

Example: ERR=MAIN ; errors to main.err

2.10.13 Formfeed

Syntax: [NO]Formfeed

Description: FORMFEED puts a form-feed character between pages. NOFORMFEED uses blank lines to move between pages. The number of blank lines is dependent upon the page length. The default is FORMFEED. The class is primary.

2.10.14 Headings

Syntax: [NO]Headings

Description: Normally the output listing is divided into pages whose size is specified by the PLENGTH control. Each page is separated by a form feed or by the appropriate number of line feeds. The top of each page has a header that contains the page number, title, and date. NOHEADING puts no page breaks in the output listing. The PLENGTH control is ignored, no headers appear, and all lines are listed continuously. The default is HEADINGS. The class is primary.

2.10.15 Ilines

Syntax: [NO]ILines

Description: ILINES lists lines read from .INCLD files. The default is NOILINES. The class is general.

2.10.16 Include

Syntax: Include = *directory*

Description: Normally, when using the .INCLD directive, the program looks in the current and default directories to find a file without an explicit directory name (refer to Section 2.2.4). If not found, it flags an error. This control enables the program to search other devices and/or directories to find the file. Multiple directories may be searched by specifying each on a separate INCLUDE control. The class is invocation line only.

Example:
```
I=C:\   ; check root directory on drive C:
I=. .   ; check parent directory
```

2.10.17 Listfile

Syntax: [NO]Listfile [=*filename*]

Description: LISTFILE causes the program to write the listing to the file specified. LISTFILE with no filename uses the default extension with the source filename. NOLISTFILE inhibits any listing from being created. In this case, the default is that all errors are displayed on the console. (Refer to Section 2.10.12.) The default is LISTFILE with only error lines appearing in the file. If LISTFILE is specified, the file contains all lines not inhibited by other options. The default filename extension is .lis. The class is invocation line only.

Example:
```
LIST=N          ; creates file n.lis
L=CON           ; list to con (MS-DOS console)
```

2.10.18 Localsymbols

Syntax: [NO]LOcalsymbols

Description: LOCALSYMBOLS causes all symbols to be put into the object module and thus be made available for the linker symbol or cross-reference table. If LOCALSYMBOLS is not used, only those symbols declared PUBLIC are available to the linker. The default is NOLOCALSYMBOLS. The class is primary.

NOTE: For very large programs, when LOCALSYMBOLS assembly control is used, the large number of symbols may exceed the available memory during link time. Even if there is enough memory, the link may be very slow. It is recommended to limit number of assemblies with LOCALSYMBOL control.

2.10.19 Masterlist

Syntax: [NO]Masterlist

Description: MASTERLIST enables the listing of the source lines in the list file. NOMASTERLIST causes only error lines to be placed into the listing file. NOMASTERLIST overrides all other list directives. The default is MASTERLIST. The class is general.

In order to generate a full listing, you must also specify the /LIST option on the invocation line. See Section 2.10.17.

2.10.20 Mcalls — Macro Calls

Syntax: [NO]MCAlls

Description: MCALLS allows the listing of macro call statements while NOMCALLS inhibits them. The default is MCALLS. The class is general.

2.10.21 Mcomments — Macro Comments

Syntax: [NO]MCOmments

Description: NOMCOMMENTS suppresses all comments within a macro definition. When this macro is expanded, the comments do not appear. These comments do appear on the output listing as part of the macro definition. A macro double-comment, e.g., ;;COMMENT, never appears in a macro expansion. This control allows all comments to be removed. Blank lines are considered comments. The default is MCOMMENTS. The class is general.

2.10.22 Mdefinitions

Syntax: [NO]MDefinitions

Description: MDEFINITIONS allows the listing of a macro definition, whereas NOMDEFINITION inhibits it. A macro definition consists of all lines from a .MACRO, .DOPARM, or .DO directive to its matching .ENDM or .ENDDO. The default is MDEFINITIONS. The class is general.

2.10.23 Memory

Syntax: [NO]MEMory

Description: MEMORY allows memory to be used for optimization. NOMEMORY uses temporary files for optimization; this need be used only if you get an ”out of memory” error during assembly. Default is MEMORY. The class is invocation line only.

2.10.24 Mexpansions

Syntax: [NO]MExpansions

Description: MEXPANSIONS causes all macro expansion lines to be listed. NOMEXPANSIONS suppresses the macro expansion lines. A macro expansion consists of those lines that are generated by a macro call, a .DO, or a .DOPARM directive. The macro call line is listed dependent on the MCALLS control. The default is NOMEXPANSIONS. The class is general.

2.10.25 Mlocal — Macro Local Symbols

Syntax: [NO]MLocal

Description: MLOCAL causes “local macro” symbols to be included in a symbol or cross-reference table. In addition, these symbols are placed into the object module if LOCALSYMBOLS(see Section 2.10.18) is in effect. The default is NOMLOCAL. The class is primary.

2.10.26 Mobject — Macro Object

Syntax: [NO]MObject

Description: NOMOBJECT allows all lines of a macro expansion to be listed. MOBJECT lists only those macro lines that generate object code. This is very useful for those macros that contain many conditional statements but only a few instructions. This control is overridden by NOMEXPANSIONS(see Section 2.10.24). The default is NOMOBJECT. The class is general.

2.10.27 Model—Memory Size Model

Syntax: MODel=s[mall]|m[edium]|l[arge]

Description: MODEL specifies an alternative memory model to the default specified by the .CHIP directive(see Section 2.10.2). By default, all chips are small memory model, except if ROM size is >4K then LARGE memory model is used.

If program size is <4K, SMALL model should always be used; no 3-byte JMPL or JSRL are generated. If program size is >4K, either MEDIUM or LARGE should be used. MEDIUM requires all ROM sections to not cross 4K boundaries (LNCOP will enforce this). Jumps between ROM sections are 3-byte jumps; jumps within sections are one or two bytes. LARGE places no restrictions on ROM section placement, but all jumps (except 1-byte jumps) are three bytes. You can build all modules with MEDIUM and link, then with LARGE and link, to see which is more efficient program size. In general, many small code sections are more efficient in MEDIUM model; large code sections are more efficient in LARGE model.

NOTE: All assembly modules should be built with the same model. The memory model is shown at the end of the listing.

2.10.28 Numberlines

Syntax: [NO]Numberlines

Description: NONUMBERLINES causes each line in the output listing to have the next sequential line number. However, because of macros or .INCLD files, the line number may not reflect the actual position in the source file. While this is useful for a cross-reference table, it may not be for other purposes. NUMBERLINES causes each source line to have the same line number as its position in the file. In this case, lines from macro expansions or .INCLD files do not have line numbers. The default is NUMBERLINES. The class is primary.

2.10.29 Objectfile

Syntax: [NO]Objectfile [=*filename*]

Description: OBJECTFILE causes the program to write the object module to the specified file. OBJECTFILE with no *filename* uses the default extension with the source filename. NOOBJECTFILE inhibits any object module from being created. The default is OBJECTFILE. The default filename extension is .obj. The class is invocation line only.

Example:
```
O=DISPLAY  ; uses file display.obj
NOOBJ      ; no object module
```

2.10.30 Pass

Syntax: PAss = {*number* | ALL}

Description: PASS specifies the number of passes through the source code that the assembler should perform. An argument of ALL causes the assembler to perform the minimum number of passes necessary to optimize the size of the code. A count of 0, 1 or 2 causes the assembler to perform a standard two-pass assembly. Any other value causes the assembler to perform that number of passes (this may result in phase errors; it is best to specify ALL for pass>2). The default is ALL. The class is invocation line only.

NOTE: It is rarely necessary to use pass=2, as the optimization time is minimal. The .CONTRL directive(see Section 2.9.5) can be used to selectively turn off optimization for sections of code.

2.10.31 Plength

Syntax: PLength = *number*

Description: PLENGTH specifies the number of lines for each page in the listing file. This is the physical length of the page, not the number of lines that will appear on the page. Thus, a printer using 8 lines/inch specifies PLENGTH=88 using standard 11-inch paper. Values between 11 and 255 can be specified. The default is 66. The class is primary.

PLENGTH is overridden by the NOHEADINGS control.

Example: PL=60

2.10.32 Pwidth

Syntax: PWidth = *number*

Description: This control sets the number of characters per line that are printed on the output listing. Characters past this position are ignored. This is also used as the position to which the page number and date are aligned in the header. Values between 64 and 255 can be specified. Although some terminals are 80 characters wide, writing a CR-LF in the 80th spot causes a subsequent blank line to appear. In this case, you should use 79 as the width. The default is 79 and the class is primary.

Example: PW=100

2.10.33 Quick

Syntax: Quick

Description: Specify QUICK to get a fast, errors-only assembly. QUICK causes NOMASTERLIST, NOLISTFILE, NOOBJECTFILE, NOASERRORFILE, NOTABLESYMBOLS, and NOCROSSREF to be in effect. It also enables ERRORFILE, which defaults to the console. If any of the above controls are used after QUICK, they can be turned on again. In most cases, you can enter the source file to assemble followed by the QUICK control. The class is invocation line only.

Example:
```
Q               ; just errors to console
QUICK OBJ       ; also get object file
```

2.10.34 Remove

Syntax: [NO]Remove

Description: When a source error file is created by ASERRORFILE control(see Section 2.10.1), all errors are flagged with an “error string” at the beginning of the error message line(s). You can then edit this file to correct the errors, while leaving the “error string” lines. If this file is subsequently processed by the assembler, REMOVE can be used to ignore these “error string” lines so that the assembly proceeds normally. If there is still an error, it appears on an error line. The default is NOREMOVE. The class is primary.

NOTE: See ASERRORFILE control (Section 2.10.1) for definition of “error string.”

2.10.35 Restore

Syntax: REStore

Description: This control restores the state of the controls that were saved with the SAVE control(see Section 2.10.36). An error is flagged if RESTORE is used without a previous SAVE. The class is general, although RESTORE is not normally used on the invocation line.

2.10.36 Save

Syntax: SAve

Description: This control saves the state of general controls, except for SAVE and RESTORE themselves. SAVE's can be nested to eight levels. The state of these controls can be restored with the RESTORE control. The typical use is to save the state of the MASTERLIST flag before calling a macro that turns off the listing. After the macro, RESTORE is used to set the MASTERLIST flag to its starting state. The class is general, although SAVE is normally not used on the invocation line.

2.10.37 Signedcompare

Syntax: [NO]SIGnedcompare

Description: NOSIGNEDCOMPARE causes all expressions containing conditional operators (i.e., GT, GE, LT, or LE) to be evaluated using unsigned arithmetic. SIGNEDCOMPARE evaluates these expressions using signed arithmetic. The default is NOSIGNEDCOMPARE. The class is general.

2.10.38 Sizesymbol

Syntax: SIZesymbol = *number* (6-64)

Description: SIZESYMBOL specifies the maximum symbol size. All symbols are stored as variable lengths, so there is usually no need to change the maximum symbol size. However, in certain cases, such as for compatibility with older assemblers, you may need to limit symbols to a maximum size. In this case, only the maximum number of characters are used to represent the symbol, the rest are ignored. The default is 64 and the class is primary.

2.10.39 Sym_debug

Syntax: SYm_debug

Description: SYM_DEBUG causes the assembler to generate source line and symbolic debugging information in the object module. This information is then available to the COP8 linker which can process it to create a COFF file with debugging information. The COFF file can be used by the COP8 debuggers.

Debugging information is generated for all symbols, except for multiple occurrences of “local labels” (labels which start with $), in which case only the first occurrence is handled. Symbols by default are marked static unless explicitly declared public or external.

The class is invocation line only.

2.10.40 Tablesymbols

Syntax: [NO]Tablesymbols

Description: NOTABLESYMBOLS specifies that no symbol table is listed, whereas TABLESYMBOLS creates a symbol table listing. This control is overridden by the CROSSREF control. Note that NOMASTERLIST(see Section 2.10.19) does not override TABLESYMBOLS. The default is NOTABLESYMBOLS. The class is primary.

2.10.41 Tabs

Syntax: [NO]TABS

Description: TABS specifies that tabs should be retained in the output listing, whereas NOTABS causes tabs to be expanded into the appropriate number of blanks. Tab stops are at every 8 columns. TABS is useful in reducing the size of the output listing and (possibly) increasing the printing speed. The default is NOTABS. The class is primary.

2.10.42 Undefine

Syntax: Undefine=*symbol*

Description: UNDEFINE is used to undefine a symbol that was defined by the DEFINE control(see Section 2.10.10). Note that it cannot be used to undefine a symbol defined in the source program. The class is invocation line only.

Example: U=COUNT

2.10.43 Uppercase

Syntax: [NO]UPpercase

Description: UPPERCASE causes all lower-case characters to be converted to upper-case in symbols and opcodes. This does not affect characters used in strings. NOUPPERCASE allows symbols using upper- and lower-case to be considered differently. For example, ABCD and abcd are different symbols if NOUPPERCASE is in effect. All assembler keywords and opcodes can be any combination of upper- and lower-case. Thus, NOP, nop, and NoP are all recognized as an opcode, regardless of the state of this control. The default is NOUPPERCASE. The class is primary.

2.10.44 Verify

Syntax: [NO]Verify

Description: VERIFY causes an output listing to be generated during each pass of the assembler. This is mainly used to examine the effect of multi-pass optimization or to determine how phase errors occurred. The default is NOVERIFY. The class is primary.

2.10.45 Warnings

Syntax: [NO]Warnings

Description: WARNINGS causes any warning messages to be included in the output listing. NOWARNINGS inhibits the messages. The default is WARNINGS. The class is general.

2.10.46 Xdirectory

Syntax: [NO]Xdirectory

Description: Normally, when searching for an include file, the current, default, and any directories specified via the INCLUDE control are searched. XDIRECTORY causes only those directories specified by the INCLUDE control to be searched. This allows files in another directory that have the same name as those in the current directory to be accessed. The default is NOXDIRECTORY. The class is invocation line only.

Chapter 3

CROSS-LINKER (LNCOP)

3.1 INTRODUCTION

This chapter describes the operation of the COP8 Linker, LNCOP. LNCOP reads object modules produced by ASMCOP and combines them into an absolute object file that may be executed on the processor. This file is also suitable for use by a debugger or emulator.

The object modules may reside in a file or be obtained from a library produced by LIBCOP. See Chapter 4 for detailed information on the library.

A load map is also produced that shows how memory is allocated. Memory allocation is under complete control of the user. In addition, a symbol table or cross-reference table may be obtained.

3.2 INVOCATION AND OPERATION

This section discusses invocation of LNCOP, the default configuration file, search order for library and help files, and default filenames and extensions. See the release letter for installation instructions.

3.2.1 Invocation

The invocation of the Linker for the MS-DOS system is as follows:

LNCOP (gives help)

LNCOP **[***options***]** *objfile***[***,objfile***[***,...***]] [***options***]**

where: *objfile* is the name of an object module or library to link. Multiple files may be linked by joining them with a ",". Default extension is .obj.

The Linker determines the file type from the file contents. Only those modules in the library that satisfy undefined externals are loaded. Thus, the order of the libraries when used in this manner is important. To use a library, the complete filename is given. No default extension is assumed. For another way to specify libraries, see the LIBFILE command, which searches standard directories.

options is an optional list of Linker commands described in Section 3.2.3. If no options are specified, the Linker defaults, described in Section 3.6, are used.

NOTE: MS-DOS supports *@cmdfile*, where *cmdfile* contains additional invocation line object filenames and/or options. This file has a default extension of .CMD. For example, if the file *a.cmd* contains:

```
test
/table
```

and file *b.cmd* contains

```
/o=testdata
```

then the command

```
LNCOP @a /e @b
```

is equivalent to

```
LNCOP test /table /e /o=testdata
```

Also see FILE control, for alternative to cmdfiles.

Any error detected on the invocation line causes the Linker to stop execution and display the part in error with an appropriate message.

The following are sample invocation lines for an MS-DOS system:

```
LNCOP  TEST  /NOOUTPUTFILE /TABLE /MAP=CON

LNCOP  MOD1,MOD2  /CROSS /BRIEF /MAP

LNCOP  \DEMO\SAMPLE /SECT data=0x40
```

The first command line links the file *test.obj* and outputs the load map with a symbol table to the console. No output object file is produced.

The second command line links the object modules *mod1.obj* and *mod2.obj* and places the output object file into *mod1.cof*. A cross-reference table is produced and the brief form of the map is used. The map is written to file *mod1.map*.

The next example links the file *sample.obj*, which resides in directory *demo*. It produces *sample.cof* in the current directory. A section, called data, in the object module is assigned a starting address of 0x40.

3.2.2 Default Configuration File

Upon startup, the Linker reads a default configuration file named *lncop.cfg*. This file search order is the same as the help file order search (see Section 3.2.6). If it is missing, a warning message is issued and the following “built-in” default configuration is used:

```
Format=cof
```

The default ranges are specified by the .CHIP directive in ASMCOP. These defaults can be changed to match user requirements. The default configuration file uses the same commands and syntax as a /file file (see /file option).

Example: Special configuration file to specify ranges for a future chip.

```
Format=cof
Range=BASE=(0x00:0x0F)
Range=RAM=(0x0:0x7F)
Range=EERAM=(0x80:0xBF)
Range=REG=(0xF0:0xFB)
Range=SEG=(0x100:0x17F,0x200:0x27F)
Range=SEGB=(0x100:0x10F, 0x200:0x20F)
Range=ROM=(0x0:0x1FFF)
```

3.2.3 Linker Options

An option is a Linker command specified on the invocation line.

The Linker invocation line options for MS-DOS systems start with a /, which may be preceded and followed by a blank space. Linker options are not case-sensitive and may be abbreviated to the minimum number of characters as specified in the command descriptions. For example:

```
/CROSSREF / MAP
```

See Section 3.6 for all Linker commands.

3.2.4 Default Filenames and Extension

For those options that require a filename, you may specify a filename with a full path name, or you may specify just a directory path; in this case, the default filename is used with that directory. The default filename is the name of the first object file specified with any extension removed.

For MS-DOS systems, a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists only of a directory, it should be terminated by a "\" on MS-DOS systems. If not, it is treated as a filename. Thus, /M=txt\ outputs the map to file *txt\cat.map* (assuming *cat.obj* is the input file). /M=txt outputs the map to *txt.map*.

3.2.5 Library File Search Order

When searching for a file specified by the /LIBFILE option, directories are searched in the following order:

1. current directory
2. directories specified by the /LIBDIRECTORY option
3. default directories
    a. directory specified by environment variable LNCOP, if it exists
    b. directory specified by environment variable COP, if it exists
    c. directory \COP

If the /X option is specified, only directories in category 2 above are used.

Any filename that contains an explicit directory is checked for only in that directory. No other directories are searched.

3.2.6 Help and Configuration File Search Order

When searching for the help file, *LNCOP.HLP,* and configuration file, *LNCOP.CFG,* directories are searched in the following order:

1. current directory
2. default directories (as noted in Section 3.2.5).

3.2.7 Temporary File Directory

Temporary files are generated in the current directory, unless the environment variable TMP specifies another directory, e.g., DOS command: set TMP=d:\. It is recommended to specify TMP as a directory on a RAM drive, of size 256K.

3.2.8 Error Level Return

If no errors occur, an error level of zero is returned. If errors occur, a nonzero error level is returned (warnings are not considered errors).

3.3 MEMORY ALLOCATION AND LOAD MAP

The Linker places each section in memory based on the attributes of the section and the memory that is available, which is specified by the RANGE command(see Section 3.6.16) or by default. Each section has the following attributes:

memory type	BASE, RAM, EERAM, REG, SEG, SEGB, or ROM (see Section 2.9.32, .SECT directive)
size	determined from object modules
absolute	section is specified as absolute in assembler
fixed	starting address is specified by the SECT command(see Section 3.6.17)
ranged	memory range is specified by the SECT command

Memory is allocated section by section. Typically, there is no way to control the order in which sections are allocated. However, sections are processed in the order in which they become known to the Linker. A section is known when it appears in a SECT command or is processed in an object module.

Sections are allocated in the following order:

1. Each absolute or fixed section is placed in memory at its specified address. This includes placing the start section at zero. The memory required for the section must be a part of memory made available through the RANGE command(see Section 3.6.16).

2. Each ranged section is placed in memory within the specified range. This range must lie within a portion of memory specified by the RANGE command.

3. All remaining sections are allocated as follows: As each section is processed, the ranges for its memory type are examined to find enough free space to allocate the section. Each range is examined in order. The first space large enough to contain the section is used. At this point, the memory allocated is marked “used.” If not enough memory is available to allocate the section, an error message is displayed. INPAGE sections must be placed between address xx00 and xxFE (see assembly .SECT directive, Section 2.9.32). For SMALL and MEDIUM memory models, ROM sections must not cross 4K boundaries (see assembly MODEL control, Section 2.10.27).

The load map shows the following information (see Section 3.4 for example):

- The Range Definitions show the memory ranges specified by the /RANGE option (Section 3.6.16) or by the default.

- The Memory Order Map shows the starting and ending addresses of each contiguous range of memory. It also indicates the type of memory.

- The Memory Type Map shows how memory is allocated but is organized by the type of memory. Within each type, the allocation is shown in memory order.

- The Total Memory Map shows allocation of all ROM and all RAM.
- The Section Table, which is listed by the /NOBRIEFMAP option(see Section 3.6.1), shows each section in the link, along with its starting and ending address. The section attributes are also displayed. The modules that comprise each section are displayed under the section names along with its addresses.
- Display of:
  - — Checksum of all ROM bytes
  - — Number of ROM bytes used
  - — Output filename
  - — Memory model
  - — Chip family name

If there is an overlap between sections, this is indicated in the Memory Order Map by the message `** memory overlap **` next to the overlapping section(s).

If sections do not fit in memory, their address is shown in the Section Table as "_ _ _ _." Also, no object code is generated for these sections.

3.4 LINKER EXAMPLE

The following is a very simple example. Two modules were assembled and their object module was processed by the Linker. First, the two source programs are shown followed by the Linker commands used to process them, followed by the Linker outputs.

Assembly file 1. *SAMPLE1.ASM*

```
.chip 820
;public code label
.public p1
;external data
.extrn
regdata:reg,ramdata:ram
;local base data
.sect otherdata,base
basedata: .dsb 1
;routine
.sect code,rom
p1:
ld a,regdata
add a,ramdata
add a,basedata
ret
.end
```

Assembly file 2. *SAMPLE2.ASM*

```
.chip 820
;public data
.public regdata,ramdata
;external code label
.extrn p1:rom
.sect data,reg
regdata: .dsb 1
.sect moredata,ram
ramdata: .dsb 1
;start of program
.sect code,rom
start:
jsr p1
jp .
.end start
```

File 1 has a module name of SAMPLE1, while file 2 has one of SAMPLE2. It is important to give each assembly module a different name, since this is used in the Linker load map. The default is to use the filename.

The following commands are used to generate the COFF file. Note that local option of assembler is used to show all non-public symbols in the Linker cross-reference.

ASMCOP sample1/local

ASMCOP sample2/local

LNCOP sample1,sample2 /crossref

The Linker outputs the following load map(see Section 3.3 for description):

```
-- Range Definitions --

BASE    0000:000F
ROM     0000:03FF
RAM     0000:002F
RAM     REG
REG     00F0:00FB
REG     00FF:00FF

-- Memory Order Map --

   Code Space
0000  000B   ROM

   Data Space
0000  0000   BASE
0001  0001   RAM
00F0  00F0   REG



-- Memory Type Map --

BASE
  0000  0000
  [size = 0001]

RAM
  0001  0001
  [size = 0001]

REG
  00F0  00F0
  [size = 0001]

EERAM
  [size = 0000]

SEG
  [size = 0000]

SEGB
  [size = 0000]

ROM
  0000  000B
  [size = 000C]
```

```
-- Total Memory Map --

TOTAL RAM = BASE + RAM + REG + EERAM + SEG + SEGB
  0000  0000
  0001  0001
  00F0  00F0
  [size = 0003]


TOTAL ROM = ROM
  0000  000B
  [size = 000C]


-- Section Table --

start end    attributes       Section
                                Module

0000  0000   BASE  BYTE       OTHERDATA
0000  0000                      SAMPLE1
0000  000B   ROM   BYTE       CODE
0000  0002                      SAMPLE2
0003  000B                      SAMPLE1
00F0  00F0   REG   BYTE       DATA
00F0  00F0                      SAMPLE2
0001  0001   RAM   BYTE       MOREDATA
0001  0001                      SAMPLE2


basedata  0000 Byte BASE  Local
    -SAMPLE1
 p1        0003 Null ROM
    -SAMPLE1    SAMPLE2
 ramdata   0001 Byte RAM
    -SAMPLE2    SAMPLE1
 regdata   00F0 Byte REG
    -SAMPLE2    SAMPLE1
 start     0000 Null ROM   Local
    -SAMPLE2

Checksum:     0x05D0
Byte Count:   0x000C (12)
Output File:  sample1.cof
Memory Model: Small
Chip:         820
```

3.5 LINKER ERRORS

Table 3-1 lists of the loader messages. These are divided into command line errors, link errors, link warnings, and object module errors. Object module errors are caused by a bad object file; you should check the filename and/or assemble and link again.

Table 3-1 Linker Errors
(Sheet 1 of 5)

Error	Description
Command Errors	
Invalid Command	The command specified is an invalid Linker command.
Ambiguous Command	This command abbreviation can be more than one command.
File Not Found	The object or library file on the invocation line cannot be found. Maybe the wrong extension is assumed or the file is in a different directory.
Invalid File Name	The file specified is an invalid filename.
Invalid Section Name	The section name on the SECT command is invalid.
Invalid Numeric	The number contains a digit invalid for the radix.
Missing Operand	The operand not found.
Invalid or missing Memory Type	Memory type must be ROM, RAM, BASE, EERAM, REG, SEG, or SEGB.
Invalid or missing range	Bad argument on the range command.
Low address range > High address	The lower address must be given first.
Invalid Object Type or Option	Invalid FORMAT option.
“NO” not valid for command	Command cannot use NO.
No Object file specified	No file found.
File Conflict	Output file matches an input file. Change output filename option.
Can't nest indirect files	Don't nest cmdfiles.
Expected an option	Missing “/” sign on the command line.
No Symbol specified	No symbol was specified on an EXTRACTSYMBOL command.
Invalid Symbol	Invalid symbol was specified on an EXTRACTSYMBOL command.
Not Valid for Absolute Section	Can't use /sect option for Absolute section.
Symbol/Module not found in library(s)	Can't find Symbol/Module specified by EXTRACT or EXTRACT-SYMBOL command.

Table 3-1 Linker Errors
(Sheet 2 of 5)

Error	Description
Files Nested Too Deep	/File files may be nested to depth of 100.
File Conflict for Post Processing Program	The name of the /output option will cause an error during Post Processing.
No such file or directory	Wrong syntax for /outputfile option.
File is not a library	Incorrect library name is specified.
Can't find Help File	LNCOP does not find file lncop.hlp.
Link Errors	
Public, external byte, word type mismatch	External does not match public in byte or word type.
Duplicate PUBLIC Symbol	The symbol shown has already been defined as a PUBLIC in a previous object module.
No .END Address has been Specified	No input module has a reset address (see the assembler .END directive).
Section Mismatch	A section with the same name from two different modules. Reassemble one of the modules after changing the attributes or rename the section.
Base page reference not on basepage	Result of basepage expression >0F Hex.
Undefined external	No public definition for external symbol.
Undefined Section	The specified section has been used on a SECT command but never appeared in an object module. Probably a misspelled name.
No Ranges Left For Section	There is not enough memory available via the Range definitions to allocate all object modules (see load map).
Multiple .END Addresses have been specified	Only 1 reset address is allowed (see the assembler .END directive).
File is not a Library	Bad filename or file corrupted.
Conflicting .CHIP name	The .CHIP strings in different modules must agree (see Section 2.9.4).
Divide by 0	External or relocatable expression contains divide by zero.
File Conflict for Post Processing Program	The name of the /output option will cause an error during Post Processing (i.e., processing by PROMCOP).

Table 3-1 Linker Errors
(Sheet 3 of 5)

Error	Description
Can't find program	The program required to post process the COFF output file was not found in the current directory, the directory LNCOP is in, or a directory in the PATH.
Error accessing program	There is some error executing the program required to post process the COFF output file.
Start Address Must be place at zero	Start address from assembler must be placed at zero.
No code at ROM location zero	ROM section should have at least one instruction.
Overlapping Memory	Have overlapping memory for different section type or for different sections with the same memory type.
JMP across 4K boundary	Can't jump >4K for relocatable sections with non-large model.
ROM must be less than 0x8000	Don't exceed the maximum address of ROM.
Data in range 0xC0-0xEF	Reserved range.
Data in range 0xFC-0xFE	Don't overwrite registers A, B, X, or SP.
BASE must be in range 0x0-0xF	BASE must be in that range.
RAM must be less than 0x100	RAM must be in that range.
REG out of range 0xF0-0xFF	REG must be in that range.
SEG must be greater than 0xFF	SEG must be in that range.
EERAM must be in range 0x80-0xBF	EERAM must be in that range.
Object Module not Valid for processor	Don't use for LNCOP an object file created by ASMHPC.
Chip Other than Anycop must be specified	We can't link files that assembly only with /chip=Anycop.
Section crosses 4K boundary	Error should be given if section placed across 4K boundary in non-large model.
Memory Full, Program Terminated, Type:	Too many symbols, models, etc.
Base or register value out of range	Don't use wrong value for Base or register variable.
External BASE or SEGB value not in range xx00 to xx0F	Symbol defined as BASE or SEGB type must be in that range.

Table 3-1 Linker Errors
(Sheet 4 of 5)

Error	Description
External REG value not in range 0xF0 to 0xFF	Symbol defined as REG type must be in that range.
Link Warnings	
Public, external byte, word type mismatch	External does not match public in byte or word type.
BASE public used as non-BASE external	The public symbol declared in the BASE section, not used as BASE external. This may generate inefficient code.
BASE external used with non-BASE	The usage of external may generate a link error.
Absolute section defined with SECT command	The SECT command is ignored for absolute sections.
Default configuration file not found	The file *lncop.cfg* is not in search path (see Section 3.2.6).
Mixed Memory Models	Object files were created with different assembly MODEL control. Largest model used.
Public and extrn do not match in section type	Don't use the same variable as public and extrn with different section type.
BASE/SEGB External used with non-BASE/SEGB public	Mixing BASE/SEGB and non-BASE/SEGB section type.
BASE/SEGB public used with non-BASE/SEGB external	Mixing BASE/SEGB and non-BASE/SEGB section type.
REG external used with non-REG PUBLIC	Mixing REG and non-REG section type.
REG public used as non-REG external	Mixing REG and non-REG section type.
Object Module Errors	
Bad Object Module	This file is not recognized as a valid object module created by the assembler. Probably the wrong filename.
Record Disp out of Range	Bad relocation record. Probably bad file.
Invalid Relocation CMD	Bad object module relocation record. Probably bad file.
Missing Module Header	The first record of the object module is not valid. Perhaps the wrong file was read.

Table 3-1 Linker Errors
(Sheet 5 of 5)

Error	Description
Multiple Headers	The object module contains two header records. Must be a corrupted file.
SYNC Error	Something happened to this file or the system between pass 1 and pass 2 of the loader. Try to load program again.
Bad Section ID	A section ID in the object module is out of range.
Record out of order	A record in the object module is in the wrong place.
Checksum Error	An object record has a checksum error. Reassemble the file.
Invalid Record Type	An object module is invalid.
Read Past Record	The record length didn't correspond with the information record.
Bad Object Symbol	A symbol in the object module is invalid.
Bad Library File	A library file contains invalid information. Recreate the library.

3.6 COMMANDS

This section describes the Linker commands, giving the command name, arguments, and default state. Table 3-2 is a summary of the Linker commands discussed in this section.

All commands may be given an invocation line, command file (@file), or in linkfile (/FILE) file. Commands on invocation line or in @file start with /.

Table 3-2 Summary of Linker Commands

COMMAND	FUNCTION	
[NO]BRiefmap	[NO]Brief load map	Section 3.6.1
[NO]Crossref	[NO]Cross-reference table in output map file	Section 3.6.2
[NO]Debug	[NO]Debug symbols to object	Section 3.6.3
[NO]Echo	[NO]Echo command file	Section 3.6.4
Extract=(*lib=module*)	Extract modules from libraries	Section 3.6.5
Extractsymbol=(*lib=symbol*)	Extract modules from libraries based on symbol name	Section 3.6.5
FIle=*linkfile*	Specifies linkfile command file	Section 3.6.6
Format=*type*	Specifies format of output object module	Section 3.6.7
[NO]Ignoreerrors	[NO]Force object file	Section 3.6.8
LIBDirectory=*directory* LD=*directory*	Search other directories for libraries	Section 3.6.9
LIBFile=*file* LF=*file*	Library files	Section 3.6.10
Load=(*files*)	Load modules	Section 3.6.11
[NO]LOCALSymbols	[NO]Assembly local symbols to map	Section 3.6.12
[NO]Mapfile[=*file*]	[NO]Map file	Section 3.6.13
[NO]Outputfile[=*file*]	[NO]Absolute object module	Section 3.6.14
PWidth=*width*	Set width of map file	Section 3.6.15
Range=*type=(ranges*)	Specifies ranges for section type	Section 3.6.16
Sect=*section=addr*	Specifies address or range of section	Section 3.6.17
SIzesect=*section=size*	Specifies section size	Section 3.6.18
[NO]Tablesymbols	[NO]List symbol table	Section 3.6.19
[NO]Warnings	[NO]Output warning messages	Section 3.6.20
[NO]Xdirectory	[NO]Search only LIBDirectory directories	Section 3.6.21

3.6.1 Briefmap — Set Map Format

Syntax: [NO]BRiefmap

Description: This command allows you to get a full or brief load map. BRIEFMAP removes the “Section Table” (see Section 3.3) from the load map.

A full map (NOBRIEF) consists of the full map, as shown in Section 3.4. Default is NOBRIEFMAP.

3.6.2 Crossref — Cross-Reference

Syntax: [NO]Crossref

Description: CROSSREF causes a cross-reference table to be included in the output map file. This table consists of each symbol and its value along with the names of the module in which it is defined and all the modules in which it is used. Non-global symbols are shown in the table, if passed by the assembler (see Section 2.10.18). The module, in which the symbol is defined, is preceded by a "-" on the listing. Default is NOCROSSREF.

CROSSREF overrides TABLESYMBOLS. A cross-reference listing is not generated if NOMAPFILE is in effect.

NOTE: See note on LOCALSYMBOLS, Section 2.10.18.

3.6.3 Debug — Debug Symbols

Syntax: [NO]Debug

Description: This command passes debugger symbols selectively by module to the COFF file. Therefore, it is not necessary to recompile and assemble to control which modules pass symbols to the COFF file. Default is DEBUG.

Example: LNCOP file1.obj,file2.obj, /NODEBUG file3.obj, /DEBUG file4.obj

If all objects are built with symbols (/sym option for compiler or assembler) then only symbols for file1, file2, and file4 are passed to the COFF file.

3.6.4 Echo — Echo Command Files

Syntax: [NO]Echo

Description: ECHO displays all lines read from an MS-DOS command file (@filename) or linkfile (/File=filename). ECHO only applies to commands that follow it. The default is NOECHO.

3.6.5 Extract, Extractsymbol — Extract Module from Library

Syntax: Extract=(library [,library,...] =module[,module,...])
Extractsymbol=(library[,library,...] =symbol[,symbol,...])

Description: For EXTRACT, the name of each module in the library is checked against those specified on the command; if a match is found, that module is loaded. For EXTRACTSYMBOL, each PUBLIC symbol in each module in the library is checked against the symbols specified in the command; if a match is found in a module, that module is loaded.

If the only modules linked are obtained from EXTRACT commands, then the default file names for output files are obtained from the first EXTRACT library.

Example:
```
; get some trig routines
EXTRACT= (mathlib=sine,cosine,tan)
; extract the module containing the symbol buffer which
; was created by the compiler
EXTRACTSYMBOL=(mylib = _buffer)
```

NOTE: EXTRACT and EXTRACTSYMBOL force a module to be linked even if it is not required to satisfy any externals, unlike LIBFILE.

3.6.6 File — Specify Linkfile

Syntax: FIle = *linkfile*

Description: The FILE command specifies the name of a linkfile from which the Linker commands are read. The default extension is .FIL.

Linkfiles are recommended as an alternative to command files (@file). The main advantage of linkfiles is that the commands are operating system independent.

Commands in a linkfile are specified without a leading /, and only one command per line. Comment lines may start with ;.

Linkfiles may be nested to a depth of 100.

Example: FILE = link.fil

where link.fil is

```
FORMAT=HEX
LIBFILE=LIBRARY
;RANGES
RANGE=ROM=(0:02000)
RANGE=BASE=(0:0f)
RANGE=RAM=(0:07F)
RANGE=REG=(0F0:0FB,0FF)
```

3.6.7 Format — Specify Output Format

Syntax: Format = *type* [= *options*]

Description: This command specifies the format of the output object module. *Type,* along with *options,* may be one of the following:

lm	for National Semiconductor load module format.
hex [=[no]fill[=*value*]]	for an Intel hex format. By default, unused bytes are filled with zero. Nofill or fill value can be optionally given.
coff [=strip]	for a COFF (Common Object File Format) format. A COFF object module may contain symbols and, thus, is normally used for symbolic debugging purposes. The default is to put the symbolic information in the object module. The optional **strip** argument may be used to keep the information out of the object module.

Any number of characters that uniquely identify the command arguments may be used for an argument. Thus **hex** can also be specified as **he** or **h**. The format type may be specified in either upper or lower-case.

Except for lm format, a COFF file is always generated, then converted to another format if appropriate.

Example:
```
F=hex
F=coff=strip
```

3.6.8 Ignoreerrors — Force Object File

Syntax: [NO]Ignoreerrors

Description: IGNOREERRORS forces an output file whenever possible even if link errors occur. This command should be used with caution because the output file may not execute properly; carefully note all errors. Default is NOIGNOREERRORS.

3.6.9 Libdirectory — Specify Library Search Directory

Syntax: LIBDirectory = *directory*, LD = *directory*

Description: Normally, when the LIBFILE command is used, the program looks in the current and default directories to find the library (refer to Section 3.2.5). If not found, it flags an error. This command enables the program to search other directories to find the file. A file that has an explicit directory is checked only in that directory, no others are searched.

Multiple directories may be searched; each is specified with a separate LIBDIRECTORY command.

Example:
```
LD=C:\      ; check root directory on drive C:
LD=..       ; check parent directory
```

NOTE: A LIBDIRECTORY command only specifies library search directories for the LIBFILE commands which follow it on the invocation line (or in command file).

3.6.10 Libfile — Specify Library File to Search

Syntax: LIBFile = *library*, LF = *library*

LIBFile=(*library* [,*library* . . .][=*symbol, symbol, . . .*])

Description: This command specifies library files that are used to resolve any undefined externals. If a library module contains a PUBLIC symbol that matches an unresolved external, that module is automatically loaded by the Linker. The library search is performed at the end of the linking process. Multiple libraries are searched in the order specified. If optional symbols are specified on command, then libraries are only searched for those symbols.

To search a library at some other point in the load process, the library file must be specified on the invocation line, as described in Section 3.2.1.

The default extension for a library file is .lib.

Example: LF=MATH /LF=FLOAT

LIBFILE=(MATH, FLOAT = SIN, COS)

3.6.11 Load — Load Object File

Syntax: Load=(module[,module, . . .])

Description: This command loads the specified object files. If the file specified is a library, then any modules within the library that satisfy an entry point are loaded at this time.

This command is only needed in a linkfile (see FILE command), because filenames may be specified alone on the invocation line.

Example:
```
; this is a linkfile, note that parentheses are not required around files
; load utility programs into bank
LOAD=BINHEX,ASCBIN,FLOAT
; load some others
LOAD=HELPER
```

3.6.12 Localsymbols — Assembly Local Symbols

Syntax: [NO]LOCalsymbols

Description: This command passes assembler local symbols selectively by module to the Linker cross-reference appearing in the .MAP file. Therefore, it is no longer necessary to reassemble to control which modules pass these symbols to the Linker cross-reference.

Example: LNCOP file1.obj,file2.obj /NOLOCAL file3.obj /LOCAL file4.obj /CR

If all objects are assembled with LOCALSYMBOL option (see assembler LOCALSYMBOL option in Section 2.10.18), then only the assembler local symbols for file1, file2, and file4 are passed to the cross-reference in the .MAP file.

3.6.13 Mapfile — Specify Map File

Syntax: [NO]Mapfile [= *file*]

Description: *File* specifies the name of the file to which the load map and any symbol table or cross-reference table will be written. If NOMAPFILE is specified, then no map is produced. MAPFILE without a filename causes the map to be written to the first object filename with an extension of .map. The default is MAPFILE. The default filename extension is .MAP.

The BRIEF command is used to specify the kind of map.

Example:
```
NOMAP
M=test      ; map to file test.map
```

3.6.14 Outputfile — Specify Output Object File

Syntax: [NO]Outputfile [=*file*]

Description: *File* specifies the name of the file to which the absolute object module, produced by the Linker, will be written. If NOOUTPUTFILE is specified, then no file is produced. The default is OUTPUTFILE with default filename.

The format of the output file as well as its default extension is given by the FORMAT command. OUTPUTFILE without a filename causes the output to be written to the first object filename, with an extension depending upon the FORMAT command. A format of LM uses an extension of .lm, a format of COFF uses .cof as an extension, and a format of HEX uses .hex as an extension.

Example:
```
O           ; default output file
O=test.abs  ; output to file test.abs
```

3.6.15 Pwidth — Specify Width of Map File

Syntax: PWidth = *number*

Description: This control sets the number of characters per line that are printed in the map file. Characters beyond this position are ignored. This is also used as the position to which the date is aligned in the header. Values are from 64 to 255. Although some terminals are 80 characters wide, writing a CR-LF in the 80th spot causes a subsequent blank line to appear. In this case, the user should use 79. The default is 79. The class is primary.

Example: PW=100

3.6.16 Range — Specify Memory Ranges

Syntax: Range=*memtype= ranges*

where:

memtype indicates the memory type and may be BASE, RAM, EERAM, REG, SEG, SEGB, or ROM.

ranges is one or more memory ranges. Multiple ranges must be separated by commas. In MS-DOS, multiple ranges must be enclosed within parentheses. Each range must be in the form:

low address:high address

or

type

Description: The RANGE command allows you to indicate those areas of memory that are available to the program. The Linker attempts to place all sections, both relocatable and absolute, in this memory. Any memory not defined in a RANGE command cannot be used by the program.

If an address range is given, it implies that the “memtype” for the command may reside in this part of memory. *Type* specifies a memory type just like “memtype” and implies that the range may also consist of the ranges for the memory type given by *type*. The default is dependent on the chip.

Refer to Section 3.3 for additional information.

Examples:

1. RANGE=REG=(0xf0:0xf6,0xff)
2. R=RAM=(0x0:0x2f,REG)

Example 1 tells the Linker that REG code can reside only between 0xf0:0xf6 and 0xff.

Example 2 says to put any RAM data into the range 0x0:0x2f. If no room exists, then use the range specified for memory type REG.

NOTES: 1. A new RANGE command overrides a previous one, e.g.,

```
R=ROM=(0:01fff)
R=ROM=(0:0fff)
```

defines range 0:0fff. The keyword NULL may be used to remove previously defined range, e.g.,

```
R=ROM=(0:01fff)

R=ROM=(NULL)
```

disables ROM range.

2. For MS-DOS, inside a command file (*@file*), a RANGE command may be continued over several lines by placing a minus sign (–) at the end of each continued line. This is useful if there are a large number of ranges on one RANGE command. A line may ONLY be continued before or after a range, e.g., NOT in the middle of a range. A valid example is:

```
RANGE=ROM16=(0:0fff,-

01000:01fff,02000:02fff,03000:03fff,04000:04fff,–

05000:05fff)
```

3.6.17 Sect — Specify Section Address

Syntax: Sect=name= *addr*, Sect=name= *range*, Sect=name= *section*

Description: This command specifies a starting address or address range for the relocatable section "name." If *addr* is used, this section is placed in memory at the given address. If *range* is used, the section is allocated within that range of addresses. If a previous section name is given, the address or range used for that section is used for this one.

Example:

```
s=one=0x566             ; section one starts at 0x566
s=two=0x200:0x300       ; section two must reside in this range
S=three=two             ; section three is also between 0x200:0x300
```

NOTE: SECT command places sections independent of the range definitions. Refer to Section 3.3 for additional information.

3.6.18 Sizesect — Specify Section Size

Syntax: SIzesect=name=size

Description: This command allows you to specify the size of relocatable section "name." This section would typically be a stack, whose size would be accessed with the B_SECT (name) and E_SECT (name) assembly operators. All sections may only have their size increased.

The section is considered relocatable and is allocated like all other sections. Its starting address can be specified with the SECT command.

It is an error if this section does not exist in one of your object modules.

Example: SIZE=stack=0x40

3.6.19 Tablesymbols — Enable Symbol Table

Syntax: [NO]Tablesymbols

Description: NOTABLESYMBOLS specifies that no symbol table will be listed in the map file, whereas TABLESYMBOLS creates a symbol table listing. Non-global symbols are shown in the table, if passed by the assembler (see Localsymbols, Section 2.10.18). This command is overridden by the CROSSREF control. The default is NOTABLESYMBOLS.

A symbol table listing is not generated if NOMAPFILE is in effect.

NOTE: See note on LOCALSYMBOLS, Section 2.10.18.

3.6.20 Warnings — Display Warning Messages

Syntax: [NO]Warnings

Description: WARNINGS causes any warning messages to be included in the map file. NOWARNINGS inhibits the messages. The default is WARNINGS.

3.6.21 Xdirectory — Exclude Standard Directories

Syntax: [NO]Xdirectory

Description: Normally, when searching for a library file specified by LIBFILE, the current, default, and any directories specified via the LIBDIRECTORY command are searched. XDIRECTORY causes only those directories specified by the LIBDIRECTORY command to be searched. This allows files in another directory that have the same name as those in the current directory to be accessed. The default is NOXDIRECTORY.

NOTE: A XDIRECTORY command only applies to LIBFILE commands which follow it on the invocation line (or in command file).

Chapter 4

CROSS-LIBRARIAN (LIBCOP)

4.1 INTRODUCTION

This chapter describes the operation of the COP8 Librarian, LIBCOP.

LIBCOP reads object modules produced by ASMCOP and combines them into one file called a library. The Linker can then search a library for any public symbols that match undefined external symbols. If a symbol is found, the Linker reads its object module. Thus, one or more from a group of standard routines may be included in a program by forming the group into a library.

4.2 INVOCATION AND OPERATION

This section discusses invocation of LIBCOP, search order for help files, default filenames and extensions. See the release letter for installation instructions.

4.2.1 Invocation

The invocation of the Librarian, LIBCOP, for MS-DOS systems is as follows:

LIBCOP (gives help)

LIBCOP **[***options***]** *libfile* **[***name* **[**,*name*...**]] [***options***]**

where:

libfile is the name of a library to process. If it does not already exist, then a new library is created. Default extension is .lib.

name is a list of one or more object files that are processed by LIBCOP. Default extension is .obj.

options is a list of library commands described in Section 4.4.

NOTE: MS-DOS supports *@cmdfile,* where *cmdfile* contains additional invocation line object filenames and/or options. This file has a default extension of .cmd.

Any error detected on the invocation line causes the Librarian to stop execution and display the part in error with an appropriate message.

The following are sample invocation lines for MS-DOS systems:

```
LIBCOP FLOAT SINE,COSINE /REPLACE
LIBCOP TEST  DATASET /DELETE
```

The first command line replaces the modules SINE and COSINE in the library *float.lib* by object files *sine.obj* and *cosine.obj*.

The second command line deletes the module *DATASET* from the library *test.lib*.

4.2.2 Object Files and Module Names

An object file is generated by ASMCOP. This file usually has an extension of .obj. The module name by which the Librarian stores these files in a library is the same as the filename without the extension. Thus, an object file of *tangent.obj* has a module name of *tangent* within the library. Names of modules added to a library are case-sensitive.

4.2.3 Library Options

The invocation line options for MS-DOS systems start with a /, which may be preceded and followed by a blank space. Library options are not case-sensitive and may be abbreviated to the minimum number of characters as specified in the command descriptions. For example:

```
/ADD / Backup
```

See Section 4.4 for all the library commands.

4.2.4 Default Filenames and Extensions

Default extensions depend on the operating system and how the file is specified on the invocation line. For MS-DOS systems, a default extension is always placed on a file unless one is explicitly specified.

If an output filename consists of just a directory, it should always be terminated by a "\" on MS-DOS systems; if not, it is treated as a filename. Thus, /L=txt\e outputs the listing to file *txt\cat.lis* (assuming *cat.lib* is the input file). /L=txt outputs the listing to *txt.lis*.

4.2.5 Help File Search Order

When searching for a help file, directories are searched in the following order:

1. current directory
2. default directories
    a. directory specified by environment variable LIBCOP, if it exists
    b. directory specified by environment variable COP, if it exists
    c. directory \COP

4.2.6 Error Level Return

If no errors occur, an error level of zero is returned. If errors occur, a nonzero error level is returned (warnings are not considered errors).

4.3 LIBRARY ERRORS

Except for those messages listed under Command Warnings in Table 4-1, all other error conditions cause the Librarian to stop operation with no changes to the library file being operated on. For object errors, reassemble and try again.

Table 4-1 Library Errors

Error	Description
Command Errors	
Invalid Command	The command specified is an invalid Librarian command.
Expected an option	Missing "/" on the command line.
File Not Found	The object file cannot be found. Maybe a wrong extension is assumed or it is in a different directory.
Duplicate Module Name	An attempt to ADD a module to a library which already contains a module with this name.
File is not an Object Module	The specified file is not an object module.
File is not a Library	The specified file does not look like a library.
Already specified library operation	Only one library operation may be specified for each execution.
No Library Specified	No library has been specified on the invocation line.
No Library Operation specified	Need option.
No Modules to Operate on	An operation other than LIST has been specified, but no modules have been given on which to operate.
Duplicate PUBLIC symbol	The specified symbol duplicates a PUBLIC symbol in another module.
Cannot create library file	Library file or backup file cannot be created. Check filename.
"NO" not valid for command	Command cannot use NO.

Table 4-1 Library Errors

Error	Description
Object module not valid for processor	Object module is probably HPC object module.
Library not valid for processor	Library is probably HPC library.
Command Warnings	
Module not Found	An attempt to delete a module that is not in the library.
Object Errors	
Bad Object Record	Something is wrong with the object module file.
Checksum Error	An object record has a checksum error. Reassemble the file.
SYNC Error	File or system changed pass 1 to pass 2. Try to link program again.
Invalid relocation CMD	An object module relocation command does not have a valid value.

4.4 LIBRARY COMMANDS

This section describes the library commands. The descriptions show the command name, arguments, and default state. Table 4-2 is a summary of the library commands.

Only one of the primary commands, ADD, DELETE, LIST, REPLACE, or UPDATE may appear on the invocation line at a time.

Table 4-2 Summary of Library Commands

Command	Function	
Add	Place specified object file(s) in library	Section 4.4.1
[NO]Backup	[NO]Backup library file	Section 4.4.2
Delete	Remove module from library	Section 4.4.3
[NO]Echo	[NO]Echo command file	Section 4.4.4
List[=*file*]	List all modules in library	Section 4.4.5
Replace	Replace specified object file(s) in library	Section 4.4.6
Update	Replace object file(s) with object file with a later date	Section 4.4.7

Table 4-2 Summary of Library Commands

Command	Function	
[NO]Warnings	[NO]Output warning messages	Section 4.4.8

4.4.1 Add — Add Object Module

Syntax: Add

Description: This command causes the Librarian to place the specified object file(s) into the library. If the module already exists in the library, an error is displayed. An object module has the default extension .obj.

Example: LIBCOP lib1 mod1,mod2,mod3 /add

This adds files *mod1.obj*, *mod2.obj* and *mod3.obj* to the library *lib1.lib*. The library is created if it doesn't already exist. The modules will have module names of *mod1*, *mod2*, and *mod3*.

4.4.2 Backup — Create Backup Library

Syntax: [NO]Backup

Description: The Librarian always creates a new library to avoid any possible damage to the old library. The old library is then renamed with a default extension of .bak. NOBACKUP can be used to request that no backup library be created. In any case, any error causes the library to remain unchanged. The default is BACKUP.

4.4.3 Delete — Delete Object Module

Syntax: Delete

Description: DELETE causes the object modules specified to be removed from the library. A warning is given if the module does not exist in the library.

Example: LIBCOP math add,sub /delete

NOTE: The module name is the same as the filename containing the object file without an extension.

4.4.4 Echo — Echo Command Files

Syntax: [NO]Echo

Description: ECHO displays all lines read from a command file. A command file is specified by *@filename*. ECHO only applies to commands that follow it. The default is NOECHO.

4.4.5 List — List Library

Syntax: List [= *file*]

Description: LIST displays a list of all modules in the library. This list is written to the specified file. If no file is given, it goes to the console. The default file extension is .lis.

4.4.6 Replace — Replace Object Module

Syntax: Replace

Description: This command tells the Librarian to replace the specified object file(s) in the library. If the module already exists in the library, it is replaced. If not, the new module is added to the library.

This command is similar to ADD, but it is not an error if the module exists in the library.

Example: LIBCOP lib1 mod1,mod2,mod3 /replace

4.4.7 Update — Replace Object Module if Newer

Syntax: Update

Description: This command tells the Librarian to replace the specified object file(s) in the library, if the new object file has a later date than the one in the library. A module that does not already exist in the library is added to it.

Example: LIBCOP lib1 mod1,mod2,mod3 /update

4.4.8 Warnings — Display Warning Messages

Syntax: [NO]Warnings

Description: WARNINGS causes any warning messages to be included in the output listing. NOWARNINGS inhibits the messages. The default is WARNINGS.

Chapter 5

COFF DISPLAY UTILITY (DUMPCOFF)

5.1 INTRODUCTION

DUMPCOFF is a utility program used to format and display the information in a Common Object Format File (COFF) file. This is the type of file output by LNCOP, the COP8 Linker, and used by the COP8 Debugger program.

The typical user will want to examine the object code, symbols, or line number information in the COFF file. However, command line options are provided to display more detailed information. This could include COFF header information, symbols used internally by the tool set, and others. In some cases, the information displayed requires an understanding of the internal COFF format (see the *Specification for COFF for the National Semiconductor HPC and COP8 Microcontroller*s, available from National Semiconductor Corporation).

5.2 OPERATION AND INVOCATION

The invocation line is as follows:

DUMPCOFF (gives help)

DUMPCOFF [*options*] *coffile* [*options*]

where: *coffile* is a Linker COFF file to process. The default extension is .cof.

options is one or more of the following program options:

/b /e /h /l /s /t

5.2.1 Options

Options may be specified before and/or after the COFF filename. Each option consists of a "/" followed by the option letter. Multiple options need not be separated by blanks, e.g., /t/s is valid. The option letters may be upper or lower case. The following are sample invocation lines:

DUMPCOFF /e graphics

DUMPCOFF control /T /S

DUMPCOFF /b/h system

/b

The /b option displays the .BNKINFO section. This is a special COFF section created by the Linker. It includes information on the sections, modules, and ranges used in the program. There is always one bank called SHARED.

If the /h option is also used, the data for this section is displayed as raw data.

Note, a section in the COFF file has no relationship with a program section created by the compiler or assembler.

/e

The /e option enables all the other options except the /h option. Thus /e is equivalent to specifying /b/l/s/t on the command line.

/h

The /h option displays information that is usually "hidden;" it consists of information that would be of use to people writing programs that process a COFF file. The information displayed requires some knowledge of the internal COFF format.

/l

The /l option displays the line number entries. Line number entries only exist if the assembler generated them using the appropriate command line options.

/s

The /s option displays symbolic information contained in the file. When used with the /h option, assembler-generated symbols that are not normally included are also displayed.

/t

The /t option causes the data bytes contained in each COFF text section to be displayed in both hex and ASCII.

5.2.2 Error Level Return

If no errors occur, an error level of zero is returned. If errors occur, a nonzero error level is returned (warnings are not considered errors).

5.3 EXAMPLES

The Linker example in Section 3.4 is used, except that /Sym option is used to generate symbolic information for COFF.

The programs were assembled with the following commands:

```
ASMCOP    sample1/sym

ASMCOP    sample2/sym
```

The programs were then linked as follows:

```
LNCOP     sample1,sample2
```

The following is the output of DUMPCOFF using various command line options as shown. Note that the link placed files into two different banks. The program signon header is not shown for the examples.

NOTE: The following examples may change somewhat due to future changes in Assembler, Linker, e.g., section names or code generated may change.

5.3.1 DUMPCOFF Sample1

With no options specified the default display is shown below. It consists of:

1. Name of the file.
2. The file header, which in this case shows only the date the file was created. If the /h option had been used additional information would have been shown.
3. The COFF section information. These sections which are different from sections in the assembler, consists of contiguous bytes of data starting at the address shown. *Paddr* and *Vaddr* are always the same. The section name is always .text.

```
File: sample1.cof
File Header
  Creation Date: Aug 28 13:24:02 1992

Section: .text
  Paddr: 0x00000000, Vaddr: 0x00000000
  Size of raw data: 0x0000000c
```

5.3.2 DUMPCOFF /b /s /t /l main

This set of options displays everything in the file except those items that are included by the /h option. You can do the same thing with the /e option. The display consists of:

1. Name of the file.
2. The file header, which in this case shows only the date the file was created. If the /h option was used, additional information is shown.

3. The bank information. This part of the output shows the program ranges and sections; one bank is called “SHARED.” The ranges declared during the link, or the default ranges if none were declared, are shown. These are identical to those shown on the Linker load map.

   Next, the section and modules are shown along with their start and end addresses, and section attributes. The attributes are the same as on the load map with the addition of the type of section. This can be CODE, or DATA.

   The module names are shown in the section they reside in. This is identical to the load map. If a module name is preceded by an “*,” it indicates that symbolic information is available for the module in the COFF file. This could be assembler generated symbolic information.

4. The COFF sections are displayed again, except this time the raw data for the section is also shown.

5. If there are any line numbers in the file, they are displayed after the first .text section as shown below. The line numbers are listed on a function by function basis. Note that the line numbers are relative to the start of the function, not an absolute line number of the file.

6. Finally, the symbol table is shown, listing the local function symbols. Local and global symbols are grouped separately and shown bank by bank.

```
File: sample1.cof
File Header
   Creation Date: Aug 28 13:24:02 1992

================================================================

Bank: SHARED
      ROM     0000:03FF
      RAM     0000:002F
      RAM     REG
      REG     00F0:00FB
      REG     00FF:00FF
      BASE    0000:000F

-- Sections --

start end    attributes           Section
                                    Module

0000  0000   BASE  BYTE  DATA       OTHERDATA
0000  0000                         *SAMPLE1
0000  000B   ROM   BYTE  CODE       CODE
0000  0002                         *SAMPLE2
0003  000B                         *SAMPLE1
00F0  00F0   REG   BYTE  DATA       DATA
00F0  00F0                         *SAMPLE2
0001  0001   RAM   BYTE  DATA       MOREDATA
```

```
0001  0001                          *SAMPLE2

Section: .text
   Paddr: 0x00000000, Vaddr: 0x00000000
   Size of raw data: 0x0000000c
Raw Data:
00000000 30 03 ff 9d f0 bd 01 84 bd 00 84 8e            0...p=..=...
Line Number Entries
   Function: .sect_CODE_1
         2 at address 0x00000003
         3 at address 0x00000005
         4 at address 0x00000008
         5 at address 0x0000000b
   Function: .sect_CODE_1
         2 at address 0x00000000
         3 at address 0x00000002

*** Symbol Table - Local Symbols

Bank: SHARED
File: sample1.asm
absolute static int .sect_CODE_1() with value 0x00000003
absolute static unsigned char basedata with value 0x00000000

File: sample2.asm
absolute static int .sect_CODE_1() with value 0x00000000
label start at address 0x00000000

*** Symbol Table - Global Symbols

Bank: SHARED

absolute extern unsigned char regdata with value 0x000000f0
absolute extern unsigned char ramdata with value 0x00000001
label p1 at address 0x00000003
```

5.3.3 DUMPCOFF /e /h main

This is the same as the previous example except that the /h option was specified to display any normally “hidden” information.

1. The file header now shows additional information. It indicates the number of sections in the file and the number of symbols. It also shows some flags, which are described in the COFF manual.

2. The optional header is shown next. This is also described in the COFF manual. The only fields in the optional header used by the COP8 software are the magic number, the version number, and the starting address.

3. On the first line of each bank entry, some index numbers are shown. These are the starting and ending symbol table index numbers for the local and global symbols, respectively. The symbol table index numbers are shown in the symbol table display. The final number preceded by a # is the index number of the start of any line number entries for this bank. A minus one indicates no entries.

4. For the section display, there is some additional information shown such as "Flags." This is described in the COFF manual.

5. For the symbol table entries, each entry is now shown with its symbol table index number on the left of each entry. These values are in hexadecimal. An index number containing a "+" indicates that this entry contained an auxiliary symbol entry which took up the next entry. Hence the next index number is two larger.

   There are some symbols shown in the symbol table, e.g., .bf and .ef, that did not appear in the previous example. These are assembler generated symbols used for debugging purposes. The use of these types of symbols are explained in the COFF manual.

```
File: sample1.cof
File Header
   Creation Date: Aug 28 13:24:02 1992
   Magic Number: COP8MAGIC (0542o)
   Number of Sections: 2
   Number of Symbols: 31
   Size Of Optional Header: 40 (bytes)
   Flags: RELFLG,EXEC,AR32WR
Optional Header
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   00 00 00 00 00 00 00 00                          ........

================================================================

Bank: SHARED   index:  0x0/0x1b  0x1c/0x1e  #0
      ROM     0000:03FF
      RAM     0000:002F
      RAM     REG
      REG     00F0:00FB
      REG     00FF:00FF
      BASE    0000:000F

-- Sections --

start end    attributes           Section
                                    Module

0000  0000   BASE  BYTE  DATA      OTHERDATA
0000  0000                        *SAMPLE1
```

```
0000  000B   ROM   BYTE  CODE      CODE
0000  0002                        *SAMPLE2
0003  000B                        *SAMPLE1
00F0  00F0   REG   BYTE  DATA      DATA
00F0  00F0                        *SAMPLE2
0001  0001   RAM   BYTE  DATA      MOREDATA
0001  0001                        *SAMPLE2

Section: .BNKINFO
   Paddr: 0x00000000, Vaddr: 0x00000000
   Size of raw data: 0x0000017c
   Flags: REG
   No line number records
Raw Data:
00000000 00 00 00 00 38 32 30 20 20 20 20 20 00 00 00 00  ....820     ....
00000010 00 00 00 00 b2 01 00 00 00 00 00 00 f8 01 00 00  ....2.......x...
00000020 00 00 00 00 53 48 41 52 45 44 00 00 02 00 00 00  ....SHARED......
00000030 ff 03 03 00 00 00 2f 00 03 05 00 00 00 00 05 00  ....../.........
00000040 f0 00 fb 00 05 00 ff 00 ff 00 01 00 00 00 0f 00  p.{.............
00000050 ff 00 00 00 00 00 00 00 53 41 4d 50 4c 45 31 00  ........SAMPLE1.
00000060 53 41 4d 50 4c 45 32 00 4f 54 48 45 52 44 41 54  SAMPLE2.OTHERDAT
00000070 41 00 43 4f 44 45 00 44 41 54 41 00 4d 4f 52 45  A.CODE.DATA.MORE
00000080 44 41 54 41 00 00 00 00 04 01 00 00 11 01 00 00  DATA............
00000090 00 00 00 00 f4 00 00 00 00 00 00 00 07 00 00 00  ....t...........
000000a0 00 00 00 00 00 00 00 00 0e 01 00 00 0a 01 00 00  ................
000000b0 00 00 0b 00 fc 00 00 00 00 00 02 00 0b 00 f4 00  ....|.........t.
000000c0 00 00 03 00 0b 00 07 00 00 00 00 00 00 00 00 00  ................
000000d0 00 00 13 01 00 00 15 01 00 00 f0 00 f0 00 fc 00  ..........p.p.|.
000000e0 00 00 f0 00 f0 00 0b 00 00 00 00 00 00 00 00 00  ..p.p...........
000000f0 00 00 18 01 00 00 13 01 00 00 01 00 01 00 fc 00  ..............|.
00000100 00 00 01 00 01 00 0b 00 00 00 00 00 00 00 00 00  ................
00000110 00 00 00 00 00 00 c0 00 00 00 c8 00 00 00 24 01  ......@...H...$.
00000120 00 00 00 00 00 00 00 00 00 00 1b 00 00 00 1c 00  ................
00000130 00 00 1e 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000150 00 00 00 00 00 00 00 00 00 00 00 00 1c 00 00 00  ................
00000160 1c 00 00 00 ff ff ff ff ff ff ff ff ff ff ff ff  ................
00000170 ff ff ff ff 1c 00 00 00 1d 00 00 00              ............

Section: .text
   Paddr: 0x00000000, Vaddr: 0x00000000
   Size of raw data: 0x0000000c
   Flags: REG
   Number of line number records: 8
Raw Data:
00000000 30 03 ff 9d f0 bd 01 84 bd 00 84 8e              0...p=..=...
Line Number Entries
   Function: .sect_CODE_1
         2 at address 0x00000003
         3 at address 0x00000005
         4 at address 0x00000008
         5 at address 0x0000000b
   Function: .sect_CODE_1
         2 at address 0x00000000
```

```
         3 at address 0x00000002

*** Symbol Table - Local Symbols

0:+Bank: SHARED
2:+File: sample1.asm
4:+absolute static int .sect_CODE_1() with value 0x00000003
6:+.bf with value 0x00000003
8:+ .bb with value 0x00000003
a:+ .eb with value 0x0000000c
c:+.ef with value 0x0000000c
e: absolute static unsigned char basedata with value 0x00000000

f:+File: sample2.asm
11:+absolute static int .sect_CODE_1() with value 0x00000000
13:+.bf with value 0x00000000
15:+ .bb with value 0x00000000
17:+ .eb with value 0x00000003
19:+.ef with value 0x00000003
1b: label start at address 0x00000000

*** Symbol Table - Global Symbols

   Bank: SHARED

1c: absolute extern unsigned char regdata with value 0x000000f0
1d: absolute extern unsigned char ramdata with value 0x00000001
1e: label p1 at address 0x00000003
```

Chapter 6

PROM UTILITY (PROMCOP)

6.1 INTRODUCTION

PROMCOP is a utility program that is used to convert the COFF file output by LNCOP, the COP8 Linker, into an Intel hex file for the purpose of burning a PROM. PROMCOP is directly invoked by LNCOP when the LNCOP /f=hex option is specified. PROMCOP can also be invoked independent of LNCOP, as explained below.

6.2 INVOCATION

The invocation line is as follows:

PROMCOP (gives help)

PROMCOP **[***options***]** *coffile* **[***options***]**

where: *coffile* is the linker COFF file to be processed. The default extension is .cof.

options is one or more of the following program options. An option can be abbreviated to any number of characters.

/f[ormat] = h[ex][=[no]fill[=*value*]]

/o[utput] = filename

NOTE: If no options are specified, a hex file of name *coffile.hex*, with unused bytes set equal to zero, is generated.

6.2.1 Options

/Format=hex

/f**[**ormat**][**=**]**h**[**ex**][**=**[**no**]**fill**[**=*value***]]**

This option indicates that one hex file is generated containing all the data bytes from the COFF file. The default output file name is *coffile.hex*. By default, unused bytes are filled with zero up to ROM range. The **nofill** option prevents filling unused bytes. The **fill=***value* option fills unused bytes with the specified byte value.

Output

/o[utput][=]filename

/output specifies the output filename. If the name is given without an extension, a default extension of .hex is used.

6.2.2 Error Level Return

If no errors occur, an error level of zero is returned. If errors occur, a nonzero error level is returned (warnings are not considered errors).

6.3 ERRORS

Can't allocate room for bank info
: There is not enough memory in your system to run the program.

Can't allocate room for section tables
: There is not enough memory in your system to run the program.

Can't create file
: Error trying to create an output file.

File not found
: The COFF input file or command input file can not be found. Possibly wrong file specified.

First section isn't .BNKINFO
: The first COFF section created by the linker containing range and section information was not found. Possibly wrong input file is specified.

Hit end of file on COFF file
: While reading the COFF file the end of file was reached. This indicates a corrupted file. Try generating the file again.

Input File contains no range information
: The COFF input file has already been processed by this program and hence contains no range or section information. Thus it cannot be processed; use original input COFF file.

Invalid COFF file
: The COFF input file does not appear to be a COFF file. Possibly you specified the wrong file.

Invalid or missing option
: An option given on the invocation line was not recognized. Must be /format or /output.

Invalid or missing format
: The /format option has an invalid argument.

Invalid File Name
: The COFF, /format, or /output file names are bad.

More than 1 COFF file specified
Only 1 file may be specified on the invocation line.

No COFF file specified
A COFF file must be given on the invocation line.

No sections in file
The COFF file is not valid. Probably not produced by the COP8 Linker.

Fill value > 255
Fill value is limited to 8 bits.

Chapter 7

HEXLM, LMHEX UTILITIES

7.1 INTRODUCTION

This chapter describes the utilities LMHEX and HEXLM. The utilities are provided so that you can convert NSC load modules into Intel-hex format object files and vice versa.

These utilities are of special interest to users who have data files in Intel-hex format and wish to use them with the microcontroller development system. HEXLM, for example, can be used to convert Intel-hex files to NSC LM format to be downloaded by COMM.

LMHEX converts NSC load modules to Intel-hex files. See Section 7.2.

HEXLM converts Intel-hex files to NSC load modules. See Section 7.3.

7.2 LMHEX

Syntax: LMHEX *lmfile* **[***.ext***]**

where:

	lmfile	is the filename (without extension) of the load module to be converted.
	ext	is the filename extension. The default extension is .lm.

The Intel-hex format output file created is named

lmfile.hex

7.3 HEXLM

Syntax: HEXLM *hexfile*[*.ext*]

where:

	hexfile	is the filename (without extension) of the Intel-hex file to be converted.
	ext	is the filename extension. The default file extension is .hex.

The NSC load module created is named

hexfile.lm

WARNING

HEXLM does **not** check that the Intel-hex file contains record type 02, such as generated by a program with address > 0xffff. An erroneous LM file is generated in this case.

Appendix A

ASCII CHARACTER SET IN HEXADECIMAL

Char.	7-bit Hex Number	Char.	7-bit Hex Number	Char.	7-bit Hex Number	Char.	7-bit Hex Number
NUL	00	SP	20	@	40	`	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	'	27	G	47	g	67
BS	_	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	-	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F	O	4F	o	6F
DLE	10	0	30	P	50	p	70
DC1	11	1	31	Q	51	q	71
DC2	12	2	32	R	52	r	72
DC3	13	3	33	S	53	s	73
DC4	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	\	5C	|	7C
GS	1D	=	3D	]	5D	}	7D
RS	1E	>	3E	↑	5E	~	7E
US	1F	?	3F	←	5F	DEL, rubout	7F

Appendix B

CHIP ARGUMENTS AND DEFAULT RANGES

The valid .chip directive and chip control arguments are listed below. Each argument maps to a chip family; this determines instruction set and default range. Please see release letter for new parts.

Table B-1 Chip Arguments for Each Chip Family

Chip Arguments	Device Type
820	820, 821, 822, 820C, 821C, 822C, 620, 621, 622, 620C, 621C, 622C
820CJ	820CJ, 822CJ, 823CJ, 620CJ, 622CJ, 623CJ, 920CJ, 922CJ, 923CJ, 87L20CJ, 87L22CJ
840	840, 841, 842, 840C, 841C, 842C, 640, 641, 642, 640C, 641C, 642C
840CJ	840CJ, 842CJ, 87L40CJ, 87L42CJ
87L40RJ	87L40RJ, 87L42RJ
8780	8780*, 8781*, 8782*
880	880, 881, 882, 880C, 881C, 882C
888BC	888BC, 884BC, 684BC, 87L84BC
888CF	888CF, 884CF, 87L88CF, 87L84CF
888CG	888CG, 884CG
888CL	888CL, 884CL, 688CL, 684CL, 87L88CL, 87L84CL
888CS	888CS, 884CS, 688CS, 684CS
888EB	888EB, 889EB, 87L88EB, 87L89EB
888EG	888EG, 884EG, 688EG, 684EG, 87L88EG, 87L84EG
87L88RG	87L88RG, 87L84RG
888EK	888EK, 884EK, 87L88EK, 87L84EK
87L84RK	87L88RK, 87L84RK
888FH	888FH, 884FH, 688FH, 684FH, 87L88FH, 87L84FH
888GD	888GD, 87L88GD
87L88RD	87L88RD
888GG	888GG, 87L88GG
87L88RG	87L88RG, 87L84RG
888GW	888GW
87L88RW	87L88RW
888HG	888HG, 688HG
888KG	888KG, 688KG, 87L88KG
8ACC	8ACC, 8ACC5, 8ACC7

Table B-1 Chip Arguments for Each Chip Family (Continued)

Chip Arguments	Device Type
8SAA	8SAA
8SAB	8SAB
8SAC	8SAC
912C	912C, 912CH

* Please refer to the data sheets of these devices for the correct configuration.

Table B-2 Default Ranges for Each Chip Family

Chip Family	Default Ranges (in Hex)					
	ROM	RAM	BASE	REG	SEG	SEGB
820	0:03FF	0:2F, REG	0:F	F0:FB,FF		
820CJ	0:03FF	0:2F, REG	0:F	F0:FB,FF		
840	0:07FF	0:6F, REG	0:F	F0:FB,FF		
840CJ	0:07FF	0:6F, REG	0:F	F0:FB,FF		
87L40RJ	0:7FFF	0:6F, REG	0:F	F0:FB,FF		
8780	0:0FFF	0:6F, REG	0:F	F0:FB,FF		
880	0:0FFF	0:6F, REG	0:F	F0:FB,FF		
888BC	0:07FF	0:2F, REG	0:F	F0:FB,FF		
888CF	0:0FFF	0:6F, REG	0:F	F0:FB,FF		
888CG	0:0FFF	0:6F, REG	0:F	F0:FB	100:13F	100:10F
888CL	0:0FFF	0:6F, REG	0:F	F0:FB,FF		
888CS	0:0FFF	0:6F, REG	0:F	F0:FB	100:13F	100:10F
888EB	0:1FFF	0:6F, REG	0:F	F0:FB	100:13F	100:10F
888EG	0:1FFF	0:6F, REG	0:F	F0:FB	100:17F	100:10F
87L88RG	0:7FFF	0:6F, REG	0:F	F0:FF	100:17F	100:10F
888EK	0:1FFF	0:6F, REG	0:F	F0:FB	100:17F	100:10F
87L88RK	0:7FFF	0:6F, REG	0:F	F0:FF	100:17F	100:10F
888FH	0:2FFF	0:6F, REG	0:F	F0:FB	100:17F2 00:27F 300:37F	100:10F 200:20F 300:30F

Table B-2 Default Ranges for Each Chip Family (Continued)

Chip Family	Default Ranges (in Hex)					
	ROM	**RAM**	**BASE**	**REG**	**SEG**	**SEGB**
888GD	0:3FFF	0:6F, REG	0:F	F0:FB	100:17F	100:10F
87L88RD	0:7FFF	0:6F, REG	0:F	F0:FF	100:17F	100:10F
888GG	0:3FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F	100:10F 200:20F 300:30F
87L88RG	0:7FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F	100:10F 200:20F 300:30F
888GW	0:3FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F	100:10F 200:20F 300:30F
87L88RW	0:3FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F	100:10F 200:20F 300:30F
888HG	0:4FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F	100:10F 200:20F 300:30F
888KG	0:5FFF	0:6F, REG	0:F	F0:FB	100:17F 200:27F 300:37F 400:47F 500:57F 600:67F 700:77F 800:83F	100:10F 200:20F 300:30F 400:40F 500:50F 600:60F 700:70F 800:80F
8ACC	0:0FFF	0:6F, REG	0:F	F0:FB		
8SAA	0:03FF	0:2F, REG	0:F	F0:FB		
8SAB	0:07FF	0:6F, REG	0:F	F0:FB		
8SAC	0:0FFF	0:6F, REG	0:F	F0:FB		
912C	0:02FF	0:2F, REG	0:F	F0:FB,FF		