

Tektronix®

8500
MODULAR MDL SERIES
ASSEMBLER
CORE USERS MANUAL
for B Series Assemblers



This manual supports the following TEKTRONIX products:

8550 Options	Products	8560 Options	Products
1T	8300B15	1A	8560B01
1U	8300B20	1B	8560B02
1V	8300B26	1C	8560B04
		1G	8560B10
		1J	8560B15
		1K	8560B16
		1L	8560B17
		1M	8560B18

This manual supports the following software modules:

TEKTRONIX B Series Assembler V 01 (8550)
TEKTRONIX B Series Linker V 01 (8550)
TEKTRONIX B Series LibGen V 01 (8550)
TEKTRONIX B Series Assembler V 01 (8560)
TEKTRONIX B Series Linker V 01 (8560)
TEKTRONIX B Series LibGen V 01 (8560)

These modules are compatible with:

DOS/50 V 02 (8550)
TNIX V 01 (8560)

**PLEASE CHECK FOR CHANGE INFORMATION
AT THE REAR OF THIS MANUAL.**

8500
MODULAR MDL SERIES
ASSEMBLER
CORE USERS MANUAL
for B Series Assemblers

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

070-3856-00
Product Group 61

Serial Number _____

First Printing AUG 1981
Revised FEB 1982

LIMITED RIGHTS LEGEND

Software License No. _____

Contractor: Tektronix, Inc.

Explanation of Limited Rights Data Identification Method

Used: Entire document subject to limited rights.


Those portions of this technical data indicated as limited rights data shall not, without the written permission of the above Tektronix, be either (a) used, released or disclosed in whole or in part outside the Customer, (b) used in whole or in part by the Customer for manufacture or, in the case of computer software documentation, for preparing the same or similar computer software, or (c) used by a party other than the Customer, except for: (i) emergency repair or overhaul work only, by or for the Customer, where the item or process concerned is not otherwise reasonably available to enable timely performance of the work, provided that the release or disclosure hereof outside the Customer shall be made subject to a prohibition against further use, release or disclosure; or (ii) release to a foreign government, as the interest of the United States may require, only for information or evaluation within such government or for emergency repair or overhaul work by or for such government under the conditions of (i) above. This legend, together with the indications of the portions of this data which are subject to such limitations shall be included on any reproduction hereof which includes any part of the portions subject to such limitations.

RESTRICTED RIGHTS IN SOFTWARE

The software described in this document is licensed software and subject to **restricted rights**. The software may be used with the computer for which or with which it was acquired. The software may be used with a backup computer if the computer for which or with which it was acquired is inoperative. The software may be copied for archive or backup purposes. The software may be modified or combined with other software, subject to the provision that those portions of the derivative software incorporating restricted rights software are subject to the same restricted rights.

Copyright © 1981 Tektronix, Inc. All rights reserved. Contents of this publication may not be reproduced in any form without the written permission of Tektronix, Inc.

Products of Tektronix, Inc. and its subsidiaries are covered by U.S. and foreign patents and/or pending patents.

TEKTRONIX, TEK, SCOPE-MOBILE, and  are registered trademarks of Tektronix, Inc. TELEQUIPMENT is a registered trademark of Tektronix U.K. Limited.

Printed in U.S.A. Specification and price change privileges are reserved.

CONTENTS

SECTION 1. LEARNING GUIDE	Page
Introduction	1-1
About This Manual Package	1-1
System Overview	1-3
Assembler Features	1-4
Linker Features	1-5
Library Generator Features	1-5
Syntax Notation	1-5
Introduction	1-5
Command Name	1-6
Parameters	1-6
For Continued Learning	1-7
SECTION 2. THE ASSEMBLER	
Introduction	2-1
Assembler Invocation	2-1
Assembler Input	2-2
Introduction	2-2
Statement Fields	2-3
Symbols	2-7
Values	2-9
Text Substitution	2-12
Expressions	2-13
Functions	2-21
Assembler Execution	2-36
Two Passes	2-36
Forward Referencing	2-36
Execution Sequence	2-36
Assembler Output	2-37
Object Module	2-37
Assembler Listing	2-37
Sample Source Program	2-44

	Page
SECTION 3. ASSEMBLER DIRECTIVES	
Introduction	3-1
The Assembler Directive Dictionary	3-3
SECTION 4. MACROS	
Introduction	4-1
Macro Expansion Process	4-1
Macro Definition	4-2
The MACRO Directive	4-2
The Macro Body	4-3
Macro Body Operators	4-3
The ENDM Directive	4-6
The EXITM Directive	4-6
Macro Invocation	4-7
Parameters	4-7
Macro Parameter Conventions	4-7
Macro Examples	4-10
SECTION 5. THE LINKER	
Introduction	5-1
Linker Invocation	5-1
Explanation	5-5
Command Options	5-5
Examples	5-15
Linker Execution	5-16
Section Attributes	5-16
Allocation of Sections	5-18
ENDREL	5-19
Linking a Library File	5-19
Typechecking	5-20
Linker Completion Condition	5-20
Linker Output	5-21
Listing File	5-21

	Page
SECTION 6. THE LIBRARY GENERATOR	
Introduction	6-1
LibGen Invocation	6-1
Command Option Parameters	6-3
Command Options	6-4
Examples	6-6
LibGen Execution	6-9
LibGen Output	6-10
The New Library File	6-10
The Listing	6-10
 SECTION 7. PROGRAMMING EXAMPLES	
Introduction	7-1
Use of Conditional Assembly in Macros	7-2
Save-and-Restore Macro	7-2
The SAVE Macro	7-3
The RESTORE Macro	7-4
Sample Invocations	7-4
SVC Generation	7-5
Creating Service Request Blocks	7-5
Generating Service Calls	7-9
Creating Constant Values	7-10
The CONSTANT Macro	7-11
The VARIABLE Macro	7-13
Macro Invocation	7-13
Creating and Using a Subroutine Library	7-14
The ADD Module	7-15
The SUBTRACT Module	7-16
Assembling the Modules	7-17
Creating the Library	7-20
Using the ADD Module from a Program	7-20
Using the SUBTRACT Module from a Program	7-25
Linking Overlays	7-30
Using the "@" Constant within a Macro	7-33
Delay Loop Macro	7-33
Macro Invocation	7-34

	Page
SECTION 7. PROGRAMMING EXAMPLES (Cont.)	
The Assembler INCLUDE Directive	7-34
Including Constant Definitions	7-34
Including COMMON Declarations	7-35
Including Macro Definitions	7-35
Authorship and Copyright Notices for Listings	7-36
 SECTION 8. HOST SPECIFICS	
 SECTION 9. ASSEMBLER SPECIFICS	
 SECTION 10. TECHNICAL NOTES	
Note 1. Differences Between the A Series and B Series Assemblers.....	10-1
 SECTION 11. TABLES	
Source Module Character Set	11-1
Assembler Directives	11-3
ASCII-Binary-Hexadecimal-Decimal Conversion	11-5
Decimal-Hexadecimal-Binary Equivalents.....	11-6
Hexadecimal Addition.....	11-7
Hexadecimal Multiplication.....	11-7
 SECTION 12. ERROR MESSAGES	
Introduction	12-1
Assembler Errors	12-1
Linker Errors	12-10
LibGen Errors	12-15
 SECTION 13. GLOSSARY	
 SECTION 14. INDEX	

Section 1 LEARNING GUIDE

	Page
Introduction	1-1
About This Manual Package	1-1
System Overview	1-3
Assembler Features	1-4
Linker Features	1-5
Library Generator Features	1-5
Syntax Notation	1-5
Introduction	1-5
Command Name	1-6
Parameters	1-6
Required Parameters	1-6
Optional Parameters	1-7
Choice of Parameters	1-7
Repeated Parameters	1-7
For Continued Learning	1-7

ILLUSTRATIONS

Fig. No.		Page
1-1	Sample B series assembler users manual package	1-2
1-2	Assembler programming process	1-3
1-3	Sample syntax block	1-6

Section 1

LEARNING GUIDE

INTRODUCTION

This Learning Guide gives an overview of features and functions of the TEKTRONIX 8500 Modular MDL B series assembler, linker, and library generator. The Learning Guide is divided into the following topics:

- **About This Manual Package.** Explains how to use this manual with your assembler.
- **System Overview.** Describes the functions of the assembler, linker, and library generator. Shows how these system programs interact with each other and with other programs in the operating system.
- **Features of the Assembler, Linker, and Library Generator.** Lists features of these programs that make them especially useful and powerful.
- **Syntax Notation.** Describes the syntax conventions used throughout this manual.
- **For Continued Learning.** Helps you decide where to go next in this manual to accomplish your own tasks.

ABOUT THIS MANUAL PACKAGE

The TEKTRONIX 8500 Modular MDL B Series Assembler Users Manual Package includes:

- the Core Users Manual
- the Host Specifics Users Manual
- the Assembler Specifics Users Manual
- the Reference Card

The **Core Users Manual** contains information that applies equally to all microprocessors and host systems supported; for example, the assembler directives, operand expressions, symbols, constants, and some advanced programming features. The linker and library generator are also discussed in the Core Users Manual.

The **Host Specifics Users Manual** contains the host-dependent information for your specific host system, installation procedures, and a sample Demonstration Run.

The **Assembler Specifics Users Manual** contains information about the instruction set, registers, addressing modes, and other processor-dependent information for your specific microprocessor.

The **Reference Card** contains brief reference information, such as the instruction set for your microprocessor, assembler directives, assembler invocation, linker command options, and library generator command options.

Each host system has a separate Host Specifics Users Manual. Each microprocessor has a separate Assembler Specifics Users Manual. The Host Specifics Users Manual and Assembler Specifics Users Manual are designed to be inserted into Section 8 and Section 9, respectively, of the Core Users Manual. Figure 1-1 shows the B Series Assembler Users Manual Package for the 8550 MDL and 8086/8088 Assembler.

Programming examples in the Host Specifics section are specific to each host system. All examples in other sections of the core users manual are completely host-independent and processor-independent. Some examples use 8086/8088 instructions, but similar instructions for any other microprocessor may be substituted without changing the validity of any example.

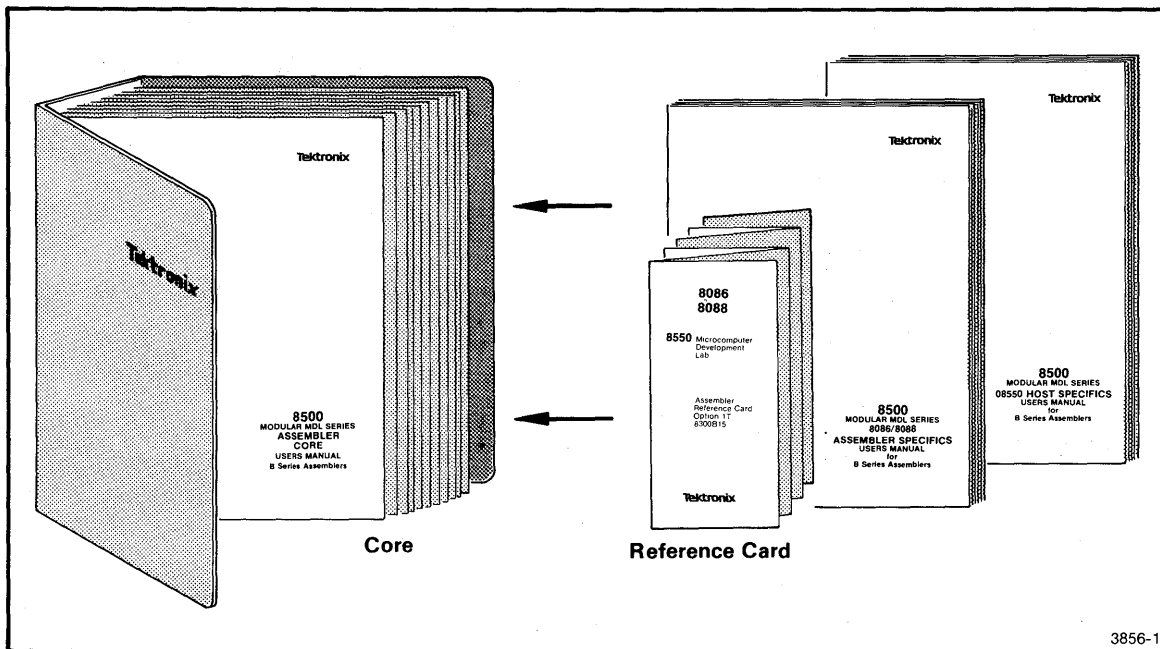


Fig. 1-1. Sample B series assembler users manual package.

This figure shows the B Series Assembler Users Manual Package for the 8550 MDL and 8086/8088 Assembler. The Host Specifics and Assembler Specifics Users Manuals are to be inserted into the Core Users Manual. The Reference Card is specific for the 8550 MDL and 8086/8088 Assembler.

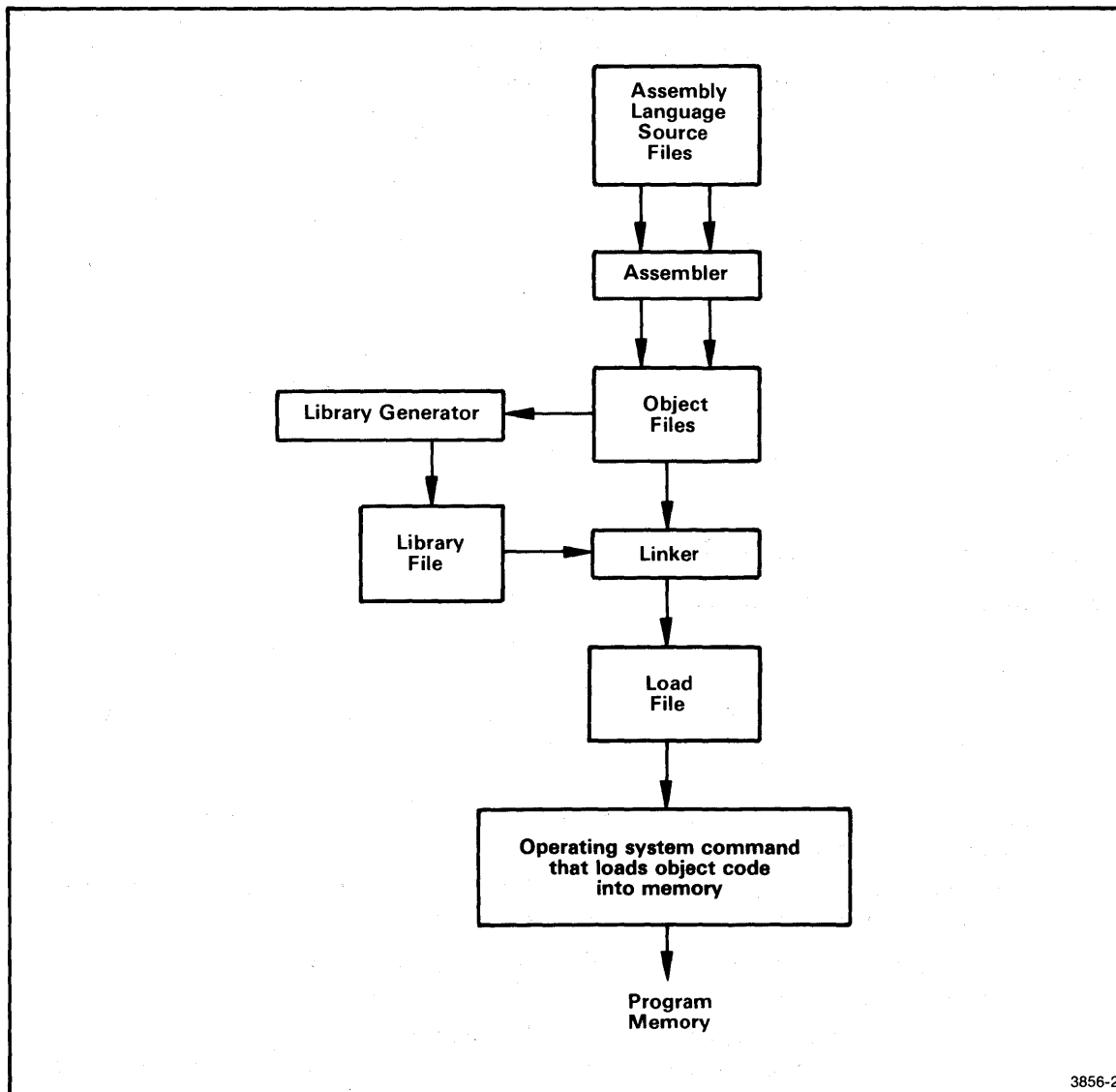


Fig. 1-2. Assembler programming process.

The assembler translates assembly language programs (source code) into relocatable machine language (object code). Frequently used object modules may be stored together in library files created by the library generator. The linker combines object modules from specified object files and library files into a load file of executable object code. The appropriate operating system command copies object code from load files into program memory.

SYSTEM OVERVIEW

Figure 1-2 shows how an executable program is produced from assembly language source files.

An assembly language source program may be written by a programmer or may be produced by a high-level language compiler.

The **assembler** translates assembly language statements (**source code**) into machine instructions (**object code**) and stores the resulting **object module** in a file called an **object file**. The TEKTRONIX 8500 Modular MDL B series assembler supports the translation of assembly languages for microprocessors having addresses up to 32 bits.

The **linker** collects object modules from specified files, determines where in memory each section of object code will reside, and produces a **load file** which contains the executable program. You may then copy the executable code into memory using the appropriate operating system command. (Under certain conditions you may load object modules without linking them. See The Assembler section of this manual.)

Frequently used subroutines can be developed and assembled separately. The resulting object code can then be stored with other object modules in a **library file**. When you include calls to library routines in your source program, the linker inserts the necessary object modules into the load file. The **library generator** creates and modifies library files.

ASSEMBLER FEATURES

Here are some important features of the TEKTRONIX 8500 Modular MDL B series assembler:

- Macros provide a convenient and powerful means for inserting and modifying frequently used segments of source code.
- Conditional assembly allows a sequence of source code to produce object code that varies according to specified conditions. This feature reinforces the assembler's macro capabilities.
- Linker-related assembler directives allow you to specify in your source code how the object code will be arranged in memory.
- Operand expressions may contain bit and string manipulations and special assembler functions as well as the standard arithmetic operations.
- Data constants may be entered as numbers in binary, octal, decimal, or hexadecimal notation, or as strings of ASCII characters enclosed in single quotes.
- Each error message contains a brief description of the error, an error number that helps you to locate more information in this manual, and a code that indicates the severity level of the error (warning, non-fatal error, or fatal error). You may also write your own error messages for use in conditional assembly.
- The assembler listing shows your source code, the object code, error messages, symbol table, and cross reference listing produced by the assembler. Listing directives allow you to select which segments of code or types of code are listed.

LINKER FEATURES

Here are some important features of the linker:

- You may link object modules from any number of object files or library files.
- You may define or change any of the following section attributes at link time:
 - the class that groups logically related sections;
 - the relocation type of a section of object code;
 - the exact or approximate location of a section in memory;
 - the values assigned to global symbols;
 - the address of the first instruction to be executed.
- You may enter linker command options from the system terminal or from a command file.
- Each error message contains a brief description of the error, an error number that helps you to locate more information in this manual, and a code that indicates the severity level of the error (warning, non-fatal error, serious error, or fatal error).
- The linker listing gives a detailed account of linker activity, showing the command options executed, global symbols, module names, section names, memory maps, and statistics.

LIBRARY GENERATOR FEATURES

Here are some important features of the library generator (LibGen):

- You may create libraries with 100 modules, assuming five global symbols per module.
- You may modify libraries by inserting, deleting, or replacing object modules.
- You may copy individual object modules into files.
- You may enter library generator command options from the system terminal or from a command file.
- Each error message contains a brief description of the error, an error number that helps you to locate more information in this manual, and a code that indicates the severity level of the error (warning, non-fatal error, or fatal error).
- The library generator listing shows the command options executed, global symbols, and a summary of library generator activities.

SYNTAX NOTATION

Introduction

This manual uses syntax blocks to present operating system commands, assembler directives, and assembler functions. The conventions used in the syntax blocks are described in this subsection. Figure 1-3 illustrates a sample syntax block.

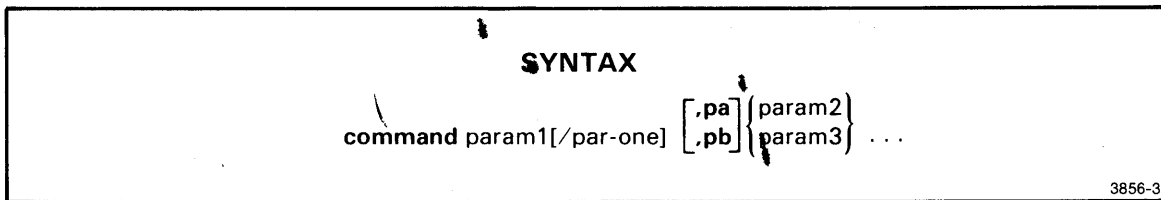


Fig. 1-3. Sample syntax block.

This figure illustrates a syntax block for a fictitious command line.

In Fig. 1-3, **command** represents a command name. **pa**, **pb**, param1, param2, param3, and par-one represent the command parameters.

Delimiters (usually spaces or commas) separate the parameters from the command name and from each other. For the linker and library generator, spaces are the only valid delimiters.

Command Name

A command name is a word that represents a command or assembler directive. In Fig. 1-3, the command name is **command**. Boldface characters in the command line must be entered exactly as shown.

Parameters

Parameters specify or modify how the command is executed. Parameters may be names, addresses, devices, numbers, characters, or symbols. Boldface parameters and any special characters, such as the comma, parentheses, "at" sign (@), slash (/), and equals sign (=), must be entered exactly as they appear in the syntax block.

Regular type (not boldface) parameters are descriptive terms that identify the type of information to be entered. Allowable entries appear in the PARAMETERS explanation for each command. In this manual, parameters are sometimes represented in a syntax block by two words, joined with a hyphen. The hyphen shows that they are not two separate parameters. In the example, "par-one" represents one parameter.

Required Parameters

Parameters may be required or optional. Required parameters appear in the command line without braces or brackets. For example, "param1" is a required parameter.

Optional Parameters

Optional parameters are enclosed in brackets [] in the syntax block. In Fig. 1-3, "/par-one" is an optional parameter. The special character slash (/) is required if "par-one" is used.

Choice of Parameters

Parameters are stacked one above another when there is a choice of two or more parameters. If the parameters are stacked within braces { }, one of the parameters must be selected. If the parameters are stacked within brackets [], the selection is optional. In the example, either "param2" or "param3" **must** be selected. Either **pa** or **pb** **may** be selected, or they may be omitted entirely. Notice that if either **pa** or **pb** is selected, it must be preceded by a comma.

Repeated Parameters

When three dots follow a parameter, the parameter may be repeated any number of times up to the end of the current line. In the example, the choice of "param2" or "param3" may be repeated as many times as the line permits.

FOR CONTINUED LEARNING

This Learning Guide has presented an overview of features and functions of the TEKTRONIX 8500 Modular MDL B series assembler, linker, and library generator. For a more detailed explanation of how to use these system programs, refer to the following sections.

Section 2, The Assembler. Describes the fundamental elements of assembly language statements. Gives rules for creating symbols, constants, and expressions. Describes special characters and assembler functions. Explains the assembler listing.

Section 3, Assembler Directives. Describes the function and use of each assembler directive. Includes one or more examples for each directive. Directives are arranged in alphabetical order.

Section 4, Macros. Shows you how to create and use assembler macros. Demonstrates the macro features of the TEKTRONIX 8500 Modular MDL B series assembler.

Section 5, The Linker. Describes the function and use of the linker. Explains each command option in the linker subsystem.

Section 6, The Library Generator. Describes the function and use of the library generator (LibGen). Explains each command option in the LibGen subsystem.

Section 7, Programming Examples. Demonstrates and explains useful applications of the assembler, linker, and library generator.

Section 8, Host Specifics. Provides a place to insert your Host Specifics supplements. Each supplement gives reference information that is specific to a particular host system, and also contains a Demonstration Run and installation instructions.

Section 9, Assembler Specifics. Provides a place to insert your Assembler Specifics supplements. Each supplement gives information that varies with each microprocessor: registers, instruction sets, special error messages, and any exceptions to the standard reference material in this manual.

Section 10, Technical Notes. Provides miscellaneous technical information. Technical Note 1 discusses the differences between the A series and B series assemblers.

Section 11, Tables. Summarizes reference information in tabular form.

Section 12, Error Messages. Lists the error messages for the assembler, linker, and library generator. Each error message is accompanied by a description of the problem.

Section 13, Glossary. Defines special terms used in this manual.

Section 14, Index. Gives you a place to start when you don't know where else to look.

Section 2 THE ASSEMBLER

	Page		Page
Introduction	2-1	Assembler Execution	2-36
Assembler Invocation	2-1	Two Passes	2-36
Assembler Input	2-2	Forward Referencing	2-36
Introduction	2-2	Execution Sequence	2-36
Statement Fields	2-3	Assembler Output	2-37
Label Field	2-4	Object Module	2-37
Operation Field	2-5	Assembler Listing	2-37
Operand Field	2-6	Source Listing	2-38
Comment Field	2-7	Cross-Reference Listing	2-39
Symbols	2-7	Symbol Table	2-39
User-defined Symbols	2-8	Sample Source Program	2-44
Constructing Symbols	2-8	Sample Source Listing	2-45
Defining Symbols	2-8	Sample Cross-Reference Listing	2-51
Predefined Symbols	2-9	Sample Symbol Table	2-52
Values	2-9		
Numeric Values	2-9		
Scalar Values	2-10		
Address Values	2-10		
Floating Point Values	2-10		
Numeric Constants	2-10		
Numeric Variables	2-11		
String Values	2-11		
String Constants	2-11		
String Variables	2-11		
Conversions	2-12		
Text Substitution	2-12		
Expressions	2-13		
Introduction	2-13		
Hierarchy	2-14		
Operators	2-15		
Arithmetic Operators	2-16		
Logical Operators	2-17		
Relational Operators	2-19		
String Operator	2-21		
Functions	2-21		
BASE—Determines whether two values have a common base	2-22		
BITS—Returns specified bit string	2-24		
DEF—Determines if a symbol has been defined ..	2-26		
ENDOF—Returns the address of the last byte of a section	2-27		
HI—Returns high byte of low-order word	2-28		
LO—Returns low byte of numeric value	2-29		
NCHR—Returns number of characters in string ..	2-30		
SCALAR—Converts address to scalar	2-31		
SEG—Returns substring	2-32		
STRING—Converts scalar to string	2-34		
STRINGOF—Returns macro argument	2-35		

TABLES

Table No.		
2-1	Expression Operators and Functions	2-14
2-2	Hierarchy of Operators	2-15
2-3	Types of Comparisons with Relational Operators	2-20
2-4	Result-Values	2-38

ILLUSTRATIONS

Fig. No.		
2-1	Formatted source file	2-4
2-2	Sample assembler source listing	2-40
2-3	Sample assembler cross-reference listing ..	2-42
2-4	Sample assembler symbol table listing	2-43
2-5	Sample 8086/8088 source program	2-44

Section 2

THE ASSEMBLER

INTRODUCTION

The assembler translates assembly language statements (source code) into machine instructions (object code). The resulting object module, stored in a file, is suitable for input to the linker or to the library generator (LibGen).

This section describes the Tektronix Assembler, and is divided into the following subsections:

- **Assembler Invocation.** Describes how to invoke the assembler.
- **Assembler Input.** Describes the source module.
- **Assembler Execution.** Describes the operations performed by the assembler.
- **Assembler Output.** Describes the output of the assembler: the object module and the assembler listing. Includes an annotated assembler listing of a sample program.

ASSEMBLER INVOCATION

With the 8550 Microcomputer Development Lab or the 8560 Multi-User Software Development Unit, the assembler is invoked by the operating system command **asm**.

NOTE

If you are using any system other than the 8550 or 8560, the assembler invocation command may be different. Refer to the Host Specifics section of this manual for further information.

Throughout this section, the same notation conventions are used as described in the Learning Guide of this manual. Additionally, if you are using the 8550 Microcomputer Development Lab, the command name **asm** may be entered in either upper or lower case.

SYNTAX

```
asm [object] [list] source...
```

PARAMETERS

- object** The name or filespec of the file to receive the object code output by the assembler.
- list** The name or filespec of the file to receive the listing output by the assembler.
- source** The name or filespec of the file that contains the source code to be processed by the assembler.

EXPLANATION

The **asm** command invokes the Tektronix Assembler. The source code, residing in one or more files, is translated into object code (machine language), which is written to the specified file or device. An assembler listing is also generated and is written to the specified file or device.

The object file and listing file are optional; you may replace either or both with a null parameter. If you are using an **8550** system, you must enter two commas if you want to leave out one parameter and three commas if you want to leave out two parameters. For example:

```
ASM ,,MY.ASML MY.ASM        to leave out the object filespec.
ASM MY.OBJ ,,MY.ASM        to leave out the listing filespec.
ASM ,,,MY.ASM              to leave out both object and listing filespecs.
```

If you are using an **8560** system, you must replace the parameter with a null string. For example:

```
asm '' mylist mysource      to leave out the object filespec.
asm myobject '' mysource    to leave out the listing filespec.
asm '' '' mysource         to leave out both object and listing filespecs.
```

You must always include the source filespec.

ASSEMBLER INPUT

Introduction

The input to the assembler is made up of one or more source modules. The source module consists of assembly language statements. There are three types of assembly language statements:

- assembly language instructions,
- assembler directives, and
- macro invocations.

Blank lines and comment lines (lines beginning with a semicolon) may be included in the input, but have no effect on the assembler. Any other assembler input will cause an error.

If the assembler input resides in one or more source files, each filespec must be specified in the assembler invocation line. The assembler makes two passes through the source code. If the input is not read from a random access device, the statements must be entered twice. When the assembler is ready to read the source code a second time, it displays the following message on the system terminal:

```
*****Pass 2
```

If the statements entered on the second pass are not identical to those entered on the first pass, assembly errors will result. Additionally, if the input is not read from a random access device, the source code may not contain macros or REPEAT blocks. The source code within macros and REPEAT blocks is not stored in core but is accessed each time it is needed.

The rest of this subsection discusses the Tektronix Assembler language elements and is divided into the following topics:

- **Statement Fields**—Explains the four fields in an assembler source statement: label, operation, operand, and comment.
- **Symbols**—Explains how symbols are used in assembler source programs.
- **Values**—Describes numeric and string values used by the assembler.
- **Text Substitution**—Describes the use of text substitution.
- **Expressions**—Describes the type of permitted expressions and their required formats. Describes the use of operators in expressions.
- **Functions**—Defines and gives the results of assembler functions. The functions are listed alphabetically for reference.

Statement Fields

An assembly language source program consists of statements. Each statement occupies one line of text. Each statement may contain up to 127 characters; the line ends with a RETURN character (end-of-line character; See the Host Specifics section of this manual for the actual ASCII code). Blank lines can be used within the program for readability and have no effect on the assembly.

A statement consists of four fields. Each field may vary in width, and certain fields may be omitted, but the fields always occur in the following order:

```
Label  Operation  Operand  Comment
```

Programs are easier to read when each field has a constant width on each line. This columnar format can be implemented with tab settings. Figure 2-1 is an example of a formatted 8086/8088 source file.

Label	Operation	Operand	Comment
	GLOBAL	PORTN,OUTSUB	
PORTN	EQU	15	; PORT = 15
START	MOV	AL,#'?'	; CHARACTER = '?'
	CALLS	OUTSUB,OUTSUB	; SEND '?' TO PORT 15...
	HLT		; ... AND STOP.
	END	START	

3856-4

Fig. 2-1. Formatted source file.

Each field has a constant width in this 8086/8088 source program, making it easier to read.

Label Field

The label field, when used, must begin in the first character position of a line. A space, tab, semicolon, or RETURN terminates the label field. A statement's label allows the statement to be referenced by other statements.

The label is a user-defined symbol. The symbol must follow the rules for constructing symbols (described later in this section). Embedded spaces are not permitted within a symbol. Every label must be unique within each assembler source program. The assembler generates an error message when duplicate labels are used.

A label is permitted in all assembly language instructions, macro invocations, blank lines, and the following assembler directives:

ADDRESS	IF
ASCII	INCLUDE
BLOCK	LONG
BYTE	ORG
COMMON	RESERVE
END	RESUME
EQU	SECTION
FLOAT	SET
GLOBAL	WORD

A label is **not** permitted with the following assembler directives:

ELSE	NAME
ELSEIF	NOLIST
ENDIF	PAGE
ENDM	REPEAT
ENDR	SPACE
EXITM	STITLE
EXITR	STRING
LIST	TITLE
MACRO	WARNING

The meaning of the label in an **assembler directive** statement depends upon the particular directive. For many directives the label is optional and not always meaningful. However, labels are always required with the EQU and SET directives. See the Assembler Directives section of this manual for the specific meaning in each directive.

```
Label  Operation  Operand  Comment
PORTN  EQU        15          ; PORT = 15
```

In this example, the label PORTN is given the value 15.

A label used in an **assembly language instruction** or **macro invocation** represents the memory address of the first byte of the instruction.

```
Label  Operation  Operand  Comment
START  MOV        AL,#'?'    ; CHARACTER = '?'
```

In this line, the label START represents the address of the first byte of the MOV instruction.

An address is relative to the base address (beginning address) of the section in which it appears. At link time, relocatable sections are assigned a new base address. Therefore, any symbol representing an address is relocated relative to its base address at link time. (See the Address Values discussion later in this section for more information on relative addresses.)

Operation Field

The operation field begins immediately after the label field. If the label is omitted, the operation field may begin anywhere after the first character position in the line. The operation field is terminated by a space, a tab, a RETURN, or a semicolon (indicating the beginning of a comment field).

The symbol in the operation field indicates the type of action to be taken by the assembler. The symbol may be an assembly language instruction mnemonic, an assembler directive, or a macro invocation.

If the word in the operation field is an **assembly language instruction**, the assembler translates the statement into a machine instruction.

```
Label  Operation  Operand  Comment
START  MOV        AL,#'?'    ; CHARACTER = '?'
```

MOV (an 8086/8088 mnemonic) is translated into a machine instruction by the assembler.

An **assembler directive** in the operation field specifies certain actions to be performed during assembly. Assembler directives may or may not generate object code.

```
Label  Operation  Operand  Comment
GLOBAL PORTN,OUTSUB
```

In this example, the assembler directive GLOBAL in the operation field declares PORTN and OUTSUB as global symbols.

NOTE

The name of an assembly language instruction for a particular microprocessor may be identical to a standard assembler reserved word. In that case, the name of that assembler reserved word is changed. The Assembler Specifics section for your microprocessor contains information about any changed assembler reserved words.

A **macro name** in the operation field specifies the macro definition block to be expanded.

Label	Operation	Operand	Comment
	MACRO	QQQ	; MACRO QQQ DEFINED
	.		
	ENDM		
	.		
	QQQ		; INVOCATION OF MACRO QQQ

In this example, the macro QQQ is invoked when QQQ appears in the operation field.

If the operation field does not contain an assembly language instruction, an assembler directive, or a macro name, the assembler rejects the entire statement and prints an error message. See the Assembler Specifics section of this manual for a list of your processor's instruction mnemonics. Assembler directives are presented alphabetically in the Assembler Directives section of this manual. Macros are described in the Macros section of this manual.

Operand Field

The operand field specifies values required by the assembly language instruction, the assembler directive, or the macro invocation in the operation field. The word in the operation field determines the required type, number, and order of operands. For example:

Label	Operation	Operand	Comment
START	MOV	AL,#'?'	; CHARACTER = '?'

The 8086/8088 MOV instruction requires two operands: a register, followed by a value. In this example, register AL (a predefined symbol) and a string value are used.

The value in the operand field may be represented by an expression. (See the Expressions discussion later in this section.) An expression may be one of the following:

- a numeric or string constant,
- a symbol, or
- a combination of constants and symbols with operators and functions.

Symbols appearing in the operand field may be predefined or user-defined. (See the Symbols discussion later in this section.) If a symbol appearing in the operand field is not predefined, it must be defined in one of the following ways:

- the symbol must appear in the **label** field of an assembler statement (assembler directive or assembly language instruction); or
- the symbol must appear in the **operand** field of a GLOBAL, STRING, SECTION, COMMON, RESERVE, or MACRO directive.

The operand field may contain spaces to improve program readability. The spaces must not be within symbols.

Label	Operation	Operand
	BYTE	5,35,45,55
	BYTE	5, 35, 45, 55

Both of the above statement lines produce identical results.

Comment Field

The comment field is optional, but may be included in any statement line. The comment field begins with a semicolon (;) and ends with a RETURN. All characters following the semicolon are considered a part of the comment. Comments are used for program documentation and have no effect on the object code produced by the assembler. If no other fields are used, the comment field may begin anywhere in the statement line.

Label	Operation	Operand	Comment
			; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER
OUTSUB	OUT	#PORTN,AL	; OUTSUB STARTS HERE

In this example, the first statement has no effect on the object code produced; the semicolon (;) in the first column causes the entire line to be treated as a comment. In the next line, the semicolon causes 'OUTSUB STARTS HERE' to be treated as a comment.

Text substitution is the only type of action performed by the assembler within the comment field. Text substitution is discussed later in this section. The double quote (") signals substitution. Therefore, to include a double quote (") character within a comment, you must precede the " character with an up-arrow (^) character.

NOTE

The up-arrow (^) character, in a macro parameter, comment, or literal string, cancels the special significance of the immediately following character.

Symbols

A symbol is a user-defined or predefined word or mnemonic that represents a value, register name, macro name, class name, module name, or instruction. Symbols make a program easier to read, and reduce the risk of error when the program is modified.

User-defined Symbols

A user-defined symbol is a word or mnemonic that you create to represent a numeric value (scalar or address), a string value, or a macro name. By using symbols you can refer to a data value or a memory address without using the specific value.

For example, if you need to refer to a data value frequently within a program, that value can be assigned to a symbol. Then, if you need to change that value, you only need to modify the defining statement, rather than modifying each statement that references the value.

```
PORTN EQU 15
```

In this statement, the symbol PORTN is defined by the EQU directive to have the value of 15. PORTN can be used throughout the program.

Constructing Symbols

A symbol consists of one or more characters beginning with a letter and containing only letters, digits, periods, underscores, or dollar signs. Only the first 16 characters are considered significant; any additional characters are discarded. In addition, a user-defined symbol must not be a reserved word (see the reserved word list in the Assembler Specifics section of this manual).

NOTE

There is no case distinction. Internally, lowercase is converted to uppercase.

The following symbols are **valid**:

```
PORTN
HERE
LOOP.5
LOOP_6
A123456$
TO_DO
UPPERandlower (same as UPPERANDLOWER)
ANEXTREMELYLONGSYMBOL (same as ANEXTREMELYLONGS)
```

The following symbols are **invalid**:

```
1SYMBOL (must begin with a letter)
.LOOP (must begin with a letter)
STRING (must not be a reserved word)
ONE SYMBOL (must not contain blanks)
```

Defining Symbols

User-defined symbols are defined when they appear in: (1) the label field of an assembly language instruction, macro invocation, or assembler directive, or (2) the operand field of a GLOBAL, SECTION, COMMON, RESERVE, MACRO, or STRING directive. User-defined symbols are assigned values during the assembler's first pass. When the symbols are encountered in the second pass, they are replaced by the assigned values.

A symbol in the label field of an assembly language instruction represents the address of the first byte of that instruction. A label symbol allows you to transfer control to an instruction without knowing its absolute address. For example, a destination address for a jump instruction (JMP in 8086/8088) can be represented with a symbol.

```
AGAIN INC      CX
      .
      .
      .
      JMP     AGAIN
```

AGAIN is a user-defined symbol representing the address of the INC (increment) instruction.

When a symbol is used in the label field of an assembler directive, its meaning depends upon the directive. Generally, the symbol represents a data constant or the memory address of data. See the Assembler Directives section of this manual for the specific meaning of a label with each directive.

Generally, a symbol may not be redefined within a program. However, the SET directive may be used to redefine a symbol previously defined by the SET directive. This allows you to temporarily assign a value to an assembler variable during assembly.

Predefined Symbols

Predefined symbols include:

- assembler directives and options,
- assembler functions,
- assembly language instruction mnemonics,
- processor register names and symbols, and
- ENDREL—a global symbol that is assigned at link time to the lowest address that is above all relocatable sections.

The assembler directives and options are listed in the Assembler Directives section of this manual. Assembler functions are discussed later in this section. See the Assembler Specifics section of this manual for a list of instruction mnemonics and reserved words for your processor. See the Linker section of this manual for more information about ENDREL.

Values

The assembler recognizes two kinds of values: numeric and string.

Numeric Values

The assembler recognizes three types of numeric values: scalar, address, and floating point. Scalar and address values are 32-bit (2-word) quantities. Floating point values may be 32-bit (single precision) or 64-bit (double precision) quantities. Scalars are signed values. Addresses are unsigned values.

Scalar Values. Scalar values are signed integers ranging from -2,147,483,648 to 2,147,483,647. (The two's complement of a positive number represents the corresponding negative integer.) Scalar values can be used as numeric data within an assembly language program.

Address Values. An address value specifies a memory location. An address value is an unsigned 32-bit quantity that ranges from 0 to 4,294,967,295.

An address is defined relative to the beginning of the section in which it appears. The assembler generates an object module (made up of one or more sections) with address values relative to the beginning of each section. At assembly time, the beginning (base address) of each relocatable section is zero. At link time, the linker relocates the individual sections, thus redefining the base address of each section. (See the Linker section of this manual for a discussion on section relocation.) The actual address of a byte is not known until after the linking process is complete.

During assembly, a location counter (which simulates the processor program counter) holds the address of the object code being generated. The dollar sign (\$), when used in the operand field, represents the current value of the location counter (the address of the first byte of the machine instruction or data item currently being generated). For example:

```
Label  Operation  Operand
      IF          $ > OFFH
```

In this statement, the current value of the location counter is compared with the value OFFH.

Floating Point Values. A floating point value is a constant generated by the FLOAT directive. It may be 32 bits (single precision) or 64 bits (double precision).

No arithmetic may be performed on floating point constants during assembly. For more information on floating point format, see the discussion of the FLOAT directive in the Assembler Directives section of this manual.

Numeric Constants. Numeric constants may be entered in decimal, binary, octal, or hexadecimal notation. The assembler assumes that a number is in decimal unless a suffix letter identifies it as binary, octal, or hexadecimal. The following suffix letters are used:

- **B** denotes a **binary** number. For example, 1010B and 11111111B are binary numbers.
- **O** (capital letter O, not zero) or **Q** denotes an **octal** number. For example, 377Q and 177777O are octal numbers.
- **H** denotes a **hexadecimal** number. For example, 1A2CH and OFFFFH are hexadecimal numbers.

NOTE

Numeric constants must begin with a numeric character. Any hexadecimal number that has an alphabetic character in the first digit must be preceded with a zero.

A numeric constant may be assigned to a symbol with the EQU directive.

```
PORTN EQU 15
```

In this example, PORTN is assigned the value 15.

Numeric Variables. During assembly, a symbol may be temporarily assigned a value with the SET directive. A symbol defined with the SET directive is called an assembler variable. The value associated with the variable may be changed by subsequent SET directives. When the variable is encountered, the current value is used. A symbol used as an assembler variable must not have been previously defined (except with another SET directive).

```
COUNT SET      1
```

In this example, the symbol COUNT is an assembler variable, and is assigned the numeric value 1 with the SET directive. When the symbol COUNT is encountered by the assembler, the current value is used. If a subsequent SET directive assigns another value to COUNT, the reassigned value is used.

String Values

Character strings may be used with the assembler. Individual characters are translated into their ASCII representation (8 bits, with the leftmost bit set to zero).

String Constants. String values entered as constants in an assembler program are enclosed in single quotes ('):

```
'STRINGS'
```

Any ASCII character, with the exception of the RETURN character, may be included in a string constant. To include special characters, such as a double quote ("), a single quote ('), or an up-arrow (^) precede the special character with an up-arrow (^).

NOTE

The up-arrow character (^) cancels the special significance of the immediately following character.

The null string (") contains zero characters.

String Variables. A character string may be assigned to a string variable with the SET directive. The symbol to be used as the string variable must first be declared with the STRING directive. The STRING directive specifies the maximum length of the string variable. The maximum length (which defaults to 16) is enclosed in parentheses. For example:

```
      STRING      STVAR(10)
STVAR SET      'CHARACTERS'
```

In this example, the symbol STVAR is a string variable. Up to 10 characters may be assigned to the variable STVAR.

The length of the string variable is the length of the character string currently assigned to the variable. If you try to assign a character string that is longer than the declared length of the variable, the character string is truncated and an error message is generated.

Strings are only used during assembly. There is no run-time storage space reserved for string variables.

Conversions

A string constant may be assigned to a symbol with the EQU directive.

```
Label Operation Operand
SYM1 EQU 'ABCD'
```

The string is converted to a two-word (four-byte) numeric value. The numeric value is the ASCII representation of the string. If the string is longer than four characters, only the first four characters are converted and an error message is generated. If the string is less than four characters, the numeric value is padded with zeros to the left. The value of the null string (") is zero. Here are some example conversions:

Character String	Numeric Value
'A'	00000000H
'A'	00000041H
'?'	0000003FH
'ABCD'	41424344H
'ABCDE'	41424344H (truncation error occurs)
'12'	00003132H

For an ASCII-to-hexadecimal conversion table, see the Tables section of this manual.

If a numeric value is assigned to a string variable, the numeric value is converted to its string representation. The numeric value is treated as a literal string constant. For example:

```
Label Operation Operand Comment
A STRING A,B(3)
A SET 6 ;SETS A TO '6'
A SET -3 ;SETS A TO '-3'
B SET 1234 ;SETS B TO '123' AND A TRUNCATION ERROR OCCURS
```

Text Substitution

String values can be substituted within a statement line during assembly by the use of string variables. The double quote (") is the substitution delimiter. When the assembler encounters a string variable enclosed within double quotes ("variable"), the variable is replaced by the current string value assigned to that string variable. The result of the string substitution may not contain double quotes that are not enclosed in single quotes (text substitution may not be nested).

Table 2-1
Expression Operators and Functions

Type	Operator/Function	Meaning
Unary Arithmetic	+	Identity
	-	Sign inversion
Binary Arithmetic	*	Multiplication
	/	Division
	+	Addition
	-	Subtraction
	MOD	Modulus
	SHL SHR	Left shift Right shift
Unary Logical	\	NOT (bit inversion)
Binary Logical	&	AND
	!	Logical OR
	!!	Exclusive OR
Relational	=	Equal
	<>	Not equal
	>	Greater than
	>=	Greater than or equal
	<	Less than
	<=	Less than or equal
String	:	Concatenation
Logical Functions	BASE	Base address comparison
	DEF	Symbol definition
Numeric Functions	BITS	Bit string
	ENDOF	End of section
	HI	High byte
	LO	Low byte
	SCALAR	Conversion to scalar
String Functions	NCHR	Number of characters
	SEG	Substring
	STRING	Conversion to string
	STRINGOF	Return macro argument

Hierarchy

In an expression involving more than one operator, the operators are performed according to a predefined order of precedence. Table 2-2 shows the operator hierarchy.

**Table 2-2
Hierarchy of Operators**

Precedence	Operators					Type of Operator
1.	BASE HI SEG	BITS LO STRING	DEF NCHR STRINGOF	ENDOF SCALAR		Functions
2.	:	.				Concatenation
3.	+	-	\			Unary; logical NOT
4.	*	/	MOD	SHL	SHR	Multiplication, division, modulus, shifts
5.	+	-				Addition, subtraction
6.	=	<>	>=<	<=<		Relational
7.	&					Logical AND
8.	!	!!				Logical OR, exclusive OR

Table 2-2 shows that the functions have the highest precedence and that the OR operator has the lowest. Operators with the highest precedence are performed first. Operators with the lowest precedence are performed last. Operators in the same group have equal precedence, and are performed from left to right within the expression.

Parentheses may be used to override the order of precedence. Up to three levels of nested parentheses are allowed in an expression (including function argument delimiters). The most deeply nested subexpressions are evaluated first. It is possible to create an expression that is too complex for the assembler to evaluate (parentheses nested more than 3 levels deep). If the expression entered is too complex, an expression error message is displayed.

Operators

An operator within an expression acts upon one or more terms. The operators and types of terms permitted for each operator are discussed in the following paragraphs.

If an operator requires a numeric operand, and a string operand is provided, the string operand is converted to a numeric value. (See the Conversions discussion earlier in this section.)

Arithmetic Operators. Arithmetic operators act on numeric values.

- + Unary plus** Identity operator: does not change the value of the term. May be applied to scalar or address values.
- Unary negative** Indicates sign inversion. (Two's complement.) May be applied to scalar values only.
- * Multiplication** Multiplies two scalar values.
- / Division** Divides two scalar values.
- + Addition** Adds two terms (scalar or address), as follows:
 Scalar + Scalar = Scalar
 Scalar + Address = Address
 Address + Scalar = Address
 Address + Address = error
- Subtraction** Subtracts two terms (scalar or address), as follows:
 Scalar - Scalar = Scalar
 Address - Scalar = Address
 Address - Address = Scalar (addresses must have same base)
 Scalar - Address = error
- MOD Modulus** The remainder that results when the first term is divided by the second. Both terms must be scalar values. The sign of the returned value is determined by the sign of the second term. For example:

X	Y	X MOD Y
2	2	0
5	2	1
5	-2	-1
-5	2	1
-5	-2	-1

- SHL Shift left** The first term is shifted to the left the number of bit positions specified by the second term. Both terms must be scalar values. The second term (the number of bits to be shifted) must be a non-negative scalar value. For example:

1 SHL 1 results in 2

0 0 0 0 0 0 0 1
 0000 0000 0000 0000 0000 0000 0000 0001

0 0 0 0 0 0 0 2
 0000 0000 0000 0000 0000 0000 0000 0010

When the bits are shifted, the leftmost bits are discarded; the vacated bit positions on the right become zeros. For example:

OFOFOFOFOH SHL 1 results in OE1E1E1EOH

F	0	F	0	F	0	F	0
1111	0000	1111	0000	1111	0000	1111	0000
E	1	E	1	E	1	E	0
1110	0001	1110	0001	1110	0001	1110	0000

If the second term is greater than 32, the result is zero, and an error message is generated.

SHR Shift right

The first term is shifted to the right the number of bit positions specified by the second term. Both terms must be scalar values. The second term (the number of bits to be shifted) must be a non-negative scalar value. For example:

2 SHR 1 results in 1

0	0	0	0	0	0	0	2
0000	0000	0000	0000	0000	0000	0000	0010
0	0	0	0	0	0	0	1
0000	0000	0000	0000	0000	0000	0000	0001

When the bits are shifted, the rightmost bits are discarded; the vacated bit positions on the left become zeros. If the second term is greater than 32, the result is zero, and an error message is generated.

Logical Operators. The logical operators, NOT (\), AND (&), OR (!), and exclusive-OR (!!), correspond to their Boolean algebra equivalents, as shown in the following truth table.

X	Y	\X	X&Y	X!Y	X!!Y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Logical operators may only be applied to scalar values.

\ NOT

Returns the one's complement of the following term by complementing each bit in the term. (Returns a 1 if the bit is 0, and returns a 0 if the bit is 1.)

\ OFOFOF0FH results in OFOF0F0FH.

O	F	O	F	O	F	O	F
0000	1111	0000	1111	0000	1111	0000	1111
F	O	F	O	F	O	F	O
1111	0000	1111	0000	1111	0000	1111	0000

& AND

Returns the logical AND of two terms. Compares the terms bit-by-bit; returns a 1 if both bits are 1, otherwise returns a 0.

Example:

DVAL EQU OFOF0F0FH & 0AAAAAA0H

F	O	F	O	F	O	F	O
1111	0000	1111	0000	1111	0000	1111	0000
A	A	A	A	A	A	A	O
1010	1010	1010	1010	1010	1010	1010	0000

DVAL is assigned the value 0A0A0A0A0H.

A	O	A	O	A	O	A	O
1010	0000	1010	0000	1010	0000	1010	0000

! OR

Returns the logical OR of two terms. Compares the terms bit-by-bit; returns a 1 if either bit is 1, returns a 0 if both bits are 0.

Example:

RVAL EQU OFOF0F0FH ! 0AAAAAA0H

F	O	F	O	F	O	F	O
1111	0000	1111	0000	1111	0000	1111	0000
A	A	A	A	A	A	A	O
1010	1010	1010	1010	1010	1010	1010	0000

RVAL is assigned the value OFAF0F0FH.

F	A	F	A	F	A	F	O
1111	1010	1111	1010	1111	1010	1111	0000

!! Exclusive-OR

Returns the logical exclusive-OR of two terms. Compares the terms bit-by-bit; returns a 1 when the bits are different, returns a 0 when the bits are the same.

Example:

```
ERVAL EQU OFOF0F0FH !! OAAAAAAAAH
  F   0   F   0   F   0   F   0
1111 0000 1111 0000 1111 0000 1111 0000

  A   A   A   A   A   A   A   0
1010 1010 1010 1010 1010 1010 1010 0000
```

ERVAL is assigned the value 5A5A5A50H.

```
  5   A   5   A   5   A   5   0
0101 1010 0101 1010 0101 1010 0101 0000
```

Relational Operators. Relational operators compare two terms and return the value -1 (FFFFFFFF) for a true expression and 0 for a false expression.

=	Equal
<>	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

Relational operators allow comparison of scalar values, address values, and string values.

Scalar values are compared as signed numeric values. For example:

```
Label Operation Operand
COUNT SET 1
        IF COUNT < 5
        IF COUNT > -1
        .
        .
F EQU 7 = COUNT
```

The relational operators in this example compare signed numeric (scalar) values.

Address values are compared as unsigned numeric values. Address values are compared as offsets from their base address. Address values from different sections may only be compared for equality (=) or inequality (<>): They are always unequal.

```

START  MOV      AL,#'?'
      .
      .
      .
NEXT   MOV      AH,#0
T      EQU      START < NEXT
    
```

The less-than (<) operator in this example compares two unsigned numeric (address) values within the same section.

If only one term is an address, a relational operator performs an unsigned numeric comparison between the scalar and the address offset.

String values are compared numerically according to the ASCII collating sequence. (See the Tables section of this manual.) Comparison proceeds left to right, character-by-character. Two strings are considered equal if they have the same length and contain identical character sequences. If they are identical in sequence but one string is longer than the other, the longer string is considered greater. The following examples show the results of various string comparisons:

```

'AB' = 'AB'           results in      -1 (true)
'A' > 'B'             results in      0 (false) A less than B
'ABC' > 'ABC '        results in      0 (false) the right term is longer
'ACB' > 'ABC'         results in      -1 (true) C greater than B
    
```

If only one term is a string, the first four characters of the string are converted to a scalar value and a numeric comparison is performed.

The types of comparisons are summarized in Table 2-3.

Table 2-3
Types of Comparisons with Relational Operators

Left Operand	Right Operand		
	String	Scalar	Address
String	String Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
Scalar	Signed Numeric Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
Address	Unsigned Numeric Comparison	Unsigned Numeric Comparison	Unsigned Numeric Comparison

String Operator.

: Concatenation Combines two strings into a single string. For example:

Label	Operation	Operand
	STRING	STR1(5),STR2(6),STR3(11)
STR1	SET	'HELLO'
STR2	SET	' THERE'
STR3	SET	STR1 : STR2

STR3 now is 'HELLO THERE'.

If the resulting string is assigned to a variable, the length of the resulting string must not exceed the length specified for that variable by the STRING directive.

Numeric values may not be concatenated.

Functions

The following predefined functions return a value when used in an expression:

- Logical Functions

BASE—Determines whether two values have a common base.

DEF—Determines if a symbol has been defined.

- Numeric Functions

BITS—Extracts a bit-string.

ENDOF—Returns the address of the last byte of a section.

HI—Returns the high byte of the low-order word of a numeric expression.

LO—Returns the low byte of a numeric expression.

SCALAR—Converts an address value to a scalar value.

- String Functions

NCHR—Returns the current length of a string variable.

SEG—Extracts a substring of a string.

STRING—Converts a scalar value to a string.

STRINGOF—Creates string expression from macro parameter.

Each of these functions is described in detail in the following pages. The conventions used in these descriptions are the same as those described in the Learning Guide.

SYNTAX

BASE(numvalue1,numvalue2)

PARAMETERS

numvalue Any expression that evaluates to a numeric value. Usually a label symbol.

EXPLANATION

The BASE function compares two numeric values to see if they have the same base. The BASE function returns a -1 (true) if the values have the same base. The BASE function returns a 0 (false) if the values do not have the same base.

All addresses within a section share the same base. All scalar values share the same base. Scalar values and address values do not have the same base. Each SECTION, COMMON, and RESERVE directive defines a new address base. The default section (any statements not preceded by a SECTION or COMMON directive) has a separate base. All unbound globals are assumed to have unique bases.

The BASE function is typically used to compare label symbols in a conditional assembly statement.

EXAMPLES

Label	Operation	Operand	
Q	EQU	5	} Both scalars
.	.	.	
R	EQU	15	} Statements are assembled because Q and R share common base
	IF	BASE(Q,R)	
.	.	.	
	ENDIF	.	

In this example, the two scalar values Q and R are compared. Since both Q and R represent scalar values, they share a common base. The function BASE(Q,R) returns a -1 (true) and the statement lines between IF and ENDIF are assembled.

Label	Operation	Operand	
	SECTION	SEC1	
HERE	BLOCK	100H	
THERE	BLOCK	100H	
	IF	BASE (HERE, THERE)	} Statements are assembled because HERE and THERE are in the same section
	.		
	.		
	ENDIF		

In this example, the statements between IF and ENDIF are assembled because HERE and THERE share the same base.

Label	Operation	Operand	
	SECTION	SEC2	
HERE	BLOCK	100H	
	COMMON	WKSPACE	
THERE	BLOCK	100H	
	.		
	.		
	IF	BASE (HERE, THERE)	} Not assembled because HERE and THERE are not in same section
	.		
	.		
	ENDIF		

In this example, the statements between IF and ENDIF are not assembled because HERE and THERE do not share the same base.

Label	Operation	Operand	
THERE	BLOCK	100H	
	.		
	.		
	IF	BASE (\$, THERE)	} Only assembled if THERE is in the current section
	.		
	.		
	ENDIF		

In this example, the statements between IF and ENDIF are assembled if THERE is in the current section. The dollar sign (\$) represents the current value of the location counter.

SYNTAX

BITS(bit-source,start-bit,length)

PARAMETERS

- bit-source** Any expression that results in a numeric value, either scalar or address. The expression may not be an unbound global.
- start-bit** The position of a bit in the **bit-source**: any expression that results in a positive numeric value which is less than or equal to 31. However, the sum of **start-bit** and **length** must be less than or equal to 32.
- length** The length of the **bit-string**: any expression that results in a positive numeric value which is less than or equal to 32. However, the sum of **start-bit** and **length** must be less than or equal to 32.

EXPLANATION

The BITS function returns a bit string which is a substring of **bit-source**. The bits in **bit-source** are numbered from 0 to 31: bit 0 is on the right, and bit 31 is on the left. The substring consists of bits starting from **start-bit** and continuing to the left until the substring contains **length** bits.

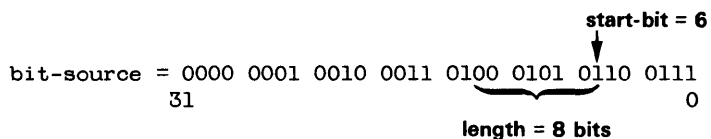
The following table shows various bit strings returned by the BITS function.

Expression	Bit String
BITS(101001B,0,3)	001
BITS(101001B,3,3)	101
BITS(101001B,6,3)	000
BITS(101001B,0,0)	0

If **bit-source** is a relocatable address, no further assembly time operations may be performed on the bit string that BITS produces. This is because the function must wait until link time to be evaluated.

EXAMPLE

Label	Operation	Operand
BITE	MOV	AL,#BITS(01234567H,6,8)



BITE = 0001 0101 = 15H

SYNTAX
DEF(symbol)

PARAMETERS

symbol Any user-defined symbol.

EXPLANATION

The DEF function tests whether a symbol has been defined during the current assembler pass. (See the discussion of Assembler Execution, later in this section, for a description of the two passes of the assembler.) A value of -1 (true) is returned if the symbol has been defined. A value of 0 (false) is returned if the symbol has not been defined.

EXAMPLE

Label	Operation	Operand
;Q	EQU	0
	.	
	.	
	IF	DEF(Q)
	WORD	15
	BYTE	5
	ENDIF	

In this example, the semi-colon (;) in the first line flags the line as a comment and the line is not assembled. Thus, the statements after the IF directive are not assembled, since Q has not been defined in the current assembler pass. If the semicolon is removed, the IF condition becomes true and the statements are assembled.

Label	Operation	Operand
	IF	DEF(P)
	BLOCK	P
	ENDIF	
P	SET	100

In this example, the statement within the IF block (BLOCK P) will not be assembled because P is not defined until further on in the source code.

SYNTAX**ENDOF**(section-name)**PARAMETERS**

section-name The name of a section defined in the assembler source program.

EXPLANATION

The ENDOF function returns the address of the last byte of a section. The linker may relocate the individual sections during linking. Therefore, the ENDOF function is evaluated at link time. (The Linker section of this manual discusses how sections are relocated.) Further arithmetic operations may not be performed on the result of an ENDOF function.

EXAMPLE

Label	Operation	Operand
	RESERVE	STACK,100
	MOV	SP,ENDOF(STACK)

This 8086/8088 example reserves 100 bytes for the stack (STACK) and loads the stack pointer register (SP) with the address of the end of the stack. The stack pointer register holds the effective address of the high byte of memory reserved for STACK.

SYNTAX

HI(numeric-expression)

PARAMETERS

numeric-expression Any expression that returns a numeric value, either scalar or address.

EXPLANATION

The HI function returns the most significant byte in the low-order word of a numeric expression. The result is a one-byte numeric value equivalent to BITS(numeric-expression,8,8). The numeric expression may be either an address or a scalar value. If the expression is an address, further operations may not be performed on the result.

EXAMPLE

Label	Operation	Operand
	SECTION	TABLE, INPAGE
Q	BLOCK	50
	SECTION	MAIN
	MOV	BH, HI (TABLE)
	.	.
	.	.
	MOV	BL, LO(Q)
	MOV	AX, [BX]

In this 8086/8088 example, the high byte of the 16-bit address of the section TABLE is loaded into the BH register. The low byte of address Q is then loaded into the BL register. Data can be transferred without changing the BL register.

SYNTAX

LO(numeric-expression)

PARAMETERS

numeric-expression Any expression that results in a numeric value, either scalar or address.

EXPLANATION

The LO function returns the least significant byte of a numeric expression. The result is a one-byte numeric value equivalent to **BITS**(numeric-expression,0,8). The numeric expression may be either an address or a scalar value. If the expression is an address, further operations may not be performed on the result.

EXAMPLE

See the HI function example.

SYNTAX

NCHR(string-expression)

PARAMETERS

string-expression Any expression that returns a string.

EXPLANATION

The NCHR function returns the current number of characters in the specified string. The result is a scalar value.

NOTE

The current length of a character string is not necessarily the same as the maximum length of a string symbol as declared with the STRING directive. See the Assembler Directives section of this manual for information on the STRING directive.

EXAMPLE

The following example shows one use of the NCHR function within a macro repeat block:

Label	Operation	Operand
	STRING	STR(5)
STR	SET	'HELLO'
Q	SET	1
	REPEAT	Q <= NCHR(STR)
	ASCII	SEG(STR,Q,1), ' '
Q	SET	Q + 1
	ENDR	

The repeat loop is repeated for Q = 1, 2, 3, 4, and 5. When Q = 6, the REPEAT condition is false and the assembly continues with the statement following ENDR. The ASCII representations of the individual characters 'H E L L O ' are stored in consecutive bytes.

SYNTAX**SCALAR**(address)**PARAMETERS**

address Any expression that returns an address value.

EXPLANATION

The SCALAR function converts an address (unsigned numeric) value into a scalar (signed numeric) value.

The only arithmetic operations that can be performed directly on address values are: addition with a scalar value, subtraction of a scalar from an address, and subtraction of addresses. To perform any other operations on address values, you must first convert the addresses to scalar values with the SCALAR function.

EXAMPLE

Label	Operation	Operand
TABLE	BLOCK	100
XXX	EQU	SCALAR(TABLE) / 2 + TABLE

This example shows how an address (TABLE) is converted to a scalar so that the division operator may be applied to it.

Note that XXX is an address because the sum of a scalar and an address (TABLE) is an address.

SYNTAX

SEG(string,start-position,char-count)

PARAMETERS

- string Any expression that returns a character string.
- start-position A numeric expression that indicates the position in the string of the first character of the substring counting from the left. Must be greater than zero.
- char-count Any numeric expression that evaluates to the number of characters to be returned. Must be positive.

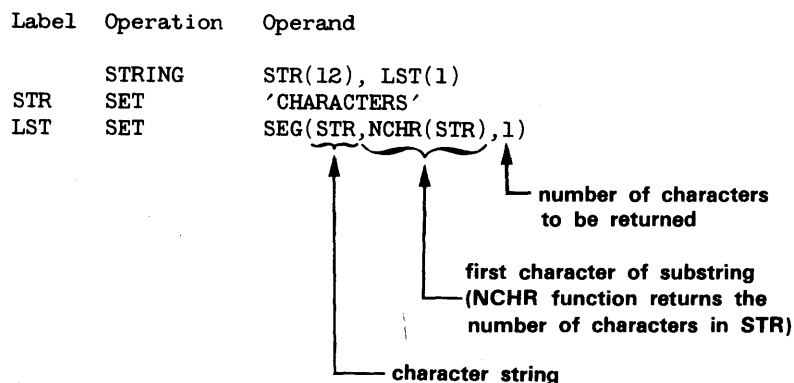
EXPLANATION

The SEG function returns a substring of a character string. The first character in the substring is the character in the **start-position**. Each successive character is included, counting from the left, until **char-count** characters are included or the end of the string is encountered.

The following table shows various substrings returned by the SEG function:

Expression	Substring
SEG('ABCDE',2,2)	'BC'
SEG('ABCDE',4,3)	'DE'
SEG('ABCDE',6,1)	"
SEG('ABCDE',1,6)	'ABCDE'

EXAMPLE



Although the character string STR has a maximum length of 12, NCHR(STR) returns the current length which is 10. The **start-position** of the substring is the tenth character. The **char-count** is 1. Thus, the tenth character 'S' is assigned to the string variable LST.

SYNTAX

STRING(scalar)

PARAMETERS

scalar Any expression that evaluates to a scalar value.

EXPLANATION

The **STRING** function converts a scalar value to its string representation. The string representation is 11 characters long. The first character is a zero or minus (-), depending on the sign of the number. The remaining ten characters are the decimal representation of the value, padded with leading zeros (if necessary). The following table shows how values are converted to their string representation.

Value	String
0	'00000000000'
-1	'-0000000001'
400	'00000000400'
200H	'00000000512'

EXAMPLE

```

Label  Operation  Operand
      STRING      MATSIZE(6), DIGIT4(1)
XVAL   SET        4
YVAL   SET        50
      .
      .
      .
MATSIZE SET      STRING(XVAL * YVAL)
DIGIT4  SET      SEG(MATSIZE,9,1)
  
```

This example converts the value of **XVAL** times **YVAL** ($4 * 50 = 200$) to the string '0000000200'. **DIGIT4** is defined to be the ninth character in the string **MATSIZE** ('2').

NOTE

*Note that there is also a **STRING** directive, which is quite different from the **STRING** function.*

SYNTAX**STRINGOF**(argument-number)**PARAMETERS**

argument-number Any expression that results in a valid macro argument number.

EXPLANATION

The **STRINGOF** function returns the macro argument (in string form) whose number is **argument-number**, if the function is used inside a macro. If the function is used outside a macro, the argument number is invalid, or the argument number is greater than the number of parameters passed, a null string is returned.

EXAMPLE

Label	Operation	Operand
	MACRO	MACRONAME
	STRING	ARG1(20),ARG2(20),ARG3(20)
ARG1	SET	STRINGOF(1)
ARG2	SET	STRINGOF(2)
ARG3	SET	STRINGOF(3)
	.	
	.	
ENDM		
	.	
	.	
	MACRONAME	'These','are','arguments' ← macro invocation
	.	
	.	

In this example, ARG1, ARG2, and ARG3 are set to be 'These', 'are', and 'arguments', respectively, when the macro MACRONAME is called with those parameters.

ASSEMBLER EXECUTION

Two Passes

The assembler makes two passes through the input. During the first pass, the assembler examines each statement, records any symbol it encounters in a symbol table, and assigns a value to each symbol. That value is used in the second pass.

When the END statement or the end of the last source file is encountered, the assembler reads the input again. During the second pass, the assembler:

- generates an object module,
- generates a listing file, and
- lists on the terminal any error messages generated.

Forward Referencing

Since the assembler generates a symbol table on the first pass, your programs can include forward referencing. For example:

```

                JMP     DOWN
                .
                .
DOWN          CALL    OUTS

```

The symbol DOWN can be referenced before it is defined. If any symbol has a different value during the second pass, a phase error results.

Execution Sequence

As the assembler reads each statement of the source program, it first makes any necessary **text substitution**. The assembler replaces any text substitution construct, such as "1", "@", or "VARNAME", with the parameter, symbol, or string that the construct stands for.

The next action depends on the type of statement:

- **assembly language instruction**—The assembler translates each assembly language instruction into the corresponding machine instruction.
- **assembler directive**—The assembler performs the action specified by the directive. Not all assembler directives produce object code. (See the Assembler Directives section of this manual for the effect of individual directives.)
- **macro invocation**—The assembler processes each statement within the previously defined macro. (See the Macros section of this manual for more information about macros.)

ASSEMBLER OUTPUT

The assembler generates an object module and an assembler listing. Any assembler errors are displayed on the system terminal.

Object Module

The assembler generates an object module which is stored in binary format. This assembler-created object module is suitable for one of the following uses:

- It may be linked with other modules to form an executable load file. (See the Linker section of this manual.)
- It may be inserted into a library file. (See the Library Generator section of this manual.)
- It may be loaded into program memory and executed provided that the module does not contain any unbound global symbols and does not contain any sections that must be relocated. (See the Linker section of this manual for information on relocatable sections.)

Assembler Listing

The assembler generates an assembler listing consisting of three parts: the source listing, the symbol table, and the cross-reference listing. Figures 2-2, 2-3, and 2-4 show the assembler listing of a sample program. Both the listing and the sample program that generates it are examined in more detail later in this section.

The assembler listing shown in this section consists of four pages: pages 1 and 2 (Fig. 2-2) show the source listing, which includes the source program and the object code generated for each statement; page 3 (Fig. 2-3) shows the symbol table; and page 4 (Fig. 2-4) shows the cross-reference listing. Refer to Figs. 2-2, 2-3, and 2-4 as you read the following descriptions.

Source Listing

Each line of the source listing (Fig. 2-2) contains the following five fields:

1. The **line number** (decimal). This value is incremented with each line processed (including blank lines and expanded macros, REPEAT blocks, and INCLUDE files).
2. The value of the **location counter** (hexadecimal) appears if the statement generates object code or alters the location counter. The value shown is the value before the statement is processed.
3. The assembled **object code** (hexadecimal) or **result-value**. (Object code is left justified in this field; result-values are right justified.) For some directives, the assembler generates a result-value in this field. Table 2-4 lists those directives and an explanation of the result-values.
4. The **line indicator**:
 - G = unbound global
 - I = text from INCLUDE file
 - M = text from macro or REPEAT expansion
 - R = relocatable
 - S = text substitution

This field provides miscellaneous information about the line and may be empty.

5. The **source statement**.

If any statement contains an error, the appropriate error message appears on the line directly after the statement. See the Error Messages section of this manual for further information about error messages.

**Table 2-4
Result-Values**

Directive	Explanation of Result-Value
BLOCK	Indicates the size (in bytes) of the block being reserved.
ELSEIF	Indicates whether the condition-value is false (0) or true (anything but 0).
EQU	Indicates the value of the symbol being assigned.
IF	Indicates whether the condition-value is false (0) or true (anything but 0).
ORG	Indicates the new value of the location counter.
SET	Indicates the value of the symbol being assigned or, if the symbol is a string, indicates the length of the string (in bytes).

Cross-Reference Listing

The cross-reference listing (Fig. 2-3) is a listing of all the user-defined symbols and the line numbers of each statement in which they appear. Only those symbols encountered while the XREF option is on will appear in the cross-reference listing. The symbols are listed in alphabetical order. If the symbol appears in the label field, that line number is followed by a pound sign. Symbols in comments are not cross-referenced.

Symbol Table

The assembler symbol table (Fig 2-4) displays the value and type of each symbol. The symbol table is divided into the following groups:

1. **Scalars**—Scalar symbols are listed in this group. The letter 'V' indicates a variable defined with the SET directive. The absence of a 'V' indicates the variable was defined with an EQU directive. The number (hexadecimal) that follows the symbol is the value assigned to the symbol. The value for each variable is the last value assigned to the variable during assembly.
2. **Strings and Macros**—Symbols that are declared as string variables or defined as macro names are listed in this group. The letter 'S' associated with the symbol indicates a string variable and 'M' indicates a macro. A number (in hexadecimal) follows each string variable. That number represents the number of bytes required by the assembler to store the character string.
3. **Sections**—Each section of the program is listed alphabetically within this group. The following information appears with each section:
 - Section type—SECTION, RESERVE, or COMMON. See the Linker section of this manual for the definition of section types.
 - Relocation type—Absolute, Aligned, Page, Inpage, or Byte Relocatable.
 - Length of section—the number of bytes of object code generated (in hexadecimal).
 - Class name—if specified.
 - All address symbols within the section—with the address of each symbol, relative to the beginning of the section. 'E' indicates that the END OF function is used to determine the address. 'T' indicates that BITS, HI, or LO is used. 'G' indicates that the symbol is a bound global.
4. **Unbound Globals**—Global symbols used in this module but defined elsewhere. Any symbols based on an unbound global are listed in this group.
5. **Undefined Symbols**—Any undefined symbols encountered by the assembler are listed at the end of the symbol table.
6. **Statistics**—Three summary lines of statistics appear at the end of the symbol table. The first line shows the number of source lines read. The second line shows the number of lines processed, including expanded REPEAT blocks, macros and INCLUDE files. The quantity in the second line is never less than that in the first line. The third line lists the number of errors. A fourth line, showing the number of undefined symbols, will appear if any undefined symbols were encountered in the source file. These lines of statistics also appear on the system terminal at the end of the assembly process.

```

Tektronix ASM 8086/8088          SAMPLE PROGRAM          Page 1
Vxx.xx-xx (xxxx)                xxxxxxxxxxxxxxxxxxxxxx

2          LIST      LINE(80),XREF
3          STRING   VOTERS(20),MYSELF(20)
4          STRING   SENTENCE(40)
5          3E8      SEATS  SET      1000
6          A        MYSELF SET      'KEN DEDATE'
7          9        VOTERS SET      'ENGINEERS'
8          C6      CONTRIB SET      198
9          ; DEFINE RESERVE SECTION 'SEATING'.
10         FFFFFFFF IF      HI(CONTRIB) = 0
11         WARNING ; CONTRIBUTION TOO SMALL
*** ASM: 1 (W)
12         1F4      SEATS  SET      SEATS - 500
13         ENDF
14         RESERVE SEATING,SEATS
15
16         ; DEFINE MACRO 'PROMISE'.
17         MACRO   PROMISE
18         ; THIS MACRO CONCATENATES ALL PARAMETERS INTO
19         ; A SINGLE SENTENCE.
20         SENTENCE SET ''
21         PARAM   SET      1 ; POINT TO FIRST PHRASE.
22         REPEAT  PARAM <= "#" ; REPEAT
23         SENTENCE SET      SENTENCE:' ': "PARAM" ; FOR
24         PARAM   SET      PARAM + 1 ; EACH
25         ENDR    ; PHRASE.
26         ASCII  "'SENTENCE'"
27         ENDM
28
29 00000000 00000000 DELIBERATE ERROR
*** ASM: 107(E) Undefined opcode "DELIBERATE"
0000
30         ; DEFINE PROGRAM SECTION 'CAMPAIGN'.
31         GLOBAL  SPEAK,KISSBABY
32         SECTION CAMPAIGN
33         E R     ELECTION EQU   ENDOF(CAMPAIGN)
34         1 R     NEXTBABY EQU   KISSBABY + 1
35 00000000 9A000000 R FIRST  CALLS  SPEAK,SPEAK
00
36 00000005 9A000000 R THEN   CALLS  KISSBABY,KISSBABY
00
37 0000000A EA010000 R LAST   JMPS   NEXTBABY,KISSBABY
00
38         ; DEFINE COMMON SECTION 'SPEECH'.
39         COMMON  SPEECH,ABSOLUTE
40         ORG    100H
41 00000100 80      APPLAUSE BLOCK 80H
42         180 R   MESSAGE EQU   $
43         PROMISE VOTERS, 'WILL ALWAYS HAVE'

```

} macro definition

line number	location counter	object code	line indicator	source text
-------------	------------------	-------------	----------------	-------------

Fig. 2-2. Sample assembler source listing (part 1 of 2).

This sample assembler listing, and the source program that generated it, are discussed in the text.

```

Tektronix ASM 8086/8088          SAMPLE PROGRAM          Page 2
Vxx.xx-xx (xxxx)                xxxxxxxxxxxxxxxxxxxx

57 00000180 20454E47 MS          ASCII ' ENGINEERS WILL ALWAYS HAVE'
    494E4545 M
    52532057 M
    494C4C20 M
    414C5741 M
    59532048 M
    415645 M

59                                PROMISE 'A FRIEND IN',MYSELF,','
77 0000019B 20412046 MS          ASCII ' A FRIEND IN KEN DEDATE . .'
    5249454E M
    4420494E M
    204B454E M
    20444544 M
    41544520 M
    2E

79                                PROMISE 'TELL YOUR FELLOW',VOTERS
93 000001B4 2054454C MS          ASCII ' TELL YOUR FELLOW ENGINEERS'
    4C20594F M
    55522046 M
    454C4C4F M
    5720454E M
    47494E45 M
    455253 M

95                                LIST ME ; SHOW FULL MACRO EXPANSION
96                                PROMISE 'TO VOTE FOR',MYSELF,','
97                                M ; THIS MACRO CONCATENATES ALL PARAMETERS INTO
98                                M ; A SINGLE SENTENCE.
99                                O M SENTENCE SET ''
100                               1 M PARAM SET 1 ; POINT TO FIRST PHRASE.
101                               MS REPEAT PARAM <= 3 ; REPEAT
102                               C MS SENTENCE SET SENTENCE: ' ': 'TO VOTE FOR' ; FO
103                               2 M PARAM SET PARAM + 1 ; EACH
106                               17 MS SENTENCE SET SENTENCE: ' ': MYSELF ; FOR
107                               3 M PARAM SET PARAM + 1 ; EACH
110                               19 MS SENTENCE SET SENTENCE: ' ': ' ' ; FOR
111                               4 M PARAM SET PARAM + 1 ; EACH
114 000001CF 20544F20 MS          ASCII ' TO VOTE FOR KEN DEDATE . .'
    564F5445 M
    20464F52 M
    204B454E M
    20444544 M
    41544520 M
    2E M

115                                ENDM
116                                END
    
```

} complete macro expansion listed

line	location	object	line	source
number	counter	code	indicator	text

3856-6

Fig. 2-2. Sample assembler source listing (part 2 of 2).

This sample assembler listing, and the source program that generated it, are discussed in the text.

Tektronix ASM 8086/8088 Vxx.xx-xx (xxxx)		CROSS REFERENCE						Page 3 XXXXXXXXXXXXXXXXXXXX	
APPLAUSE-----	41#								
CAMPAIGN-----	32	33							
CONTRIB-----	8#	10							
DELIBERATE-----	29								
ELECTION-----	33#								
FIRST-----	35#								
KISSBABY-----	31	34	36	36	37				
LAST-----	37#								
MESSAGE-----	42#								
MYSELF-----	3	6#	59	69	96	106			
NEXTBABY-----	34#	37							
PARAM-----	47#	48	49	50#	50	52	53	54#	
	54	56	63#	64	65	66#	66	68	
	69	70#	70	72	73	74#	74	76	
	83#	84	85	86#	86	88	89	90#	
	90	92	100#	101	102	103#	103	105	
	106	107#	107	109	110	111#	111	113	
PROMISE-----	17	43	59	79	96				
SEATING-----	14								
SEATS-----	5#	12#	12	14					
SENTENCE-----	4	20#	23#	23	26	46#	49#	49	
	53#	53	57	62#	65#	65	69#	69	
	73#	73	77	82#	85#	85	89#	89	
	93	99#	102#	102	106#	106	110#	110	
	114								
SPEECH-----	39								
SPEAK-----	31	35	35						
THEN-----	36#								
VOTERS-----	3	7#	43	79					

3856-7

Fig. 2-3. Sample assembler cross-reference listing.

This sample assembler listing, and the source program that generated it, are discussed in the text.

Tektronix ASM 8086/8088		SYMBOL TABLE		Page 4
Vxx.xx-xx (xxxx)				XXXXXXXXXXXXXXXXXXXX
Scalars				}
CONTRIB-----000000C6 V	PARAM-----00000004 V			
SEATS-----000001F4 V				
Strings & Macros				}
MYSELF-----00000014 S	PROMISE----- M			
SENTENCE-----00000028 S	VOTERS-----00000014 S			
Section = %SAMPLEASM, Aligned to 00000010, Size = 00000008				}
Section = CAMPAIGN, Aligned to 00000010, Size = 0000000E				
ELECTION-----0000000E E	FIRST-----00000000			}
LAST-----0000000A	THEN-----00000005			
Reserve Section = SEATING, Aligned to 00000010, Size = 000001F4				}
Common Section = SPEECH, Absolute, Size = 000001E8				
APPLAUSE-----00000100	MESSAGE-----00000180			}
Unbound Globals				
KISSBABY-----00000000	NEXTBABY-----00000001			}
SPEAK-----00000000				
Undefined Symbols				}
DELIBERATE				
48 Lines Read				}
116 Lines Processed				
2 Errors				
1 Undefined Symbols				

3856-8

Fig. 2-4. Sample assembler symbol table listing.

This sample assembler listing, and the source program that generated it, are discussed in the text.

Sample Source Program

Figure 2-5 shows the sample source program that generated the assembler listing shown in Figs. 2-2, 2-3, and 2-4. The program has no practical application, but is purposely contrived to illustrate a variety of listing features.

```

        TITLE 'SAMPLE PROGRAM'
        LIST LINE(80),XREF
        STRING VOTERS(20),MYSELF(20)
        STRING SENTENCE(40)
SEATS   SET      1000
MYSELF  SET      'KEN DEDATE'
VOTERS  SET      'ENGINEERS'
CONTRIB SET      198
; DEFINE RESERVE SECTION 'SEATING'.
        IF      HI(CONTRIB) = 0
WARNING ; CONTRIBUTION TOO SMALL
SEATS   SET      SEATS - 500
        ENDF
        RESERVE SEATING,SEATS

; DEFINE MACRO 'PROMISE'.
        MACRO  PROMISE
; THIS MACRO CONCATENATES ALL PARAMETERS INTO
; A SINGLE SENTENCE.
SENTENCE SET    ''
PARAM     SET    1 ; POINT TO FIRST PHRASE.
REPEAT   PARAM <= "#" ; REPEAT
SENTENCE SET    SENTENCE:' ': "PARAM" ; FOR
PARAM    SET    PARAM + 1 ; EACH
        ENDR    ; PHRASE.
        ASCII  ""SENTENCE""
        ENDM

        DELIBERATE ERROR
; DEFINE PROGRAM SECTION 'CAMPAIGN'.
        GLOBAL SPEAK,KISSBABY
        SECTION CAMPAIGN
ELECTION EQU   ENDOF(CAMPAIGN)
NEXTBABY EQU   KISSBABY + 1
FIRST  CALLS   SPEAK,SPEAK
THEN   CALLS   KISSBABY,KISSBABY
LAST   JMPS    NEXTBABY,KISSBABY
; DEFINE COMMON SECTION 'SPEECH'.
        COMMON SPEECH,ABSOLUTE
        ORG    100H
APPLAUSE BLOCK 80H
MESSAGE EQU    $
        PROMISE VOTERS,'WILL ALWAYS HAVE'
        PROMISE 'A FRIEND IN',MYSELF,'.'
        PROMISE 'TELL YOUR FELLOW',VOTERS
        LIST   ME ; SHOW FULL MACRO EXPANSION.
        PROMISE 'TO VOTE FOR',MYSELF,'.'
        END

```

3856-9

Fig. 2-5. Sample 8086/8088 source program.

This source program generated the sample assembler listing that was shown in Figs. 2-2, 2-3, and 2-4. The text discusses each line in this source program, and the object code that it generates.

Sample Source Listing

Let's compare the source program (Fig. 2-5) with the assembler listing (Fig. 2-2). The first line of the source program is:

```
TITLE 'SAMPLE PROGRAM'
```

The TITLE directive creates a title on each page of the assembler program listing. The TITLE directive itself does not appear on the program listing and does not generate any object code.

```
Tektronix ASM 8086/8088      SAMPLE PROGRAM      Page 1
                             └──────────┬──────────┘
                             title
```

The next statement in the source program is:

```
LIST LINE(80),XREF
```

The LIST directive controls various features of the assembler listing. The LINE(80) option prints the assembler listing in a 80-character width instead of the default width. The XREF option specifies that the cross-reference is to be included in the listing. Although this line appears in the assembler listing, it does not generate object code.

```
STRING VOTERS(20),MYSELF(20)
STRING SENTENCE(40)
```

These two lines of source code declare the symbols VOTERS, MYSELF, and SENTENCE as string variables. These lines do not generate object code. The variables appear in the symbol table of the assembler listing (Fig. 2-4). The 'S' following each symbol indicates that it is a string variable.

```
SEATS SET 1000
MYSELF SET 'KEN DEDATE'
VOTERS SET 'ENGINEERS'
CONTRIB SET 198
```

The SET directive assigns a value to a variable. In the first of these four SET statements, a numeric value is assigned to the numeric variable SEATS. The value 1000 (decimal) appears in the object code column (Fig. 2-2, line 5) as 3E8 hexadecimal. No memory location appears on the line because the value is not stored in the object program. MYSELF and VOTERS are string variables. They are assigned the values enclosed in single quotes. The length of these strings (in bytes) is shown in the listing as A and 9, respectively. The numeric value 198 (C6H) is assigned to the numeric variable CONTRIB.

```
; DEFINE RESERVE SECTION 'SEATING'.
```

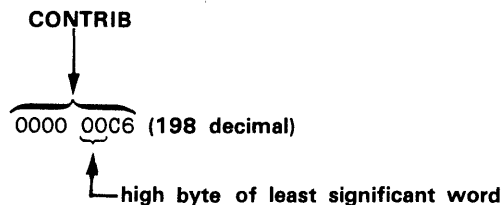
The semicolon (;) designates this line as a comment line. Comment lines appear in the assembler listing, but have no effect on the object code.


```

                IF      HI(CONTRIB) = 0
                WARNING ; CONTRIBUTION TOO SMALL
SEATS          SET      SEATS - 500
                ENDF

```

These four statements are a conditional assembly block. The IF directive causes the block of statements between the IF and ENDF to be assembled if the condition is true. In this case, the condition 'HI(CONTRIB) = 0' is evaluated. The current value of the variable CONTRIB is:



The function HI(CONTRIB) returns the value 00, the high byte of the least significant word of CONTRIB. Since the condition value of the IF statement is true (shown in the listing by the value FFFFFFFF), the block is assembled and the statements appear in the assembler listing (Fig. 2-2, lines 10-13). The WARNING directive generates a user-defined error message. This message appears both on the terminal display during assembly and in the assembler listing.

```
SEATS          SET      SEATS - 500
```

The SET directive changes the value of the symbol SEATS from 3E8H (1000 decimal) to 1F4H (1000-500 decimal). See line 12 of the assembler listing.

```
RESERVE SEATING, SEATS
```

This assembler directive reserves a section in memory. The section is named SEATING and is allocated 01F4H bytes (the current value of SEATS). The section SEATING appears in the symbol table (Fig. 2-4), with the word 'Reserve' identifying the type of section.

Next, notice the blank line in the sample program. A blank line has no effect on the object code, but it does generate a line in the assembler listing.

```
; DEFINE MACRO 'PROMISE'.
```

Although this comment line appears in the assembler listing (Fig. 2-2, line 16), it has no effect on the object code.

```

                MACRO  PROMISE
; THIS MACRO CONCATENATES ALL PARAMETERS INTO
; A SINGLE SENTENCE.
SENTENCE SET   ""
PARAM         SET    1    ; POINT TO FIRST PHRASE.
                REPEAT PARAM <= ""#""    ; REPEAT
SENTENCE SET   SENTENCE: ' ': "PARAM" ; FOR
PARAM         SET    PARAM + 1          ; EACH
                ENDR    ; PHRASE.
                ASCII  ""SENTENCE""
                ENDM

```

This block of source code is a macro definition. The location of the statements in a macro definition is stored by the assembler. When the macro is invoked, the statements within the macro are assembled, generating any indicated object code. The macro will be explained later, when it is invoked.

Another blank line in the program code improves the readability of the program, setting the macro definition apart, but has no effect on the assembler.

```
DELIBERATE ERROR
```

This line is an invalid statement: DELIBERATE, which appears in the operation field, is not an assembly language instruction, an assembler directive, or a macro invocation. Eight bytes of zeros are generated as object code and an error message is printed on the terminal and listed in the assembler listing (Fig. 2-2, line 29).

```
; DEFINE PROGRAM SECTION 'CAMPAIGN'.
```

This line is another comment line and has no effect on the object code.

```
GLOBAL SPEAK,KISSBABY
```

The assembler directive GLOBAL declares SPEAK and KISSBABY to be global symbols. They are unbound globals; that is, they are used in this module, but defined elsewhere. No object code is produced.

```
SECTION CAMPAIGN
```

The assembler directive SECTION begins the definition of program section CAMPAIGN. The lines of source code following this statement define the section.

```
ELECTION EQU      ENDOF(CAMPAIGN)
```

The assembler directive EQU assigns a value to the symbol ELECTION. The ENDOF function returns the address of the last byte of the section CAMPAIGN. This line appears in the assembler listing as:

```
33      E R      ELECTION EQU      ENDOF(CAMPAIGN)
          ↑
        relocation indicator
```

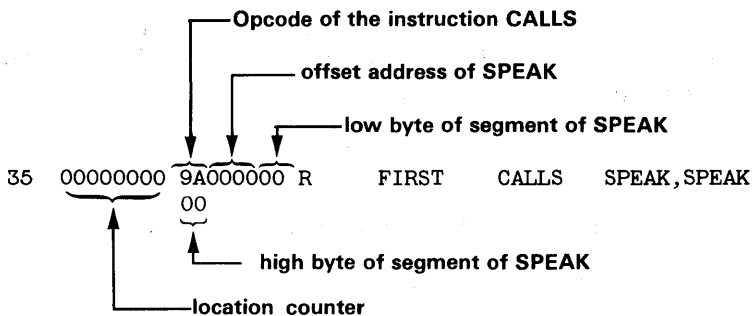
The relocation indicator (R) shows that the object code for this source line (an address) will be adjusted by the linker at link time. Since the section CAMPAIGN is relocatable, the address of the last byte is not determined until link time. The hexadecimal value E is the value assigned to ELECTION, telling us that there are 15 bytes (00000000-0000000E) in the section. (The beginning address of every relocatable section is 00000000 at assembly time.)

```
NEXTBABY EQU      KISSBABY + 1
```

The assembler directive EQU assigns a value to the symbol NEXTBABY. The value assigned (KISSBABY + 1) is dependent on the value of the unbound global KISSBABY. In the assembler listing (Fig. 2-2, line 34), the relocation indicator again shows that the object code will be adjusted by the linker. The 1 indicates that the adjusted address will be +1 relative to the address of KISSBABY.

```
FIRST CALLS SPEAK, SPEAK
```

This 8086/8088 assembly language instruction calls the subroutine SPEAK. The assembler listing shows the object code that is generated:



Since this is the first statement in section CAMPAIGN that produces object code, the memory location assigned is 00000000. 9A is the opcode for the instruction CALLS. Since SPEAK is an unbound global variable, it does not have an address in this module. (The dummy value 0000 appears in the object code for the address and the segment.) The 'R' indicates that the address and segment values will be adjusted at link time.

```
THEN CALLS KISSBABY, KISSBABY
```

This statement calls the subroutine KISSBABY, another unbound global. In the assembler listing (Fig.2-2, line 36), the memory location is 00000005, since the previous instruction (CALLS SPEAK, SPEAK) occupies bytes 00000000-00000004.

```
LAST JMPS NEXTBABY, KISSBABY
```

This 8086/8088 assembly language instruction generates the object code EA01000000. (See line 37 in the assembler listing.) EA is the opcode for the instruction JMPS. NEXTBABY has the value 1. This value will be adjusted by the linker, depending on the address of the section KISSBABY.

```
; DEFINE COMMON SECTION 'SPEECH'.
```

This is another comment line.

```
COMMON SPEECH, ABSOLUTE
ORG 100H
```

The assembler directive COMMON declares the next block of statements to be a new section of type COMMON. The name of the section is SPEECH and it is an absolute section. The ORG statement defines the location of the first byte of the section to be 100H.

```
APPLAUSE BLOCK 80H
```

This statement generates the first byte of the common section SPEECH. The memory location of the first byte is 00000100H.

```

41 00000100      80      APPLAUSE BLOCK 80H
      ^
      |
      |-----location counter

```

This BLOCK directive reserves a block of 80H bytes. The symbol APPLAUSE represents the address of the first byte of the block (00000100).

```
MESSAGE EQU $
```

The EQU directive assigns a value to the symbol MESSAGE. The dollar sign (\$) in the operation field returns the value of the location counter. The assembler listing shows that the value 0180 was assigned to MESSAGE.

```
42          180 R      MESSAGE EQU $
```

The location counter was advanced to 180H when the directive 'BLOCK 80H' was assembled. MESSAGE represents the address of the next byte of object code to be generated.

```
PROMISE VOTERS, 'WILL ALWAYS HAVE'
```

This statement invokes the macro PROMISE, which was previously defined. There are two macro parameters: (1) the symbol VOTERS and (2) the character string 'WILL ALWAYS HAVE'. This single source line generates eight lines in the assembler listing:

```

43          PROMISE VOTERS, 'WILL ALWAYS HAVE'
57 00000180 20454E47 MS      ASCII ' ENGINEERS WILL ALWAYS HAVE'
          494E4545 M
          52532057 M
          494C4C20 M
          414C5741 M
          59532048 M
          415645  M
          ^
          |
          |-----ASCII representation of ' ENGINEERS WILL ALWAYS HAVE'

```

↖ text substitution indicator

When the macro is invoked, the assembler processes the lines of the macro definition. The assembler listing shows us only the one source line that generates object code, namely:

```
ASCII ' ENGINEERS WILL ALWAYS HAVE'
```

Notice that the line number jumps from 43 to 57. The line counter is incremented with every line processed. More of these lines would have been listed if the ME/MEG list option was set to ME. (This is done for the last macro invocation in this sample program.)

At this time, let's look at the statements in the macro definition:

```
SENTENCE SET ''
```

This SET directive assigns the null string ('') to SENTENCE.

```
PARAM SET 1 ; POINT TO FIRST PHRASE.
```

This SET directive assigns the value 1 to the numeric variable PARAM.

```
REPEAT PARAM <= "#" ; REPEAT
SENTENCE SET SENTENCE:' ': "PARAM" ; FOR
PARAM SET PARAM + 1 ; EACH
ENDR ; PHRASE.
```

This block of statements (a repeat block) is assembled repeatedly until the REPEAT operand (PARAM <= "#") is false. When a macro is assembled, the "#" is replaced with the number of parameters passed from the macro invocation. In this statement, the "#" is replaced with 2 (two parameters), so the block of statements is repeated twice. (See 'Determining Parameter Count' in the Macros section of this manual.)

The first time the block is assembled "PARAM" is replaced with VOTERS, since PARAM has the value 1 and VOTERS is the first parameter. The second statement in the block concatenates the current value of the string variable SENTENCE (""), a space (' '), and the value of VOTERS ('ENGINEERS'); the resulting string is assigned to SENTENCE. SENTENCE now has the value of:

```
' ENGINEERS'
```

The next statement increments the current value of PARAM by one. PARAM now holds the value 2. Since the repeat condition (PARAM <= "#") is still true, the block of statements is repeated. This time, "PARAM" is replaced with 'WILL ALWAYS HAVE', the second parameter. The statement concatenates the current value of SENTENCE (' ENGINEERS'), a space (' '), and the character string 'WILL ALWAYS HAVE'. SENTENCE now has the value of:

```
' ENGINEERS WILL ALWAYS HAVE'
```

PARAM is incremented to 3. The repeat condition is no longer true, so the assembly continues with the statement following the ENDR:

```
ASCII ""SENTENCE""
```

This statement generates object code and is therefore listed in the assembler listing. The object code generated is the ASCII representation of each character of the string in the operand field. The assembler first makes the text substitution indicated by the double quotes (" "). ""SENTENCE"" is replaced with ' ENGINEERS WILL ALWAYS HAVE'. Notice that the text substitution is shown on the source listing, along with the text substitution indicator (S) and the macro expansion indicator (M).

Assembly continues with the statement following the macro invocation.

```
PROMISE 'A FRIEND IN', MYSELF, '.'
```

This statement invokes the macro PROMISE again. This invocation has three parameters: (1) the character string 'A FRIEND IN', (2) the symbol MYSELF, and (3) the string '.'. The resulting object code is the ASCII representation of ' A FRIEND IN KEN DEDATE .'.

```
PROMISE 'TELL YOUR FELLOW', VOTERS
```

This next statement invokes the macro PROMISE with two parameters, the string 'TELL YOUR FELLOW' and the symbol VOTERS. The resulting object code is the ASCII representation of:

```
' TELL YOUR FELLOW ENGINEERS'
```

The next statement in the sample program is:

```
LIST ME ; SHOW FULL MACRO EXPANSION.
```

The LIST directive turns on various features of the assembler listing. This statement sets the ME/MEG option to the ME setting. Under the ME setting, when a macro is invoked, the assembler listing shows all of the assembled statements of the macro expansion.

```
PROMISE 'TO VOTE FOR',MYSELF,'.'
```

This macro invocation returns the ASCII representation of 'TO VOTE FOR KEN DEDATE.' Notice in the assembler listing starting on line 101, that the text substitution indicator appears on five lines. The "#" is replaced by '3' and "PARAM" is replaced by the appropriate parameter.

Also notice in the assembler listing that a character is missing from the end of the line:

```
102          C MS SENTENCE SET SENTENCE: ' ': 'TO VOTE FOR' ; FO
```

In the source program, the comment was '; FOR'. The 'R' does not appear in the source listing because the LIST LINE(80) directive had previously instructed the assembler to truncate the listing to 80 characters.

The last statement of the source code is:

```
END
```

This statement marks the end of the source program.

Sample Cross-Reference Listing

The sample cross-reference listing shown in Fig. 2-3 lists all the user-defined symbols in our sample program. Matched with these symbols are the line numbers of the lines in which they appear (except those in comments). If the symbol appears in the label field of a statement, a pound sign (#) is printed after that number.

Let's look at a particular example:

```
CONTRIB----- 8# 10
```

This entry shows that CONTRIB is used on lines 8 and 10. The # shows that CONTRIB appears in the label field of line 8.

```
PARAM----- 47# 48 49 50# 50 52 53 54#
              54 56 63# 64 65 66# 66 68
              69 70# 70 72 73 74# 74 76
              83# 84 85 86# 86 88 89 90#
              90 92 100# 101 102 103# 103 105
              106 107# 107 109 110 111# 111 113
```

This entry shows that PARAM appears on many more lines than shown in the assembler source listing (Fig. 2-2). This is because, as explained in the discussion of the sample symbol table, some lines that were processed were not listed. Variables in substitution constructs are also cross-referenced.

```
110          19 MS      SENTENCE SET      SENTENCE: ' ':'. ' ; FOR
```

In this line, "PARAM" was replaced by the third parameter (PARAM was equal to 3 at this point). This is the reason that the number 110 appears in the cross-reference listing for PARAM.

Note, also, that some line numbers are listed more than once for some variables. There is a cross-reference for each occurrence of a symbol.

Sample Symbol Table

Now let's examine the symbol table for the sample program (Fig. 2-4).

Listed under **Scalars** are the scalar symbols used in the program (CONTRIB, PARAM, and SEATS). Each variable is listed with the last value assigned to it. The 'V' indicates that the values were assigned with the SET directive.

Listed under **Strings and Macros** are four symbols: MYSELF, PROMISE, SENTENCE, and VOTERS. The 'S' indicates that MYSELF, SENTENCE, and VOTERS are string variables. The 'M' indicates that PROMISE is a macro. After each string variable is a number that indicates the number of bytes (hexadecimal) reserved for that variable. This number is equal to the number of bytes specified in the string declaration (or default length, if none is specified).

There are four **Sections** in our program: %SAMPLEASM, CAMPAIGN, SEATING, and SPEECH.
 Section = %SAMPLEASM, Aligned to 00000010, Size = 00000008

This line shows that there were 8 bytes in an unnamed section. An unnamed section is the result of the generation of object code before any SECTION or COMMON directives are encountered. In this case, the 8 bytes were generated when the following line was processed.

```
DELIBERATE ERROR
```

The default name for an unnamed section is derived as follows:

1. Eliminate all characters except letters and digits from the name of the object file specified in the assembler invocation line.
2. Take the first 15 characters.
3. Add the prefix '%'.

The name of the object file which contains the object code generated from the sample source program is SAMPLE.ASM. After removing the period and adding a percent sign, the default name is %SAMPLEASM.

By default, all sections generated for the 8086/8088 microprocessor are aligned on 16-byte boundaries (addresses that are multiples of 16).

```
Section = CAMPAIGN, Aligned to 00000010, Size = 0000000E
```

```
ELECTION-----0000000E E   FIRST-----00000000
LAST-----0000000A       THEN-----00000005
```

In this section summary, the name of the section is **CAMPAIGN**, which is of type 'Section'. The section is aligned on 16-byte boundaries. It is **E (14)** bytes long. The addresses of the four symbols, **ELECTION**, **FIRST**, **LAST**, and **THEN**, are relative to the base address of the section and are subject to relocation, since the section is relocatable. The 'E' that follows the symbol 'ELECTION' indicates that the **ENDOF** function is used to determine the value.

Section **SEATING** is a 'Reserve' section that is **01F4** (hexadecimal) bytes long. Section **SPEECH** is a 'Common' section that is not relocatable (absolute) and is **01E8H** bytes long, including the 100H-byte gap at the beginning of the section.

In our sample program, the symbols **KISSBABY** and **SPEAK** are the only **unbound globals**. **NEXTBABY** is defined with an **unbound global**, so it is listed here too. Since **NEXTBABY** was defined by adding 1 to **KISSBABY**, **NEXTBABY** contains the value 1. At link time, the value of **KISSBABY** will be added to **NEXTBABY**.

Let's look at the lines of statistics:

```
48 Lines Read
116 Lines Processed
2 Errors
1 Undefined Symbols
```

There are more **Lines Processed (116)** than **Lines Read (48)** because the macro invocations and **REPEAT** block cause some of the source lines to be assembled more than once.

There are two **Errors** listed for this sample program: (1) the user-defined warning, and (2) the error generated by the line 'DELIBERATE ERROR'. **DELIBERATE** is the **Undefined Symbol**.

Section 3 ASSEMBLER DIRECTIVES

ASSEMBLER DIRECTIVES INDEX	Page
Listing Control Directives	
LIST—Turns on listing options	3-31
NOLIST—Turns off listing options	3-39
PAGE—Skips to new page in listing	3-45
SPACE—Inserts blank lines into listing	3-58
STITLE—Creates listing subtitle	3-59
TITLE—Creates listing title	3-62
WARNING—Displays warning	3-64
Symbol Definition Directives	
EQU—Assigns value to symbol	3-18
SET—Assigns value to variable	3-55
STRING—Declares string variable(s)	3-61
Data Storage Control Directives	
ADDRESS—Initializes memory with data in address format	3-3
ASCII—Generates ASCII data	3-5
BLOCK—Reserves block of memory	3-6
BYTE—Generates byte(s) of data	3-7
FLOAT—Initializes memory with data in floating point format	3-21
LONG—Initializes memory with 32-bit value(s)	3-36
WORD—Generates word(s) of data	3-65
Macro Definition Directives	
ENDM—Ends macro definition	3-16
EXITM—Stops macro expansion	3-19
MACRO—Begins macro definition	3-37
File Inclusion Directive	
INCLUDE—Assembles source code from another file	3-30
Conditional Assembly Directives	
ELSE—Begins alternate conditional block	3-12
ELSEIF—Begins alternate conditional block	3-13
ENDIF—Ends IF block	3-15
ENDR—Ends REPEAT block	3-17
EXITR—Stops repeat process	3-20
IF—Begins conditional assembly block	3-26
REPEAT—Begins repetitive assembly	3-46

Module Definition Directives	Page
COMMON—Declares common section	3-8
END—Ends source module	3-14
GLOBAL—Declares global symbol(s)	3-24
NAME—Declares object module name	3-38
ORG—Sets location counter	3-40
RESERVE—Reserves section of memory	3-50
RESUME—Resumes definition of section	3-52
SECTION—Declares program section	3-53

TABLES

Table No.	Table	Page
3-1	LIST Options	3-34

ILLUSTRATIONS

Fig. No.	Figure	Page
3-1	COMMON directive example	3-10
3-2	Single precision format	3-22
3-3	Double precision format	3-22
3-4	Allowed forms of IF block nesting	3-28
3-5	Sections before linking	3-42
3-6	Sections after linking	3-43
3-7	Allowed forms of REPEAT block nesting ...	3-47

Section 3

ASSEMBLER DIRECTIVES

INTRODUCTION

This section describes the directives you may use with the TEKTRONIX 8500 Series B Assembler. The directives are arranged in alphabetical order for easy reference. A functional index appears at the front of this section to help you when you do not know a directive by name.

Each assembler directive description may consist of any or all of the following parts: a syntax block, parameter definitions, an explanation of the use and limits of the directive, and one or more examples of its use.

The syntax block shows the required format of the directive. Assembler directive statements may contain information in any of the four fields: label, operation, operand, and comment. Since the comment field is strictly optional for any directive, it does not appear in the syntax block.

The syntax blocks in this section use the notation conventions explained in the Learning Guide of this manual. For example:

SYNTAX		
Label	Operation	Operand
[symbol]	DIRECT	expression[,expression]...

The above example shows the syntax for **DIRECT**, a fictitious directive. You may interpret this syntax block as follows:

- A label is optional for this directive.
- The operation field must contain the word 'DIRECT'.
- The operand field must contain at least one expression. If two or more expressions are entered, they must be separated by commas. The number of expressions is limited only by the maximum line length of 127 characters plus RETURN. (The line may be a maximum of 131 characters after text substitution.)

LABELS

For each assembler directive, a label may be required, optional, or prohibited, depending on the directive.

- Only the EQU and SET directives **require labels**. EQU and SET each assign the value in the operand field to the symbol in the label field.
- The following directives **must not have a label**. These directives do not affect the location counter.

ELSE	ENDM	EXITR	NAME	REPEAT	STRING
ELSEIF	ENDR	LIST	NOLIST	SPACE	TITLE
ENDIF	EXITM	MACRO	PAGE	STITLE	WARNING

- The following directives generate object code and therefore **often have labels**. The label is assigned the address of the first byte of code generated.

ADDRESS	BLOCK	FLOAT	WORD
ASCII	BYTE	LONG	

- The following directives affect the location counter but do not generate object code, so they **do not normally have labels**. The value assigned to the label depends on the directive.

COMMON	ORG	RESERVE	RESUME	SECTION
--------	-----	---------	--------	---------

- The following directives do not generate object code nor do they affect the location counter and so **do not normally have labels**. The label, if any, takes the current value of the location counter. In the dictionary entry for each of these directives, the label is shown as optional but is not discussed as a parameter.

END	GLOBAL	IF	INCLUDE
-----	--------	----	---------

See The Assembler section of this manual for information on forming labels (user-defined symbols).

SYNTAX

Label	Operation	Operand
[symbol]	ADDRESS	expression[,expression]...

PARAMETERS

- symbol** A user-defined label representing the address of the first byte of data.
- expression** Any label or expression in proper format. Some microprocessors require different parameters. See the Assembler Specifics section for information about your microprocessor.

EXPLANATION

The **ADDRESS** directive stores the specified expression(s), in address format, in consecutive bytes of the object file. The address format is microprocessor-dependent. Using this directive ensures that the data stored conforms to address format conventions for your microprocessor.

EXAMPLES

Label	Operation	Operand
JAN	ASCII	'JANUARY', 0
FEB	ASCII	'FEBRUARY', 0
MAR	ASCII	'MARCH', 0
APR	ASCII	'APRIL', 0
MAY	ASCII	'MAY', 0
JUN	ASCII	'JUNE', 0
JUL	ASCII	'JULY', 0
AUG	ASCII	'AUGUST', 0
SEP	ASCII	'SEPTEMBER', 0
OCT	ASCII	'OCTOBER', 0
NOV	ASCII	'NOVEMBER', 0
DEC	ASCII	'DECEMBER', 0
MONTHS	ADDRESS	0000, JAN, FEB, MAR, APR, MAY, JUN
	ADDRESS	JUL, AUG, SEP, OCT, NOV, DEC

ADDRESS

Initializes memory with data in address format

These statements store 12 null-terminated strings and an array of pointers to the strings. The appropriate string may be referenced indirectly through the corresponding pointer. For example:

Label	Operation	Operand
	MOV	SI, MONTHS[BX]
STRLOOP	LODB	
	.	
	.	
	LOOPNE	STRLOOP

These 8086/8088 statements could be used inside a subroutine that processes strings. The program would loop and test for the null character until the entire string had been processed.

SYNTAX		
Label	Operation	Operand
[symbol]	ASCII	expression[,expression]...

PARAMETERS

symbol A user-defined label representing the address of the first character in the string or the first byte of data.

expression Any expression that yields either a string value or a scalar in the range -128 to 255.

EXPLANATION

The ASCII directive stores the ASCII codes for the characters of the specified string(s) in consecutive bytes of the object program. Refer to the Tables section of this manual for an ASCII conversion table.

If the expression yields a numeric value, the action taken is identical to the BYTE directive.

EXAMPLES

Label	Operation	Operand
CHESMEN	ASCII	'PAWN ROOK KNIGHT'
	ASCII	'BISHOP', 'QUEEN ', 'KING

These two statements generate 36 consecutive bytes of ASCII code: one 18-character string and three 6-character strings, stored as a single 36-character sequence. CHESMEN is the address of the first character from the first string. The following hexadecimal object code is generated:

```

source: P A W N           R O O K           K N I G H T
object: 50 41 57 4E 20 20 52 4F 4F 4B 20 20 4B 4E 49 47 48 54
source: B I S H O P Q U E E N       K I N G
object: 42 49 53 48 4F 50 51 55 45 45 4E 20 4B 49 4E 47 20 20
    
```


BLOCK

Reserves block of memory

SYNTAX

Label	Operation	Operand
[symbol]	BLOCK	byte-count

PARAMETERS

symbol A user-defined label that represents the address of the first byte of the block.

byte-count The number of bytes to be reserved: any positive scalar expression.

EXPLANATION

The **BLOCK** directive reserves a specified number of bytes. It does not initialize memory. **BLOCK** is used primarily to allocate memory for data that may change during program execution.

The byte-count expression must yield a positive scalar value. Every symbol in the expression must have been defined previously.

EXAMPLES

Label	Operation	Operand
LASTNAME	BLOCK	20
SSN	BLOCK	11
AGE	BLOCK	1
SALARY	BLOCK	2

These statements allocate space for a 20-character name, an 11-character social security number, an age in the range 0 to 255, and a salary in the range 0 to 65535.

SYNTAX		
Label	Operation	Operand
[symbol]	BYTE	byte-value[,byte-value]...

PARAMETERS

- symbol A user-defined label that represents the address of the first byte of data.
- byte-value Any expression that yields a scalar in the range -128 to 255.

EXPLANATION

The **BYTE** directive stores the specified values in consecutive bytes of the object program. If a value is outside the range -128 to 255, only the least significant byte is stored and a truncation error is issued.

The **byte-value** parameter may be a single character enclosed in single quotes. In this case the ASCII value of the character is stored in memory. If the string length exceeds one character, the ASCII value of the last character is stored and a truncation error is generated (See example 2).

EXAMPLES

BYTE Example 1

Label	Operation	Operand
MONTHS	BYTE	31,28,31,30,31,30
	BYTE	31,31,30,31,30,31

In this example, 12 bytes of object code are generated. The Nth byte contains the number of days in the Nth month. MONTHS is the address of the first byte.

BYTE Example 2

Label	Operation	Operand	Comment
ME	BYTE	'I','AM','A','PROGRAMMER'	;ONLY 4 BYTES STORED

In this example, four bytes are stored; the last character of each string is converted to its ASCII representation and stored in memory. A truncation error is generated.

```
source: I M A R
object: 49 4D 41 52
```

SYNTAX		
Label	Operation	Operand
[symbol]	COMMON	section-name[,relocation-type][, CLASS =class-name]

PARAMETERS

symbol	A user-defined label (usually omitted) that represents the address of the first byte of the common section.
section-name	The name assigned to the section.
relocation-type	<p>An option to direct the relocation of the section at link time. You may specify one of the following relocation types:</p> <p>PAGE—The common section is relocated to the beginning of a page of memory. A frequently used page size is 256 bytes. See the Assembler Specifics section of this manual for the page size for your microprocessor.</p> <p>ALIGN(address-mod)—The common section may be relocated to any address that is a multiple of the address-mod. Address-mod represents a positive scalar expression. Each symbol in the expression must have been defined previously.</p> <p>INPAGE—The common section may be relocated to any address, so long as the entire section lies within one page of memory.</p> <p>ABSOLUTE—The section is not relocated. You may not assign a class name to an absolute section.</p> <p>If you do not specify PAGE, ALIGN, INPAGE, or ABSOLUTE, the section is attributed with the default relocation type. This default relocation type is microprocessor-dependent. See the Assembler specifics section for information about the default relocation type for your microprocessor.</p>
class-name	The class name assigned to the section. You may not assign a class name to an absolute section.

EXPLANATION

The **COMMON** directive declares a section of type **COMMON** and defines the name, relocation type, and class name of the section. The contents of the section are defined by the statements following the **COMMON** directive, up to the next **SECTION**, **COMMON**, or **RESUME** directive.

The class name is **not** used by the assembler. It is passed to the linker so that the linker commands may refer to this section by its class name rather than its section name. Refer to the Linker section of this manual for further information on class names.

Different source modules may declare the same common section, and thus share the contents of that section. (See Example 1.) The relocation type of the section must be the same in every module in which the section is declared.

The linker assigns the same starting address to all common sections with the same name. Memory is allocated for the largest section with that name. (See Example 2.)

You may use the directives ADDRESS, ASCII, BYTE, FLOAT, LONG, or WORD to initialize values in a common section. (See Example 3.) If two or more modules specify values for the same location in a common section, the module loaded last takes precedence; the error is not flagged.

The name of a common section is a global symbol whose value is the address of the first byte of the section. A section name should not be declared with a GLOBAL directive in any module in which the section is defined with a COMMON directive.

EXAMPLES

COMMON Example 1

This example illustrates how program modules can communicate with each other through values stored in a common section.

Assume that source modules A, B, and C each contain the following common section definition:

Label	Operation	Operand
	COMMON	CUSTOMER
CNAME	BLOCK	30
ADDRESS	BLOCK	30
CITY	BLOCK	16
STATE	BLOCK	2

During program execution, module A might define the customer's name, module B might define the address, and module C might define the city and state. All 78 bytes of customer information in the common section may be used or changed by any of the three modules.

COMMON Example 2

A common section may also be used as a scratch area. Some subroutines use blocks of memory for temporary storage. If all modules use the same common section for temporary storage, less memory is required than if each module uses a different block of memory.

This example illustrates:

- how a common section may be used as a scratch area by one or more modules; and
- how the linker treats common sections with the same name but different lengths.

In source module A, the following statements define common section SCRATCH:

Label	Operation	Operand
	COMMON	SCRATCH
X1	BLOCK	4
X2	BLOCK	6

In source module B, SCRATCH is defined as follows:

Label	Operation	Operand
	COMMON	SCRATCH
Y1	BLOCK	5
Y2	BLOCK	10

At link time, one area of memory is allocated to section SCRATCH. The size of the area is 15 bytes—the length of the larger section named SCRATCH. Both subroutines may use this area of memory.

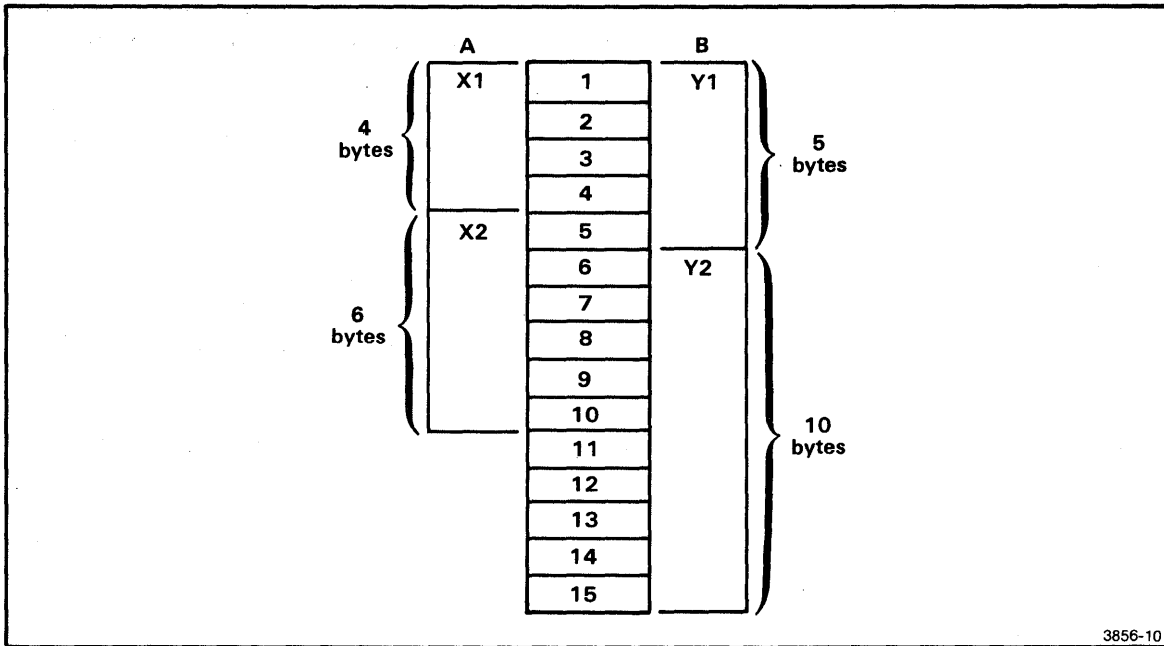


Fig. 3-1. COMMON directive example.

COMMON Example 3

This example demonstrates how you may initialize data in a common section.

Source module A defines common section CALENDAR and provides text for array DAYS:

Label	Operation	Operand
	COMMON	CALENDAR
MONTHS	BLOCK	36
DAYS	ASCII	'SUNMONTUEWED'
	ASCII	'THUFRISAT'

Source module B also defines CALENDAR and provides text for array MONTHS:

	COMMON	CALENDAR
MONTHS	ASCII	'JANFEBMARAPR'
	ASCII	'MAYJUNJULAUG'
	ASCII	'SEPOCTNOVDEC'
DAYS	BLOCK	21

Modules A and B both specify the same length for common section CALENDAR (57 bytes).

When the section is loaded into memory, its contents will be as follows:

bytes 1-9	source: J A N F E B M A R	}	MONTHS
	object: 4A 41 4E 46 45 42 4D 41 52		
bytes 10-18	source: A P R M A Y J U N		
	object: 41 50 52 4D 41 59 4A 55 4E		
bytes 19-27	source: J U L A U G S E P	}	
	object: 4A 55 4C 41 55 47 53 45 50		
bytes 28-36	source: O C T N O V D E C	}	
	object: 4F 43 54 4E 4F 56 44 45 43		
bytes 37-45	source: S U N M O N T U E	}	DAYS
	object: 53 55 4E 4D 4F 4E 54 55 45		
bytes 46-54	source: W E D T H U F R I		
	object: 57 45 44 54 48 55 46 52 49		
bytes 55-57	source: S A T	}	
	object: 53 41 54		

SYNTAX

Label	Operation	Operand
	ELSE	

EXPLANATION

The ELSE directive marks the beginning of an ELSE sub-block. The end of this sub-block is marked by an ENDIF directive. Statements within the sub-block are assembled if **all** prior condition-values within the present IF block were false (zero or undefined). Otherwise, the statements in the ELSE sub-block are not assembled.

Only one ELSE directive per corresponding IF directive is allowed.

See the IF directive for more information.

EXAMPLES

Label	Operation	Operand	Comment
	IF	YEAR MOD 4 = 0	
NDAYS	EQU	366	; LEAP YEAR ← Assembled if YEAR is divisible by 4.
	ELSE		
NDAYS	EQU	365	; NOT LEAP YEAR ← Assembled if YEAR is not divisible by 4.
	ENDIF		

If the value of YEAR is evenly divisible by 4, the first EQU directive is assembled and the symbol NDAYS is assigned the value 366. Otherwise the second EQU directive is assembled and NDAYS takes the value 365.

SYNTAX

Label	Operation	Operand
	ELSEIF	condition-value

PARAMETERS

condition-value Any expression that yields a numeric value. The condition is considered false if the value is zero or undefined and true if the value is nonzero.

EXPLANATION

The ELSEIF directive marks the beginning of an ELSEIF sub-block. The end of this sub-block is marked by either another ELSEIF directive, an ELSE directive, or an ENDIF directive. Statements within the sub-block are assembled if **all** prior condition-values within the present IF block were false (zero or undefined) **and** the condition-value associated with the ELSEIF is true.

See the IF directive for more information.

END

Ends source module

SYNTAX

Label	Operation	Operand
[symbol]	END	[transfer-address]

PARAMETERS

transfer-address The address of the first instruction to be executed.

EXPLANATION

The END directive marks the end of the source module. If the source module contains no END directive, assembly continues to the end of the last source file named in the assembler invocation command.

The transfer address, if present, is the address of the first instruction to be executed when the program is run. The transfer address is usually specified in a source module, often in the module that contains the main program. However, the transfer address can also be defined or changed at link time. (See the Linker section of this manual.) If more than one module contains a transfer address, the transfer address in the first module linked is used.

EXAMPLES

Label	Operation	Operand
START	MOV	AX,#VALUE
	.	
	END	START

In this example, END is the last statement in the main program source module. START is the transfer address: program execution starts with the 8086/8088 MOV instruction.

SYNTAX

Label	Operation	Operand
	ENDIF	

EXPLANATION

The **ENDIF** directive marks the end of an **IF** block of statements. See the **IF** directive for more information.

ENDM

Ends macro definition

SYNTAX

Label	Operation	Operand
	ENDM	

EXPLANATION

The ENDM directive marks the end of a macro definition. See the MACRO directive for more information.

SYNTAX

Label	Operation	Operand
	ENDR	

EXPLANATION

The ENDR directive marks the end of a REPEAT block of statements. See the REPEAT directive for more information.

SYNTAX

Label	Operation	Operand
symbol	EQU	expression

PARAMETERS

symbol A user-defined symbol to be assigned a value by this statement.

expression Any expression that yields a numeric value.

EXPLANATION

The EQU directive assigns a value to a symbol. All other attributes are also transferred (for example, relocation type). The symbol cannot be redefined in the same source module.

A symbol defined in an EQU directive may be used by any statement in the module, with the following restriction: the EQU directive **must** precede any BLOCK, ELSEIF, EQU, IF, ORG, REPEAT, SET, or STRING directive that refers to the symbol that is defined by the EQU directive.

EXAMPLES

Label	Operation	Operand	Comment
	MOV	AL,#ROWS	; NUMBER OF ROWS TO AL REGISTER.
	MOV	AH,#COLS	; NUMBER OF COLUMNS TO AH REGISTER.
	.	.	.
ROWS	EQU	10	; DEFINE NUMBER OF ROWS...
COLS	EQU	3	; ... AND NUMBER OF COLUMNS.
	.	.	.
TABLE	BLOCK	ROWS*COLS	; ALLOT SPACE FOR A 30-BYTE TABLE.

The symbol ROWS is assigned the value 10 and the symbol COLS is assigned the value 3. Note that the two 8086/8088 MOV instructions may refer to ROWS and COLS, even though the symbols are not defined until later in the module. On the other hand, the BLOCK directive that refers to the symbols must follow the EQU directives that define the symbols.

SYNTAX

Label	Operation	Operand
	EXITM	

EXPLANATION

The EXITM directive terminates the current macro **expansion**. Note that EXITM does **not** mark the end of a macro **definition**.

EXITM is valid only in macros. It is generally used to stop macro expansion in the middle of an IF block or REPEAT block.

EXAMPLES

Label	Operation	Operand	Comment
	MACRO	TESTBYTE	
PARAM	SET	1	; POINT TO FIRST PARAMETER.
	REPEAT	PARAM <= "#"	; DO FOR EVERY PARAMETER:
	IF	"PARAM" < 0	; IF PARAMETER IS BAD...
		WARNING ; NEGATIVE PARAMETER	
	EXITM		; ... ABORT MACRO EXPANSION.
	ELSE		; OTHERWISE STORE THE VALUE.
	BYTE	"PARAM"	
	ENDIF		
PARAM	SET	PARAM + 1	; INCREMENT PARAMETER POINTER...
	ENDR		; ... AND REPEAT.
	ENDM		

Macro TESTBYTE generates one BYTE directive for each parameter in the macro invocation. The variable PARAM counts from 1 to the number of parameters passed ("#"). The construct "PARAM" is replaced by the parameter pointed to by PARAM. If a negative parameter is encountered, the WARNING and EXITM directives are assembled and macro expansion ends before all parameters have been processed.

The macro invocation

```
TESTBYTE 10,20,-1,-2,30
```

yields the following macro expansion:

```
BYTE 10
BYTE 20
WARNING ; NEGATIVE PARAMETER
```

If the EXITM statement were omitted, macro expansion would continue until all parameters were processed:

```
BYTE 10
BYTE 20
WARNING ; NEGATIVE PARAMETER
WARNING ; NEGATIVE PARAMETER
BYTE 30
```

SYNTAX

Label	Operation	Operand
	EXITR	

EXPLANATION

The EXITR directive terminates the most currently active REPEAT process. It does **not** mark the end of a REPEAT block. The EXITR directive may only be used inside of a REPEAT block.

The EXITR directive is generally used in conjunction with an IF directive to provide control over a REPEAT process.

See the REPEAT directive for more information.

SYNTAX

Label	Operation	Operand
[symbol]	FLOAT	mantissa [$\left\{ \begin{array}{l} D \\ E \end{array} \right\}$ exponent] [,mantissa [$\left\{ \begin{array}{l} D \\ E \end{array} \right\}$ exponent]] ...

PARAMETERS

- symbol** A user-defined label representing the address of the first byte of data.
- mantissa** A series of one or more digits with an optional decimal point; optionally preceded by a plus or minus sign.
- exponent** A series of digits preceded by an optional plus or minus sign.

EXPLANATION

The **FLOAT** directive initializes memory with data in floating point format. The specified floating point value may be contained in 32 bits or 64 bits, depending upon which precision flag is used (E = single precision, D = double precision).

Format

Any decimal number can be expressed as a binary number in the form:

$$1. fffff... * 2^n \text{ (where } f \text{ is a binary digit)}$$

Since the first digit is always a **1**, it is left off and assumed. This allows one additional bit of accuracy in representing the mantissa. This string of binary digits, without the leading **1.**, is called the significand.

In order to be able to represent negative exponents, a number, called the bias, is added to the exponent before being stored in the exponent field (X). For single precision, the bias is 127; for double precision, the bias is 1023.

The most significant bit is the sign bit. This bit is a one if the quantity is negative and zero if the quantity is positive.

FLOAT

Initializes memory with data in floating point format

Assembler Directives—8500 Series B Assembler Core Users

Figure 3-2 shows the format for single precision (32-bit).

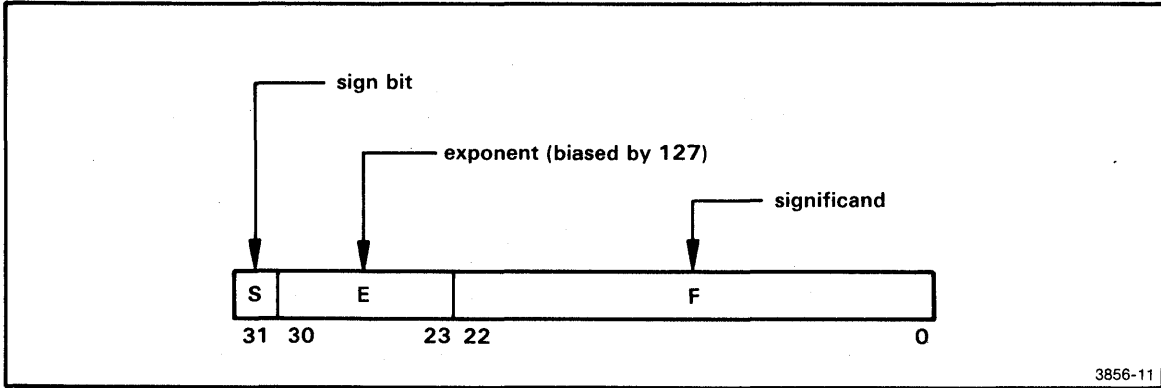


Fig. 3-2. Single precision format.

Figure 3-3 shows the format for double precision (64-bit).

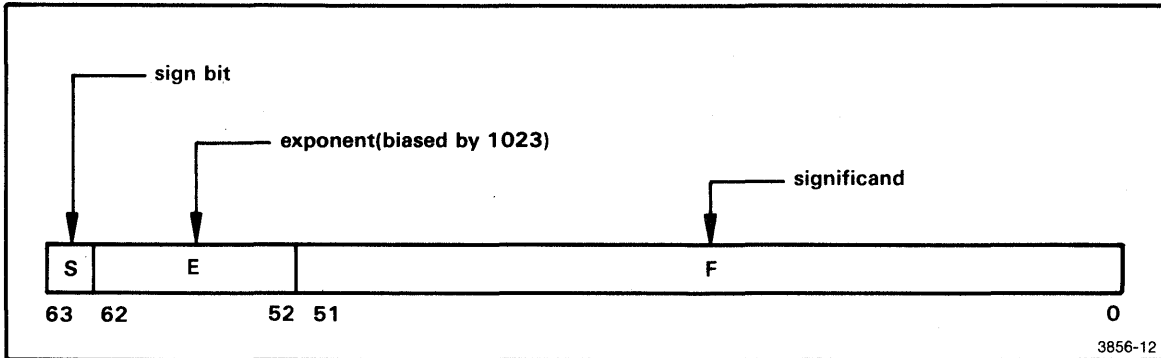


Fig. 3-3. Double precision format.

On overflow, the exponent field (X) is set to 255 (2047 for double precision) and the significand (F) is set to zero. On underflow, both X and F are set to zero.

No arithmetic operations may be performed on floating-point constants during assembly.

NOTE

If more than one floating point value is stored, it is more efficient to store them with as few FLOAT directives as possible. For example:

```
FLOAT      0,3.5E10,4E4
```

would not take as long to assemble as:

```
FLOAT      0
FLOAT      3.5E10
FLOAT      4E4
```

The floating point conversion routine must be called each time the FLOAT directive is encountered.

EXAMPLES

Label	Operation	Operand
FLOATVALUE	FLOAT	1.25E4

In this example, the value stored at FLOATVALUE is

S	X	F
0	1000100	100001101010000000000000
31	30 23 22	0

SYNTAX		
Label	Operation	Operand
[symbol]	GLOBAL	global-sym[,global-sym]...

PARAMETERS

global-sym A symbol to be declared global.

EXPLANATION

The GLOBAL directive declares one or more symbols to be global. A global symbol defined in one module may be referred to by other modules. Both the module that defines the symbol and the module that refers to it must declare the symbol to be global. The linker will make the value of the global symbol available to all modules that declare it.

The GLOBAL directive that declares a symbol must precede the statement that defines that symbol. The symbol may not be defined more than once in any group of modules to be linked.

A global symbol that is given a value in the current module is called a **bound** global. A bound global that is also an address is called an **entry point**, since it often represents an instruction that is jumped to from outside the module.

A global symbol that is not defined in the current module is called an **unbound** global; its value must be provided at link time, either by another module or by the linker command DEFINE.

A section name (defined by a COMMON, RESERVE, or SECTION directive) is a global symbol; it should not be declared with a GLOBAL directive in the same module in which the section is defined.

EXAMPLES

This example demonstrates the use of global symbols in three modules: MYMOD, HISMOD, and HERMOD.

Label	Operation	Operand
	NAME	MYMOD
	GLOBAL	HIM,HER,VALUE
	.	
VALUE	EQU	3
	CALLS	HIM,HIM
	CALLS	HER,HER
	CALL	MYSELF
	.	
MYSELF	MOV	AX,#VALUE
	.	
	.	

In module MYMOD, HIM and HER are unbound globals, but VALUE is a bound global, since it is assigned a value by the EQU directive. MYSELF does not need to be declared global, since it is defined in MYMOD (as the address of the 8086/8088 MOV instruction) and is not used in any other module.

	NAME	HISMOD
	GLOBAL	HIM, VALUE
HIM	MOV	AX, #VALUE

In module HISMOD, VALUE is an unbound global. HIM is defined as the address of the MOV instruction, so HIM is an entry point (a bound global address).

	NAME	HERMOD
	GLOBAL	HER, HIM
HER	CALLS	HIM, HIM

In module HERMOD, HIM is an unbound global. HER is defined as the address of the CALLS instruction, so HER is an entry point.

In summary:

- HIM is defined in HISMOD and used in MYMOD and HERMOD;
- HER is defined in HERMOD and used in MYMOD;
- VALUE is defined in MYMOD and used in HISMOD.

Each symbol is declared to be global wherever it is defined or used. Since MYSELF is defined in MYMOD and used only in MYMOD, it does not need to be declared global.

SYNTAX		
Label	Operation	Operand
[symbol]	IF	condition-value

PARAMETERS

condition-value Any expression that yields a numeric value. The condition is considered false if the value is zero or undefined and true if the value is nonzero.

EXPLANATION

The IF directive marks the beginning of a conditional assembly block. The end of this conditional assembly block is marked by the corresponding ENDIF directive. The blocks may be nested. Assembly is turned off if the **condition-value** is evaluated to zero or undefined.

Two optional directives that may be used to control assembly within an IF...ENDIF block are ELSE and ELSEIF. There may be any number of ELSEIF sub-blocks within an IF...ENDIF block. However, there may be only one ELSE sub-block, and it must be the last sub-block before the ENDIF directive.

The ELSE directive turns the assembly on if it has been turned off by the corresponding IF directive. The ELSEIF directive turns the assembly on if it has been turned off by the corresponding IF directive **and** the associated expression is evaluated true (nonzero). All this implies that only one group of statements will be assembled: either those following the IF statement, those following the ELSEIF statement, or those following the ELSE statement.

Since both the ELSEIF and ELSE blocks are optional, either one or both may be omitted.

IF...ENDIF

An IF...ENDIF block has the following structure:

```
IF condition-value
  (statements to be assembled
  if condition-value is true)
ENDIF
```

If the condition-value is true (nonzero), the statements between the IF directive and the ENDIF directive are assembled. If the condition-value is false (zero or undefined), those statements are skipped. (See Example 1.)

IF...ELSEIF...ELSE...ENDIF

An IF...ELSEIF...ELSE...ENDIF block has the following structure:

```
IF condition-value1  
(statements to be assembled  
if condition-value1 is true)  
ELSEIF condition-value2  
(statements to be assembled  
if condition-value1 is false  
and condition-value2 is true)  
ELSE  
(statements to be assembled  
if both condition-value 1 and  
condition-value2 are false)  
ENDIF
```

If condition-value1 is true (nonzero), the statements between the IF directive and the ELSEIF directive are assembled. If condition-value2 is true and condition-value1 is false, the statements between the ELSEIF directive and the ELSE directive are assembled. Otherwise, the statements between the ELSE directive and the ENDIF directive are assembled. (See Example 2.)

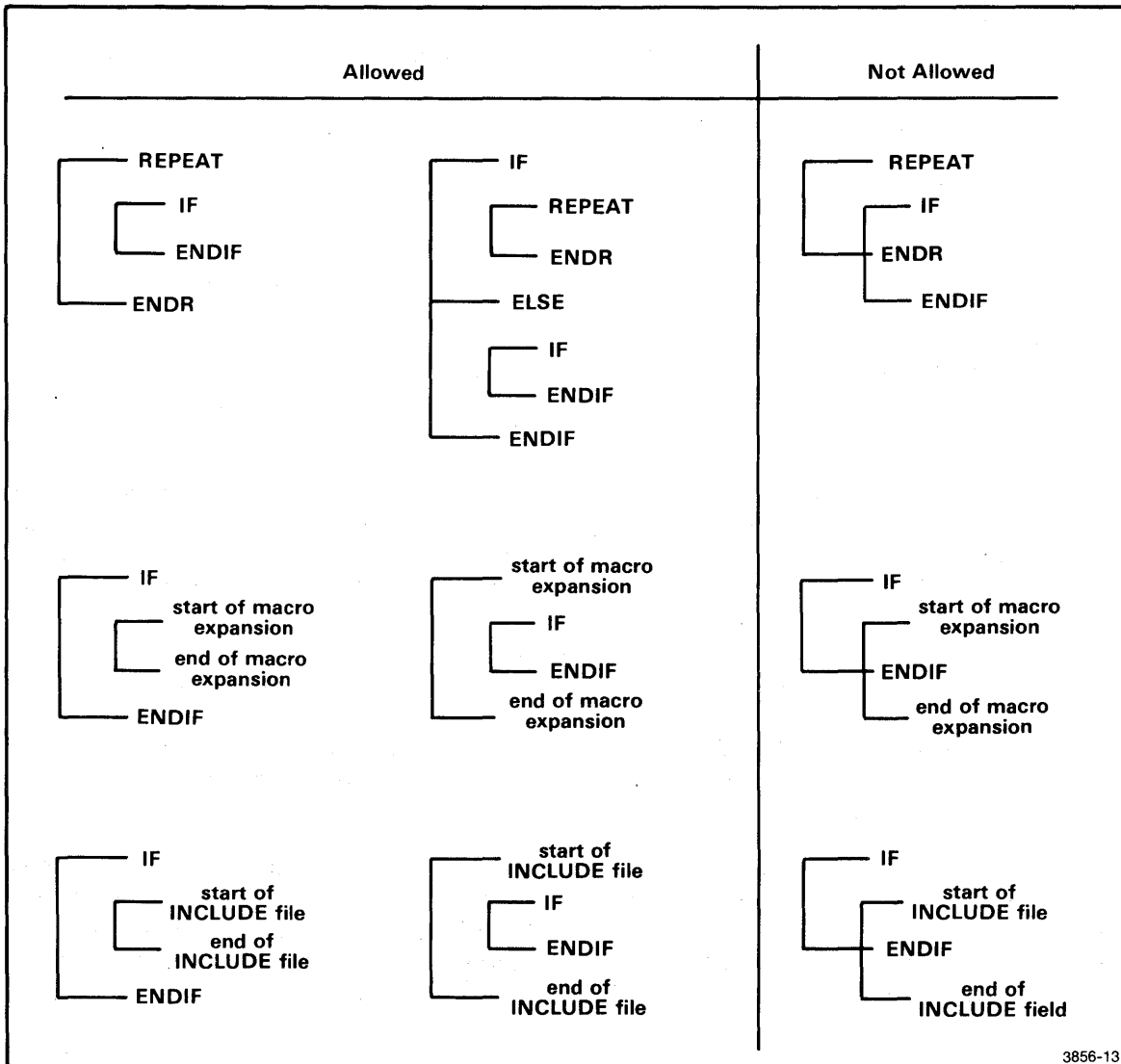
NOTE

A relational expression (for example, J <0) yields the value -1 (all bits are ones) when true and the value 0 (all bits are zeros) when false. Thus, the bit manipulation operators &(AND), !(OR), and !(XOR) may be used in complex relational expressions. (See Example 1.) The Assembler section of this manual explains expressions and operators in detail.

An IF block may be nested inside a REPEAT block or another IF block. Blocks may be nested as deep as available memory in the assembler permits. An IF block may not lie partially inside and partially outside a REPEAT block, another IF block, a macro expansion, or statements from an INCLUDE file. (Any IF block in an INCLUDE file must be wholly contained in that file.) Figure 3-4 shows the allowed forms of nesting for IF blocks.

IF

Begins conditional assembly block



3856-13

Fig. 3-4. Allowed forms of IF block nesting.

An IF block may not lie partially **inside** and partially **outside** a REPEAT block, another IF block, a macro expansion, or statements from an INCLUDE file.

EXAMPLES**IF Example 1**

Label	Operation	Operand
	IF	M>N & N<P & P=Q
	WARNING ;	TROUBLE
	ENDIF	

In this example, the conditional expression of the IF statement contains three relational subexpressions: 'M>N', 'N<P', and 'P=Q'. Each subexpression yields the value -1 (true) or 0 (false). The three subexpression values are ANDed together to yield the value (-1 or 0) to be used by the IF directive. Thus, the WARNING directive is assembled only if:

- M is greater than N, and
- N is less than P, and
- P is equal to Q.

IF Example 2

This example shows the use of an IF...ELSEIF...ELSE...ENDIF block in a macro.

Label	Operation	Operand
	MACRO	WORDS
	.	
	.	
	IF	STRINGOF(1) = 'BLUE'
COLOR	SET	1
	ELSEIF	STRINGOF(1) = 'RED'
COLOR	SET	2
	ELSEIF	STRINGOF(1) = 'YELLOW'
COLOR	SET	3
	ELSE	
COLOR	SET	0
	ENDIF	
	.	
	.	
	ENDM	

In this example, the first parameter passed to the macro is tested. If the parameter is 'BLUE', 'RED', or 'YELLOW', COLOR is set to a predetermined digit code. Otherwise, the digit code is set to zero.

INCLUDE

Assembles source code from another file

Assembler Directives—8500 Series B Assembler Core Users

SYNTAX

Label	Operation	Operand
[symbol]	INCLUDE	filespec-string

PARAMETERS

filespec-string An expression that yields a string representing a filespec.

EXPLANATION

The **INCLUDE** directive causes the assembler to process the statements in the specified file as if they were part of the current source file.

INCLUDE files may be nested (an **INCLUDE** file may contain an **INCLUDE** directive).

MACRO, **IF**, and **REPEAT** blocks begun in the original source file may **not** be terminated in an **INCLUDE** file. Also, blocks beginning in an **INCLUDE** file must be terminated in that file. Any **END** directives encountered in an **INCLUDE** file are ignored. The entire **INCLUDE** file is included.

If the **INCLUDE** directive is contained in a macro, the file is included at macro expansion time. However, statements in the **INCLUDE** file cannot use the special text substitution constructs usually allowed in macros ("N" for the Nth parameter, "#" for the number of parameters, "@" for a unique label). See the Macros section of this manual for information about these constructs.

Listing options are returned to their original values when the **INCLUDE** file is exited.

EXAMPLES

Label	Operation	Operand	Comment
	NAME	MAINMOD	
	INCLUDE	'MACR.ASM'	; DEFINE STANDARD MACROS.
	INCLUDE	'/SYS/COM.ASM'	; DEFINE COMMON BLOCK.

In this example, the statements in files **MACR.ASM** and **/SYS/COM.ASM** are assembled at the beginning of module **MAINMOD**. **MACR.ASM** contains macro definition blocks; **/SYS/COM.ASM** defines a common section.

SYNTAX

Label	Operation	Operand
	LIST	[listing-option[,listing-option]...]

PARAMETERS

listing-option One of the following listing options:

CND—Lists statements that are not assembled because of unsatisfied IF or REPEAT conditions. Defaults to OFF: Only those statements that are actually assembled are listed.

CON—Lists assembly errors on the system terminal as well as in the source listing. Defaults to ON.

DBG—Causes the assembler to generate an internal (local) symbols list in the object module. Defaults to OFF.

LINE (width)—Sets the line length to **width** characters per line. If the line is longer than **width** characters, the line is truncated. The **width** parameter must be a positive scalar. If **width** is less than 1, the line width is set to default and an error is generated. Defaults to 72 if output is to terminal, defaults to 132 otherwise.

ME—Sets the ME/MEG option to the ME setting: lists all macro expansion statements that are assembled. The ME/MEG option defaults to MEG.

MEG—Sets the ME/MEG option to MEG: lists only those macro expansion statements that generate object code. The ME/MEG option defaults to MEG.

PAGE (length)—Sets the page size to **length** lines per page. The **length** parameter must be a positive scalar and greater than or equal to 4. If **length** is less than 4, page size is set to 65536 and an error is generated. Defaults to 55.

SYM—Lists the symbol table. Defaults to ON.

XREF—Includes cross-reference information in the listing. Defaults to OFF.

If no option is specified, the **source listing** is turned ON.

EXPLANATION

The LIST directive turns on the listing option(s) named in the operand field. The NOLIST directive may be used to turn any of these options off. Line size and page size are not affected by the NOLIST directive.

Each option controls a different listing feature and may be turned on or off anywhere in the source module. If an option is changed during a macro or INCLUDE file expansion, its previous setting is restored when the expansion ends.

An assembler listing contains three parts:

1. The **source listing**, which shows the source code and object code for each statement assembled.
2. The **cross-reference listing**, which lists each symbol and its associated line number(s).
3. The **symbol table**, which lists the symbols used in the source module.

The master option (LIST/NOLIST), CND option, and ME/MEG option determine which lines of code appear in the source listing, and are discussed in the following paragraphs. The listings of the symbol table and the cross-reference table are controlled by the SYM and XREF options, respectively. The SYM and XREF options, along with CON, DBG, LINE, and PAGE are discussed under the heading **Other Options**.

Source Listing

Master Option (LIST/NOLIST). The master option (which controls the source listing) is normally ON. The directive NOLIST (without operands) turns the master option OFF, suppressing display of all statements except erroneous ones. When the master option is OFF, any PAGE and SPACE directives are suppressed, and the CND and ME/MEG options are overridden. The directive LIST (without operands) turns the master option back ON.

CND. Normally the CND option is OFF; any statement that is not assembled because of an unsatisfied IF or REPEAT condition is not listed. When the CND option is ON, even unassembled statements are listed.

ME/MEG. The ME/MEG option controls the display of statements in macro expansions. It has three settings: ME, MEG, and OFF. At the default setting, MEG, only those statements that generate object code (assembly language instructions and ADDRESS, ASCII, BLOCK, BYTE, LONG, and WORD directives) are listed. Note that other directives that directly affect the object module (COMMON, EQU, GLOBAL, NAME, ORG, RESERVE, RESUME, SECTION) are **not** listed.

The directive LIST ME changes the ME/MEG setting to ME, causing every assembled statement in a macro expansion, except IF and REPEAT conditions, to be listed. The directive NOLIST ME or NOLIST MEG turns the ME/MEG option OFF, suppressing display of all macro expansion statements except erroneous ones. The directive LIST MEG returns the ME/MEG option to its default setting.

Other Options

CON. Normally the CON option is ON, and every erroneous statement and its accompanying error message are displayed on the system terminal as well as in the source listing. When the CON option is OFF, erroneous statements and their error messages appear only in the source listing.

DBG. If the DBG option is at its default setting (OFF) when assembly ends, the object file will contain no internal (non-global) symbols list for the current module. If the DBG option is ON when assembly ends, an internal symbols list will be created, and will include all symbols in the module. The DBG option is determined by the last LIST directive in which it is specified. This option must be specified if symbolic debugging will be performed on the object module.

LINE. Normally the line size is set to 72 characters per line (if the output device is the system terminal; otherwise it is set to 132 characters per line). This includes about 32 columns containing the line number, location counter, and object code. Line size can be set to a different value with the LINE option. Lines longer than the specified line size are truncated.

PAGE. Normally the page size is set to 55 lines per page. The page size may be changed with the PAGE option. If you try to set the page size to less than four lines, an error will be generated and the page size will be set to 65535 lines.

NOTE

Make sure that you don't confuse the PAGE option in the LIST directive with the PAGE directive. The PAGE option sets the number of lines per page. The PAGE directive skips to a new page in the source listing.

SYM. If the SYM option is at its default setting (ON) when assembly ends, the assembler listing will contain the symbol table as well as the source listing. If the SYM option is OFF when assembly ends, no symbol table is listed. The symbol table is described in The Assembler section of this manual.

XREF. Normally the cross-reference will **not** be listed. If you use the XREF option in a LIST directive, however, a cross-reference listing will be generated. Only those symbols encountered while XREF is **on** will appear in the cross-reference listing. See The Assembler section of this manual for more information on the cross-reference listing.

Summary. Table 3-1 summarizes the LIST options.

LIST

Turns on listing options

Table 3-1
LIST Options

Option	Default	With LIST	With NOLIST
CND	OFF	Lists conditional assembly blocks that are not assembled.	Only lists conditional assembly blocks that are assembled.
CON	ON	Displays errors on terminal.	Does not list errors on terminal unless the terminal is the output device.
DBG	OFF	Internal symbol list is included in object file for use with linker.	Internal symbol list not included.
LINE	72,terminal 132,otherwise	Sets line size.	Illegal.
ME	OFF	Lists all macro expansions.	Suppresses listing of all macro expansions.
MEG	ON	Lists only macro expansions that produce object code.	Suppresses listing of all macro expansions.
PAGE	55	Sets page size.	Illegal.
SYM	ON	Lists symbol table.	Suppresses listing of symbol table.
XREF	OFF	Lists cross-reference	Suppresses listing of cross-reference.

EXAMPLES

```
Label Operation Operand
```

```
LIST      DBG
```

This statement causes the assembler to generate an internal symbols list for this module when it is assembled.

```
LIST      CND,ME
```

This directive causes all statements (assembled and unassembled, mainline statements and macro expansion statements) to appear in the source listing.

```
NOLIST
```

```
LIST
```

The NOLIST directive turns off the **source listing**. The LIST directive turns the source listing back on. While the **source listing** is suppressed, the settings of other options may be changed; however, changes to the CND and ME/MEG options do not become apparent until the listing is turned back on.

```
NOLIST    SYM
```

This statement suppresses display of the symbol table.

See The Assembler section of this manual for a sample listing.

LONG

Initializes memory with 32-bit value(s)

SYNTAX

Label	Operation	Operand
[symbol]	LONG	expression[,expression]...

PARAMETERS

- symbol** A user-defined label representing the address of the first byte of data.
- expression** Any expression that yields a value which may be stored in 32 bits (numbers ranging from -2,147,483,648 to 4,294,967,295 or character strings of up to 4 ASCII characters).

EXPLANATION

The **LONG** directive stores 32-bit values in 2 words (4 bytes) of contiguous memory. Depending on the microprocessor, the high and low words and/or the high and low bytes of each word may be swapped. See the Assembler Specifics section of this manual for information concerning your microprocessor.

EXAMPLES

Label	Operation	Operand
POWERS_OF_2	LONG	1,2,4,8,16,32,64,128,256,512,1024,2048,4096
	LONG	8192,16384,32768,65536,131072,262144,524288
	LONG	1048576,2097152,4194304,8388608,16777216
	LONG	33554432,67108864,134217728,268435456
	LONG	536870912,1073741824,2147483648

In this example, successive powers of 2 are stored in memory, so that a value in the form 2^n may be referenced by the expression (POWERS_OF_2 + N * 4).

SYNTAX

Label	Operation	Operand
	MACRO	macro-name

PARAMETERS

macro-name The name of the macro being defined.

EXPLANATION

The **MACRO** directive marks the beginning of a macro definition block. The macro consists of all statements between, but not including, the **MACRO** directive and the next **ENDM** directive.

The Macros section of this manual describes macros in detail.

EXAMPLES

The following macro converts the value of a variable to its two's complement:

Label	Operation	Operand
	MACRO	NEGATE
"1"	SET	\ "1" + 1
	ENDM	

The macro invocation

NEGATE VALUE1

yields the following macro expansion:

VALUE1 SET \ VALUE1 + 1

Every occurrence of the substitution construct ("1") is replaced by the first parameter (**VALUE1**). The two's complement of the specified value is then formed.

NAME

Declares object module name

SYNTAX

Label	Operation	Operand
	NAME	module-name

PARAMETERS

module-name A name for the object module being created: any symbol.

EXPLANATION

The NAME directive gives a name to the object module created by this assembly. If more than one NAME directive appears in a module, only the first name specified is used. If the source module contains no NAME directive, the default name *NONAME* is assigned to the object module.

The library generator (LibGen) requires that each module in a library file have a unique name.

EXAMPLES

Label	Operation	Operand
	NAME	SUBSMOD

This statement assigns the name SUBSMOD to the object module being created.

SYNTAX

Label	Operation	Operand
	NOLIST	[listing-option[,listing-option]...]

PARAMETERS

listing-option One of the following listing options:

CND—Suppresses listing of statements that are not assembled because of unsatisfied IF or REPEAT conditions.

CON—Suppresses display of assembly errors on the system terminal.

DBG—Suppresses the internal symbols list for this module.

ME—Suppresses display of all macro expansion statements.

MEG—Suppresses display of all macro expansion statements.

SYM—Suppresses listing of the symbol table.

XREF—Suppresses cross-reference listing.

EXPLANATION

The NOLIST directive turns off the listing option(s) named in the operand field. These options are explained in detail under the LIST directive. If the NOLIST directive is used without parameters, all output is suppressed except error messages, symbol table, and cross-reference listing. The NOLIST directive does not affect page size or line size.

SYNTAX		
Label	Operation	Operand
[symbol]	ORG	{ address /address-mod }

PARAMETERS

symbol	A user-defined label (usually omitted) that is assigned the value of the updated location counter.
address	A new value for the location counter: any expression that yields an address or a scalar. If an address is used it must be in the current section. Each symbol in the expression must have been defined previously.
address-mod	Any numeric expression. The location counter is advanced to the next address that is a multiple of the address-mod. Each symbol in the expression must have been defined previously.

EXPLANATION

The **ORG** directive sets the location counter to the specified address.

If the / (slash) operator is used, the location counter is set to the next address that is a multiple of the **address-mod**. If the current value of the location counter is already a multiple of the **address-mod**, the location counter is unaffected. If the **address-mod** is zero and the value in the location counter is even, the location counter is set to the next odd value.

The location counter is an internal counter, maintained by the assembler, that holds the address, relative to the beginning of the current section, of the next byte of code to be assembled. The location counter starts at zero for each section and is automatically updated as object code is generated.

The **ORG** directive is generally used to initialize the program counter for an absolute section, or to begin the next block of object code on a new page of memory.

NOTE

*Avoid using **ORG** in a byte-relocatable or inpage-relocatable section: the conditions that you use **ORG** to create are likely to be lost when the section is relocated.*

If, through use of the **ORG** directive, you break your section into noncontiguous blocks of code, the linker may place other sections in the gaps between these blocks. (See Example 1.) Every byte in a section retains its position relative to the beginning of the section even if the section is relocated.

NOTE

If you use ORG incorrectly, you may end up specifying more than one value for the same byte of object code. (See Example 2.) Such a situation is not detected by the assembler, linker, or loader.

EXAMPLES**ORG Example 1**

Label	Operation	Operand	Comment
	; DEFINE SECTION ABS (AN ABSOLUTE SECTION).		
	SECTION	ABS,ABSOLUTE	
	ORG	100H	; START ON PAGE 1
ABS1	BLOCK	80H	; 128 BYTES OF MEMORY
	ORG	/100H	; GO TO BEGINNING OF NEXT PAGE.
ABS2	BLOCK	40H	; 64 BYTES
	ORG	400H	; GO TO PAGE 4.
ABS3	BLOCK	80H	; 128 BYTES
	; DEFINE SECTION REL (A BYTE-RELOCATABLE SECTION).		
	SECTION	REL	
REL1	BLOCK	40H	; 64 BYTES
	ORG	/100H	; GO TO BEGINNING OF NEXT PAGE
REL2	BLOCK	80H	; 128 BYTES

In this example, two sections of object code are generated. Section ABS is divided into three blocks and section REL is divided into two blocks. Figure 3-5 shows the layout of the two sections.

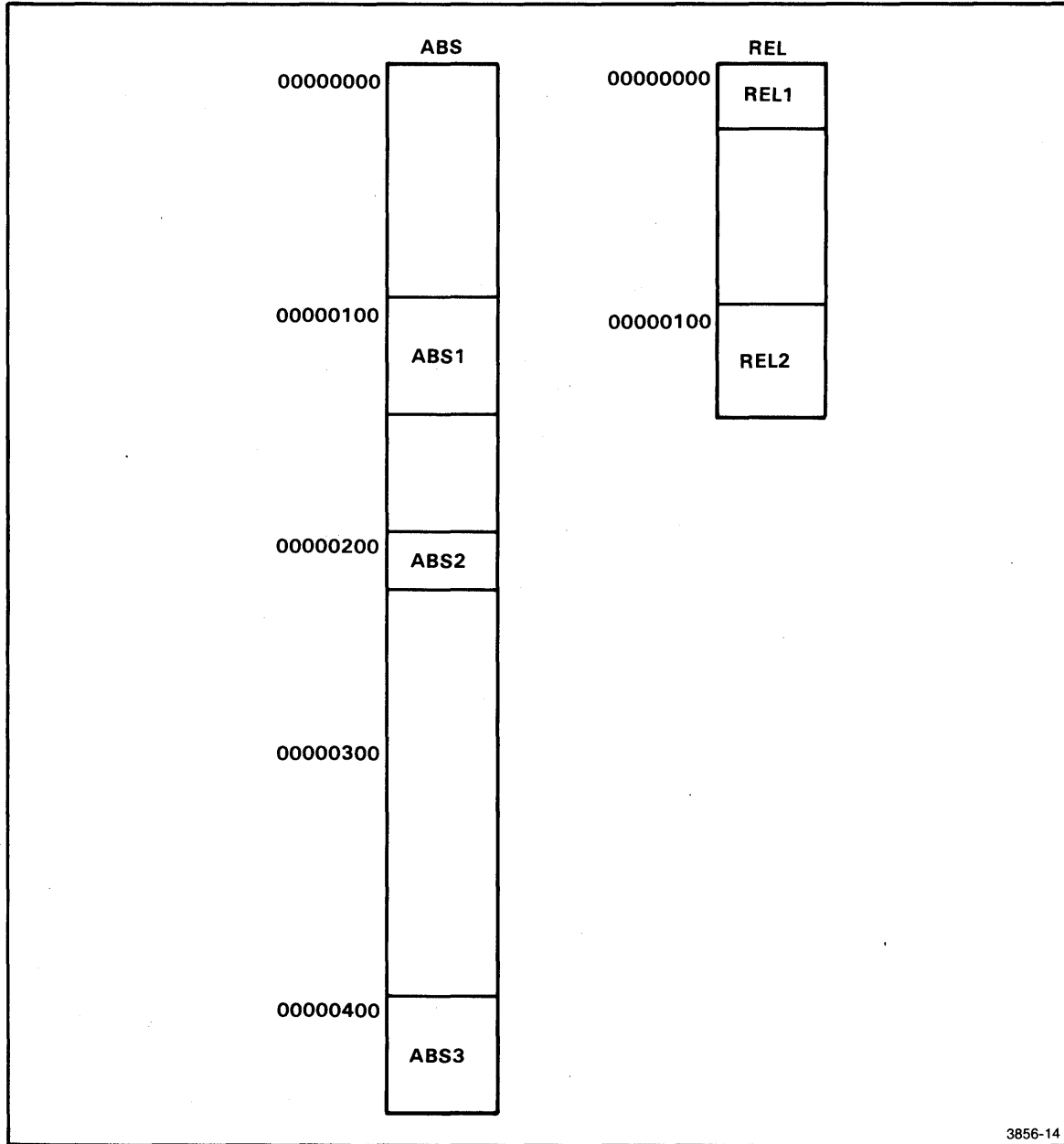


Fig. 3-5. Sections before linking.

3856-14

Figure 3-6 shows how the linker will arrange the two sections:

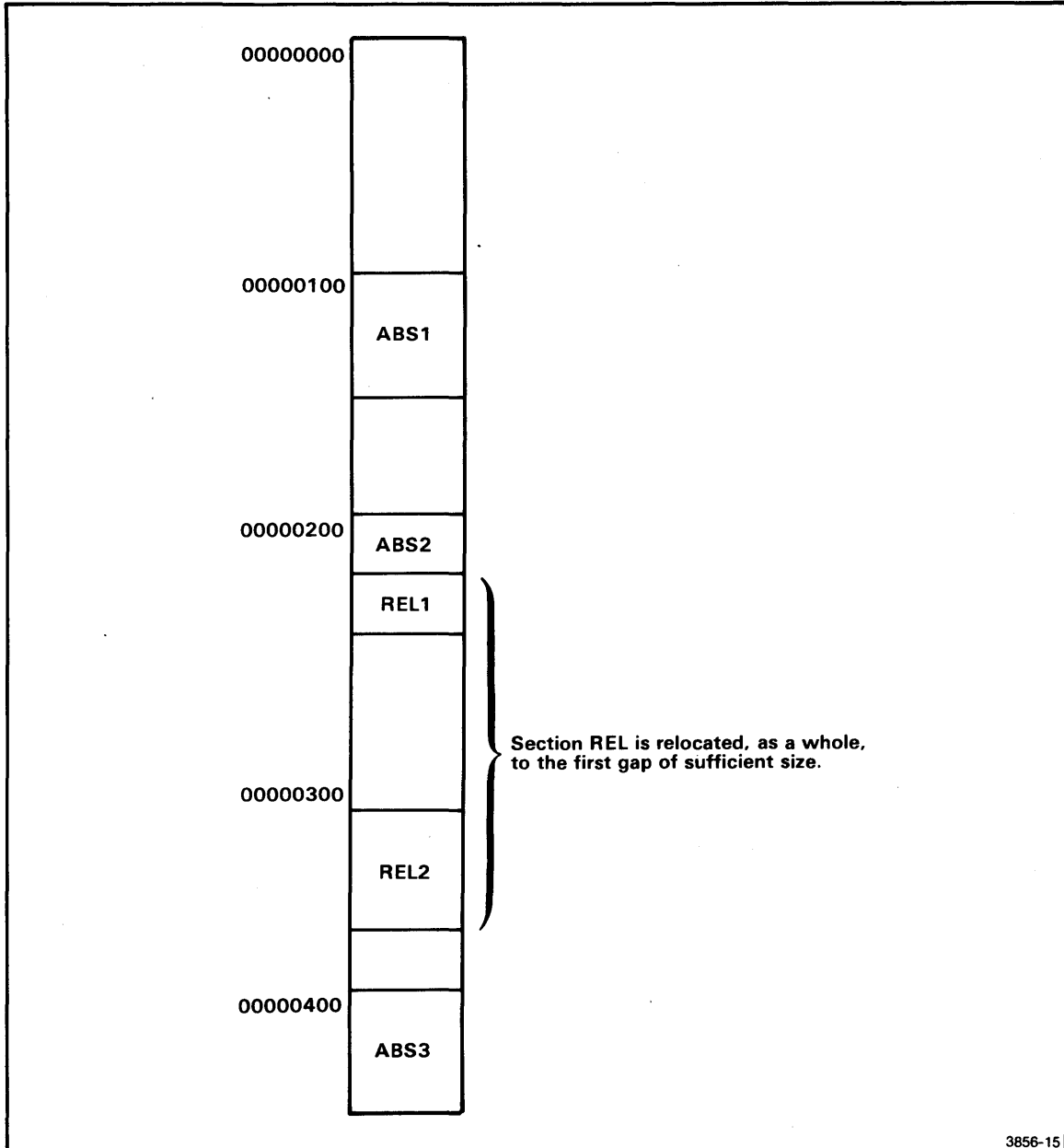


Fig. 3-6. Sections after linking.

Notice that section REL is placed between blocks ABS2 and ABS3 of section ABS. Notice also that block REL2 began on a page boundary before it was relocated, but not after.

ORG

Sets location counter

ORG Example 2

Label	Operation	Operand
	ORG	400H
	ASCII	'A LINE OF TEXT'
	ORG	405H
	ASCII	'*****'

yields the same object code as:

	ORG	400H
	ASCII	'A LIN*****TEXT'

The five asterisks overwrite the information stored previously.

SYNTAX		
Label	Operation	Operand
PAGE		

EXPLANATION

A PAGE directive causes the next source line listed to appear at the top of a new page. The PAGE directive itself is not listed.

If the source listing is suppressed by a NOLIST directive, the PAGE directive has no effect.

EXAMPLES

Label	Operation	Operand	Comment
	TITLE	'THIS IS THE TITLE'	
	.		
	PAGE		; SKIP TO A NEW PAGE TO
	SECTION	MAIN	; BEGIN CODE FOR MAIN.

These statements cause the source code for section MAIN to begin on a new page. The top of the new page looks like this:

Tektronix ASM xxxxxxxx	THIS IS THE TITLE	Page	x
Vxx.xx-xx (xxxx)		xxxxxxxxxxxxxxxxxxxx	
xxxxx	SECTION	MAIN	; BEGIN CODE FOR MAIN.

REPEAT

Begins repetitive assembly

Assembler Directive—8500 Series B Assembler Core Users

SYNTAX		
Label	Operation	Operand
	REPEAT	{ n TIMES condition-value[,limit] }

PARAMETERS

n	The number of times to repeat the block: any positive scalar.
condition-value	Any expression that yields a scalar value. The condition is considered false if the value is zero or undefined and true if the value is nonzero.
limit	The maximum number of times the block may repeat: any expression that yields a scalar in the range from 0 to 65535. Defaults to 255.

EXPLANATION

The REPEAT directive is used to control the number of times a block of assembler statements is assembled. The REPEAT block is terminated by the ENDR directive. The statements between the REPEAT and ENDR directives are assembled, repetitively, until one of the following conditions is met:

1. The condition-value is evaluated to false (zero or undefined). The evaluation occurs before each pass. If the condition-value is false before the first repetition, the REPEAT...ENDR block is not assembled at all.
2. The block has been assembled **limit** times. If this happens, an error message is generated and assembly resumes starting with the statement following the ENDR.
3. The **TIMES** option is used and the block has been assembled **n** times.
4. EXITR is assembled. This is usually used in conjunction with the IF directive.

The condition-value may be a relational expression (for example, $J < 0$). See the IF directive for a note on the relationship between numeric and relational expressions.

A REPEAT block may be nested inside an IF block or another REPEAT block. Blocks may be nested as deep as available memory in the assembler permits. A REPEAT block may not lie partially inside **and** partially outside an IF block, another REPEAT block, a macro expansion, or statements from an INCLUDE file. Figure 3-7 shows the allowed forms of nesting for REPEAT blocks.

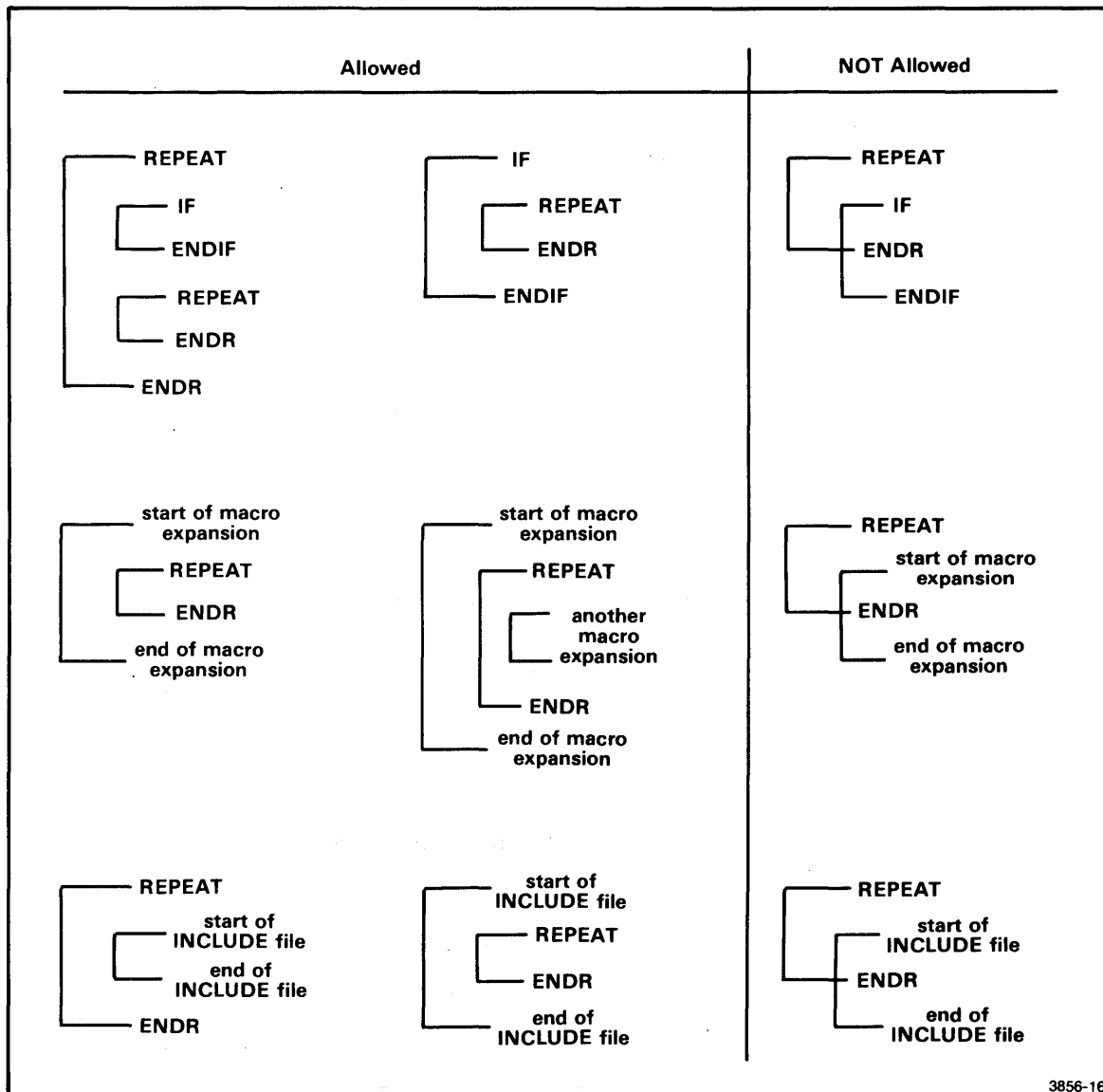


Fig. 3-7. Allowed forms of REPEAT block nesting.

A REPEAT block may not lie partially **inside** and partially **outside** an IF block, another REPEAT block, a macro expansion, or statements from an INCLUDE file.

REPEAT

Begins repetitive assembly

Assembler Directive—8500 Series B Assembler Core Users

EXAMPLES

REPEAT Example 1:

Label	Operation	Operand
	MACRO	LOOP1
COUNT	SET	1
	REPEAT	COUNT <= "1"
	IF	"1" > 1000
	WARNING	;ERROR : EXCESSIVE ITERATION
	EXITR	
	ENDIF	
	BYTE	"2"
COUNT	SET	COUNT + 1
	ENDR	
	ENDM	

The statement

```
LOOP1 3,0
```

invokes the above macro and produces the following expansion (the IF block is not shown since its condition is false here):

COUNT	SET	1	
	REPEAT	COUNT <= 3	
	BYTE	0	
COUNT	SET	COUNT + 1	(COUNT is incremented to 2.)
	ENDR		
	REPEAT	COUNT <= 3	
	BYTE	0	
COUNT	SET	COUNT + 1	(COUNT is incremented to 3.)
	ENDR		
	REPEAT	COUNT <= 3	
	BYTE	0	
COUNT	SET	COUNT + 1	(COUNT is incremented to 4.)
	ENDR		

This sequence generates three bytes of zeros. Note that with the listing options at their default settings, only the BYTE directives would appear in the listing:

```
BYTE 0
BYTE 0
BYTE 0
```

See the LIST directive for more information on listing options.

If the macro were invoked with the statement:

```
LOOP1 1001,0
```

The user-defined warning message would be generated and the REPEAT block would not be assembled at all.

REPEAT Example 2:

Label	Operation	Operand
COUNT	SET	1
	REPEAT	100 TIMES
	WORD	COUNT
COUNT	SET	COUNT + 1
	ENDR	

This sequence would produce 100 words of data. The values would range from 1 to 100, in consecutive order.

RESERVE

Reserves section of memory

Assembler Directive—8500 Series B Assembler Core Users

SYNTAX

Label	Operation	Operand
[symbol]	RESERVE	sect-name,length[,relocation-type][, CLASS =class-name]

PARAMETERS

- symbol** A user-defined label (usually omitted) that represents the first byte of the relocated reserve section.
- sect-name** The name assigned to the section.
- length** The number of bytes in the section: any non-negative scalar expression.
- relocation-type** An option to direct the relocation of the section at link time. You may specify one of the following relocation types:
- PAGE**—The section is relocated to the beginning of a page of memory. A frequently used page size is 256 bytes. See the Assembler Specifics section of this manual for the page size for your microprocessor.
- INPAGE**—The section may be relocated to any address, so long as the entire section lies within one page of memory.
- If you do not specify **PAGE** or **INPAGE**, the section is attributed with the default relocation type. This default relocation type is microprocessor-dependent. See the Assembler specifics section for information about the default relocation type for your microprocessor.
- class-name** The class name assigned to the section.

EXPLANATION

The **RESERVE** directive creates a section with the specified name, length, relocation type, and class name. Different modules may allocate space for the same reserve section; the linker concatenates all reserve sections with the same name into a single section.

Since you can specify the length, but not the contents, of a reserve section, **RESERVE** is used primarily to set aside memory for a workspace or stack.

A reserve section may not have the relocation type **ABSOLUTE** or **ALIGN**; however, you may use the linker command **LOCATE** to place the section at the desired position in memory. See the Linker section of this manual.

The **RESERVE** directive has no effect on the section currently being defined.

The relocation type of a reserve section must be the same wherever the section is declared. A section must not be declared more than once in the same module.

The name of a section is a global symbol whose value is the address of the first byte of the section. A section name should not be declared with a GLOBAL directive in any module in which the section is defined with a RESERVE directive.

The class name is not used by the assembler. It is passed to the linker so that the linker commands may refer to this section by its class name rather than its section name. Refer to the Linker section of this manual for further information on class names.

EXAMPLES

Label	Operation	Operand	Comment
	NAME	MOD1	
	SECTION	SEC1	; BEGIN DEFINITION OF SEC1.
	.		
	RESERVE	STACK,40	; SET ASIDE 40 BYTES FOR STACK.
	.		; RESUME DEFINITION OF SEC1.
	.		

In the above example, 40 bytes are allocated to a **byte-relocatable** reserve section called STACK. The statements on either side of the RESERVE directive refer to section SEC1.

	NAME	MOD2	
	.		
	RESERVE	STACK,20	; SET ASIDE 20 BYTES FOR STACK.

When modules MOD1 and MOD2 are linked, reserve section STACK will occupy 60 bytes of memory: 40 bytes from MOD1 and 20 bytes from MOD2.

RESUME

Resumes definition of section

Assembler Directive—8500 Series B Assembler Core Users

SYNTAX

Label	Operation	Operand
[symbol]	RESUME	[section-name]

PARAMETERS

- symbol** A user-defined label (usually omitted) that is assigned the current value of the location counter of the resumed section.
- section-name** The name of the section to be resumed. If no name is given, the default section is resumed.

EXPLANATION

The RESUME directive stops definition of the current section and resumes the definition of the specified section. It is illegal to RESUME a RESERVE section.

If no section name is given, the definition of the default section is continued. The default section is described under the SECTION directive.

Once a section is defined, it may be resumed any number of times.

EXAMPLES

Label	Operation	Operand	Comment
	SECTION	MAINPROG	; BEGIN DEFINITION OF MAINPROG.
	MOV	TEMP,AX	; USE A TEMPORARY LOCATION.
	SECTION	RAM	; SWITCH TO RAM ...
TEMP	BLOCK	1	; ... TO ALLOT SPACE FOR TEMP.
	RESUME	MAINPROG	; GO BACK TO ORIGINAL SECTION.

In this example, the definition of section MAINPROG is interrupted to reserve one byte for temporary storage. The RESUME directive continues the definition of section MAINPROG.

SYNTAX

Label	Operation	Operand
[symbol]	SECTION	section-name[,relocation-type][,CLASS=class-name]

PARAMETERS

symbol A user-defined label (usually omitted) that represents the address of the first byte of the section.

section-name The name assigned to the section.

relocation-type An option to direct the relocation of the section at link time. You may specify one of the following relocation types:

PAGE—The section is relocated to the beginning of a page of memory. A frequently used page size is 256 bytes. See the Assembler Specifics section of this manual for the page size for your microprocessor.

ALIGN(address-mod)—The section may be relocated to any address that is a multiple of the address-mod. Address-mod represents a positive scalar expression. Each symbol in the expression must have been defined previously.

INPAGE—The section may be relocated to any address, so long as the entire section lies within one page of memory.

ABSOLUTE—The section is not relocated. You may not assign a class name to an absolute section.

If you do not specify PAGE, ALIGN, INPAGE, or ABSOLUTE, the section is attributed with the default relocation type. This default relocation type is microprocessor-dependent. See the Assembler specifics section for information about the default relocation type for your microprocessor.

class-name The class name assigned to the section. You may not assign a class name to an absolute section.

EXPLANATION

The SECTION directive declares a section of type SECTION and defines the name, relocation type, and class name of the section. The contents of the section are defined by the statements following the SECTION directive, up to the next SECTION, COMMON, or RESUME directive.

The class name is not used by the assembler. It is passed to the linker so that the linker commands may refer to the section by its class name rather than its section name. Refer to the Linker section of this manual for further information on class names.

SECTION

Declares program section

Assembler Directive—8500 Series B Assembler Core Users

Any section that contains instructions (as opposed to data) should be of type SECTION.

NOTE

In this discussion, the word 'SECTION' (all uppercase) refers to a section declared with a SECTION directive, rather than with a COMMON or RESERVE directive.

Unlike a common or reserve section, a SECTION must be defined entirely in one module. Use the RESUME directive to add code to a section that has already been defined in the current module. If the linker encounters more than one SECTION with the same name, the linker issues an error message and links only the first SECTION with that name.

The name of a section is a global symbol whose value is the address of the first byte of the section. A section name should not be declared with a GLOBAL directive in the same module in which the section is defined with a SECTION directive.

The **default section** of a module contains all object code generated before the first SECTION or COMMON directive is assembled. The name of the default section is derived as follows:

1. Eliminate all characters except letters and digits from the name of the object file.
2. Take the first 15 characters.
3. Add the prefix '%'

For example, the default section for object file /USR/FRED/OBJ is %USRFREDOBJ. When no object file is generated, the default section is called %.

EXAMPLES

```
Label      Operation  Operand
          SECTION   MAINPROG
(source code for section MAINPROG)
.
          SECTION   TABLE,INPAGE
(source code for section TABLE)
.
          SECTION   INTERRUP,ABSOLUTE
          ORG       100H
(source code for section INTERRUP)
```

In this example, section MAINPROG may be relocated by the linker to any address. TABLE is relocatable to any address, so long as the entire section lies within one page of memory. INTERRUP, which is not relocatable, begins at address 100H.

SYNTAX		
Label	Operation	Operand
string-variable	SET	string-expression
numeric-variable	SET	numeric-expression

PARAMETERS

string-variable	A user-defined label for a string variable.
numeric-variable	A user-defined label for a numeric variable.
string-expression	Any expression that yields a character string.
numeric-expression	Any expression that yields a numeric value.

EXPLANATION

The SET directive assigns a value to a symbol. The symbol is called a **variable** because it may be assigned a new value with a subsequent SET directive. A variable may be used anywhere the value it represents is permitted.

A variable must not be a global symbol. SET may not redefine a symbol unless that symbol was originally defined with a SET directive.

There are two types of variables: string and numeric.

- A **string** variable represents a character string. A string variable must be declared with a **STRING** directive before it may be assigned a value.
- A **numeric** variable represents a scalar or address. A numeric variable need not be declared; it becomes defined the first time a SET directive assigns it a value.

If the type of the variable does not match the type of the value assigned to it, the value is converted to match the type of the variable.

- If you assign a string value to a numeric variable, the variable takes the ASCII value of the first four bytes of the string (32 bits). If the string exceeds four characters, the assembler issues an truncation error.

If the string contains less than four characters, zeros are padded into the remaining high-order bytes.

- If you assign a numeric value to a string variable, the numeric value is treated as a literal string constant.

See The Assembler section of this manual for further information on conversions.

SET

Assigns value to variable

Assembler Directive—8500 Series B Assembler Core Users

Text substitution (signaled by double quotes " ") often involves variables. A string variable in double quotes (e.g., "STVAR") is replaced by the string the variable represents. The substituted string is **not** enclosed in quotes. A numeric variable in quotes (e.g., "N") is legal only in macros, and is replaced by the Nth parameter in the macro invocation. See The Assembler section of this manual for information on text substitution.

EXAMPLES

Label	Operation	Operand	Comment
	MACRO	BYTES	
N	SET	1	; SET POINTER TO FIRST PARAMETER.
	REPEAT	N <= "#"	; REPEAT FOR EACH PARAMETER:
	BYTE	"N",-"N"	; ALLOCATE TWO BYTES FOR THE NTH PARAM.
N	SET	N+1	; INCREMENT PARAMETER POINTER.
	ENDR		
	ENDM		

In this example, N is a numeric variable that counts from 1 to the number of parameters in the macro invocation ("#"). The construct "N" is replaced by the Nth parameter. The invocation

```
BYTES 10,20,MAX
```

yields the macro expansion

```
BYTE 10,-10 ; ALLOCATE TWO BYTES FOR THE NTH PARAM.  
BYTE 20,-20 ; ALLOCATE TWO BYTES FOR THE NTH PARAM.  
BYTE MAX,-MAX ; ALLOCATE TWO BYTES FOR THE NTH PARAM.
```

In the following example, string variables VOL and FILE are assigned values and then concatenated to form the filespec of an INCLUDE file.

Label	Operation	Operand
	STRING	VOL(8),FILE(8)
	.	
VOL	SET	'/SYS'
	.	
	.	
FILE	SET	'INC.ASM'
	.	
	.	
	INCLUDE	VOL:'/:FILE

The statements from file INC.ASM in the system directory (/SYS/INC.ASM) are assembled following the INCLUDE directive.

In the following example, the name of the current section ("%") is stored in string variable SECNAME and is later substituted into the RESUME directive.

Label	Operation	Operand
	STRING	SECNAME(8)
	SECTION	MAINPROG
	.	.
SECNAME	SET	'%"'
	.	.
	RESUME	"SECNAME"

The above lines are assembled as follows:

	STRING	SECNAME(8)
	SECTION	MAINPROG
	.	.
SECNAME	SET	'MAINPROG'
	.	.
	RESUME	MAINPROG

SYNTAX

Label	Operation	Operand
	STITLE	subtitle-string

PARAMETERS

subtitle-string The subtitle for the source listing: any expression that yields a string of up to 35 characters.

EXPLANATION

The STITLE directive creates a subtitle of up to 35 characters. The subtitle is printed below the title line at the top of each page of the source listing. The STITLE directive itself is not listed.

Each subsequent STITLE directive redefines the subtitle. If the STITLE directive precedes the first source line listed on the current page, the new subtitle appears on the current page; otherwise it first appears on the next page. Thus, if a STITLE directive immediately precedes or follows a PAGE directive, the designated subtitle appears at the top of the new page.

If the subtitle string exceeds 35 characters, only the first 35 characters are used.

The STITLE directive is used for program documentation only. You may choose to change the subtitle to reflect each new section of code.

EXAMPLES

Label	Operation	Operand	Comment
	TITLE	'THIS IS THE TITLE'	
	STITLE	'SUBTITLE FOR PAGES 1 AND 2'	
	; THIS IS THE FIRST LISTABLE LINE.		
	.		
	PAGE		; SKIP TO PAGE 2.
	.		
	PAGE		; SKIP TO PAGE 3.
	STITLE	'SUBTITLE FOR PAGE 3'	

STITLE

Creates listing subtitle

Assembler Directive—8500 Series B Assembler Core Users

The preceding statements produce the following page headings in the source listing:

```
Tektronix ASM xxxxxxxxxx      THIS IS THE TITLE      Page 1
Vxx.xx-xx (xxxx) SUBTITLE FOR PAGES 1 AND 2      xxxxxxxxxxxxxxxxxxxxxx
```

```
3 ; THIS IS THE FIRST LISTABLE LINE.
```

```
Tektronix ASM xxxxxxxxxx      THIS IS THE TITLE      Page 2
Vxx.xx-xx (xxxx) SUBTITLE FOR PAGES 1 AND 2      xxxxxxxxxxxxxxxxxxxxxx
```

```
Tektronix ASM xxxxxxxxxx      THIS IS THE TITLE      Page 3
Vxx.xx-xx (xxxx) SUBTITLE FOR PAGE 3      xxxxxxxxxxxxxxxxxxxxxx
```

SYNTAX

Label	Operation	Operand
	STRING	string-variable[(length)],string-variable[(length)]...

PARAMETERS

string-variable	A symbol to be used as a string variable.
length	The length of the longest string that may be assigned to string-variable: any expression that yields a positive scalar value. Defaults to 16.

EXPLANATION

The **STRING** directive declares each symbol in the operand field to be a string variable. Each symbol may be followed by a non-negative value indicating the length of the longest string that may be assigned to that variable. If no **length** parameter is specified, the maximum length defaults to 16 characters.

A symbol must be declared with a **STRING** directive before it can be used as a string variable. When a string variable is declared, its value is initialized to the null string (zero characters). Use the **SET** directive to assign a value to a variable.

EXAMPLES

Label	Operation	Operand
	STRING	CITY(10),STATE,HOMETOWN(26)
	.	
CITY	SET	'BEAVERTON'
	.	
STATE	SET	'OREGON'
	.	
HOMETOWN	SET	CITY:', ':STATE

In this example, the **STRING** directive declares **CITY**, **STATE**, and **HOMETOWN** as string variables with maximum lengths of 10, 16, and 26, respectively. Subsequently, **CITY** is assigned a 9-character string ('BEAVERTON'), **STATE** is assigned a 6-character string ('OREGON'), and **HOMETOWN** is assigned a 17-character string ('BEAVERTON, OREGON').

TITLE

Creates listing title

SYNTAX

Label	Operation	Operand
	TITLE	title-string

PARAMETERS

title-string The title for the source listing; any expression that yields a string of up to 31 characters.

EXPLANATION

The TITLE directive creates a title of up to 31 characters to be printed at the top of each page of the source listing. The TITLE directive itself is not listed.

Each subsequent TITLE directive redefines the title. If the TITLE directive precedes the first source line listed on the current page, the new title appears on the current page; otherwise, it first appears on the next page. Thus, if the TITLE directive immediately precedes or follows a PAGE directive, the new title appears at the top of the new page.

If the title string exceeds 31 characters, only the first 31 characters are used.

The TITLE directive is used for program documentation only. You may choose to use the same title throughout the module, or you may change the title or subtitle as often as you want.

EXAMPLES

Label	Operation	Operand	Comment
	TITLE	'THE SAME OLD TITLE'	
	STITLE	'THE SAME OLD SUBTITLE'	
	.		
	PAGE		; SKIP TO PAGE 2.
	.		
	PAGE		; SKIP TO PAGE 3.
	TITLE	'A NEW TITLE'	

The preceding statements produce the following page headings in the source listing:

```
Tektronix ASM xxxxxxxxx THE SAME OLD TITLE Page 1  
Vxx.xx-xx (xxxx) THE SAME OLD SUBTITLE xxxxxxxxxxxxxxxxxxxx
```

```
Tektronix ASM xxxxxxxxx THE SAME OLD TITLE Page 2  
Vxx.xx-xx (xxxx) THE SAME OLD SUBTITLE xxxxxxxxxxxxxxxxxxxx
```

```
Tektronix ASM xxxxxxxxx A NEW TITLE Page 3  
Vxx.xx-xx (xxxx) THE SAME OLD SUBTITLE xxxxxxxxxxxxxxxxxxxx
```

WARNING

Displays warning

SYNTAX

Label	Operation	Comment
	WARNING	[;message]

PARAMETERS

message Any user-defined error message.

EXPLANATION

When a **WARNING** directive is assembled, it is treated as an erroneous statement: an error message containing the text in the comment field is displayed on the system terminal and in the source listing. The semi-colon (;) signifies the beginning of the comment field.

You may use the **WARNING** directive to detect unexpected conditions in your program.

EXAMPLES

```
Label      Operation  Operand
          SECTION   ONE
          .
          .
          .
          IF          M>N & N<P & P=Q
          WARNING ; TROUBLE IN SECTION "%"
          ENDIF
```

In this example, if $M > N$ and $N < P$ and $P = Q$, the **WARNING** is assembled and the following message is displayed:

```
xxxxx          WARNING ; TROUBLE IN SECTION ONE
*** ASM:  1(W)
```

The construct "%" is replaced by the name of the current section.

The following is the Global Symbol Listing with Cross-Reference resulting from the example linker command.

```
Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 5
For LOADFILE
```

GLOBAL SYMBOL LISTING:

```
CHARIN          0  LAS8550A
CHAROUT         1000 LAS8550B
CHARBUFFIN      FO  LAS8550A
CHARBUFFOUT     1014 LAS8550B
DEFINE          2F0  DATA.A1
ENDREL          1141 ***** [defined by the linker]
                  LAS8550A  LAS8550B
GLOBAL1         2F0  DATA.A1
PORT1           10  LAS8550A
PORT2           1010 LAS8550B
                  DATA.A1
PORT3           2F2  DATA.A1
                  LAS8550A  LAS8550B
TESTZ           250  LAS8550B
```

Notice that some global symbols are referenced by more than one object module. For instance, global symbol PORT3 is referenced by the two object modules: LAS8550A and LAS8550B.

Statistics

The Statistics listing gives the number of errors detected and reports whether an absolute load file was generated. If one was generated, it also tells whether the load file can be relinked by the Linker or debugged by the Symbolic Debugger.

The following is the Statistics listing resulting from the example linker command.

```
Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 6
For LOADFILE
```

STATISTICS:

```
Number of warning errors: 0
Number of errors: 0
Transfer Address: 1000
```

```
Load file is not relinkable
Load file is not usable for symbolic debugging
```


Section 6 THE LIBRARY GENERATOR

	Page
Introduction	6-1
LibGen Invocation	6-1
Command Option Parameters	6-3
Filespecs	6-3
Module Names	6-3
Command Files	6-3
Command Options	6-4
—c	6-4
—d	6-4
—h	6-4
—i	6-4
—l	6-4
—n	6-5
—o	6-5
—r	6-5
—v	6-5
—x	6-5
Examples	6-6
Delete Library Modules	6-6
Create a New Library File	6-6
Specify the Header of New Library File	6-6
Add a Library Module	6-6
Display the Module List on System Terminal	6-7
Output the Module List to a File	6-7
Display the Entire LibGen Listing on the System Terminal	6-7
Replace Old Modules with New Modules	6-7
Copy Module to Object File	6-8
Modify and List a Library File	6-8
Sample LibGen Command File	6-8
LibGen Execution	6-9

	Page
LibGen Output	6-10
The New Library File	6-10
The Listing	6-10
Command Log	6-10
Module List	6-11
Summary of Actions	6-11

TABLES

Table No.		Page
6-1	LibGen Command Options	6-2

ILLUSTRATIONS

Fig. No.		Page
6-1	—d command option example	6-6
6-2	—i command option example	6-7
6-3	—r command option example	6-8
6-4	LibGen information flow	6-9

Section 6

THE LIBRARY GENERATOR

INTRODUCTION

The Library Generator (LibGen) is a general-purpose utility program used to create and maintain object module libraries for use with the linker.

LibGen collects assembler-generated or compiler-generated object modules into library files. From these library files, the object modules can be individually accessed by the linker, based on the information provided in each object module.

This section describes the operations and use of LibGen, and is divided into the following subsections:

- **LibGen Invocation.** Describes how to invoke LibGen. Presents a detailed description of each command option used to control the operation of LibGen.
- **LibGen Execution.** Describes operations performed by LibGen.
- **LibGen Output.** Describes the listing file generated by LibGen.

Some typical uses of LibGen are presented in the Programming Examples section of this manual.

LIBGEN INVOCATION

With the 8550 Microcomputer Development Lab or the 8560 Multi-User Software Development Unit, the LibGen is invoked by the operating system command **libgen**.

NOTE

If you are using any system other than the 8550 or 8560, the LibGen invocation command may be different. Refer to the Host Specifics section of this manual for further information.

Throughout this section, the same notation conventions are used as described in the Learning Guide of this manual. Additionally, if you are using the 8550 Microcomputer Development Lab, the command name **libgen** may be entered in either upper or lower case.

SYNTAX

`libgen command-option ...`

PARAMETERS

`command-option` One of the command-options listed in Table 6-1.

**Table 6-1
LibGen Command Options**

Option Syntax	Function
<code>-c command-file</code>	invokes a LibGen command file
<code>-d module-name . . .</code>	deletes library module(s)
<code>-h string</code>	specifies a header for the new library
<code>-i filespec . . .</code>	inserts new module(s) into library
<code>-l</code>	specifies listing option
<code>-n filespec</code>	designates new library file
<code>-o filespec</code>	specifies old library file
<code>-r filespec . . .</code>	replaces old module(s) with new module(s)
<code>-v</code>	specifies detailed listing
<code>-x module-name [filespec]</code>	extracts (copies) module to object file

`command-file` The filespec of the command file containing a series of LibGen command options.

`module-name` The name of the library module.

`string` An ASCII string that identifies the library. The string may contain any printable characters and may not start with a dash (-). The length of the string is limited to 76 characters.

`filespec` The name of the file. Refer to the Host Specifics section of this manual for restrictions on filespecs and filenames of your operating system.

EXPLANATION

The **libgen** command is used to create new library files, modify existing library files, or list existing library files.

- To create a new library file, include the **-n** command option.
- To modify an existing library file, include both the **-o** and **-n** command options. Any unmodified contents of the old library file are copied to the new library.
- To list an existing library file, include both the **-o** and **-l** command options.

The command options may be entered in any order. Only spaces are valid as delimiters in the **libgen** command line.

Command Option Parameters

Filespecs

Any filespec included in a LibGen command option may contain up to 64 characters. Only the following characters are allowed: printable characters from ! (ASCII 21H) to ~ (ASCII 7EH), except that a dash - (ASCII 2DH) may not be the first character in a filespec and a comma (ASCII 2CH) may not be included at all. Your operating system may place further restrictions on filespecs. Refer to the Host Specifics section of this manual for further information.

If you try to create a file that already exists, the existing file will be renamed under the backup name **#filename**.

Module Names

Each module name may contain up to 16 characters. If you enter a module name longer than 16 characters, the excess characters are discarded. A module name begins with a letter or percent sign (%), and may contain only letters, digits, periods (.), underscores (_), dollar signs (\$), or percent signs (%).

Command Files

A LibGen command file contains a series of LibGen command options. It has the following features:

- Only **one** command option per line is allowed.
- Each line is limited to **80** characters long, including the carriage return.
- A command option can be extended to **multiple lines**. To indicate that a command option continues onto the next line, enter a space followed by an asterisk at the end of the current command option line, **before** any comments. The continuation characters (*) can be inserted wherever a delimiter is legal.
- **Comments** can be appended to a command option line by preceding the comment with a space followed by a semicolon. Any text following the comment characters (;) until the end of line will be ignored by LibGen. Blank lines will be ignored. A semicolon in the leftmost column indicates that the entire line is a comment.

Command Options

-c

The **-c** command option invokes a LibGen command file. Command options are read from the command file and processed as if you had entered them from the system terminal, until the end of file is reached. Command options are echoed on the system terminal as they are processed. When the end of the command file is reached, LibGen will continue with the rest of the **libgen** command line.

Nested command files are allowed: a command file may invoke another command file. When the end of the nested command file is reached, LibGen will read the next command option in the upper-level command file. You can have as many levels of nested command files as the memory allows. No checks are made to prevent recursive calls.

-d

The **-d** command option prevents the designated modules from being copied from the old library file into the new library file. The old library file is not affected.

-h

The **-h** command option is used to identify the library. The string parameter is stored as header information in the new library. It is printed on the listing generated by the **-l** and **-v** command options.

You must specify **-n** when you specify **-h**. The **-h** command option is ignored if the **-n** command option is omitted.

If more than one **-h** command option is specified, only the last **-h** command option is used to identify the new library.

-i

The **-i** command option inserts object modules into the new library. Each specified object file contains one object module. If more than one object file is specified, all designated object modules are placed together in the given order. Object modules are always placed at the end of the library.

-l

The **-l** command option displays the module list on the system terminal. Refer to the discussion of LibGen Output later in this section for information on the contents of the module list.

The module list contains information about the new library if a new library (**-n** command option) is specified. Otherwise, information about the old library is listed.

If your operating system permits you to redirect output from the system terminal default, you may output the LibGen listing to a file or device other than the system terminal. See the Examples part of this section.

-n

The **-n** command option designates the output file that is to receive the updated library (new library). If more than one **-n** command option is entered in a **libgen** command line, only the file specified in the last **-n** command option is used as the output library file.

-o

The **-o** command option designates the input file that contains the existing library (old library). If more than one **-o** command option is entered in the **libgen** command line, only the file specified in the last **-o** command option is used as the input library file.

-r

The **-r** command option replaces the library module with the contents of an object file. The old library module is deleted (as if the appropriate **-d** command option were entered), and the object module contained within the object file is inserted at the end of the library.

If the name of the object module (within the specified object file) does not match any of the library module names, an error occurs and the object module is inserted into the library.

-v

The **-v** command option displays the command log and a summary of actions on the system terminal. Refer to the discussion of LibGen Output later in this section for information on the contents of the listings.

You must specify **-l** when you specify **-v**. The **-v** command option is ignored if the **-l** command option is omitted.

-x

The **-x** command option extracts (copies) the designated library object module to a file. If the optional filespec parameter is omitted, LibGen uses the module name as the output filename.

EXAMPLES

Delete Library Modules

```
libgen -o mylib -n newlib -d mymod oldmod
```

This LibGen command prevents modules **mymod** and **oldmod** from being copied from the old library file **mylib** into the new library file **newlib**. See Fig. 6-1.

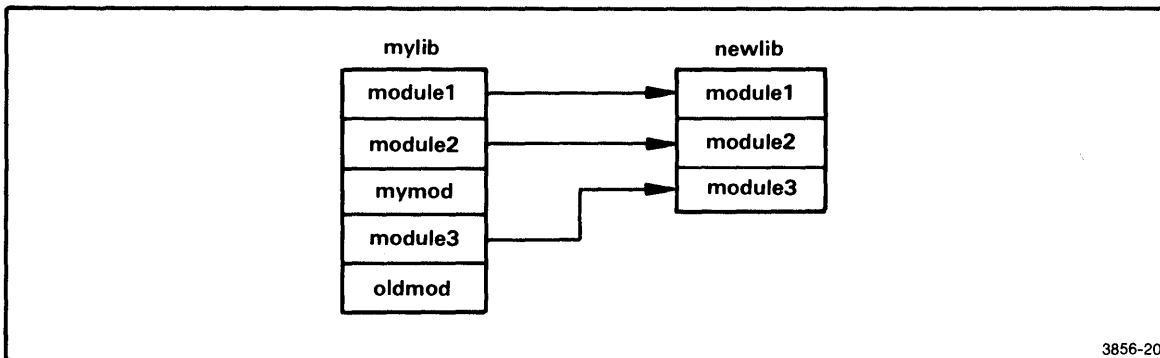


Fig. 6-1. -d command option example.

Create a New Library File

```
libgen -n newlib -i fpadd fpsub fpmult
```

This LibGen command creates a new library **newlib** which contains modules **fpadd**, **fpsub**, and **fpmult**.

Specify the Header of New Library File

```
libgen -n newlib -i fpadd fpsub -h Floating point package V1.1
```

This LibGen command creates a new library **newlib** which contains modules **fpadd** and **fpsub**. This command also identifies **newlib** as a floating point package.

Add a Library Module

```
libgen -o mylib -n newlib -i io.obj
```

This LibGen command copies the old library file **mylib** into a new library file **newlib**, and adds the contents of the file **io.obj** to the end of the library **newlib**. See Fig. 6-2.

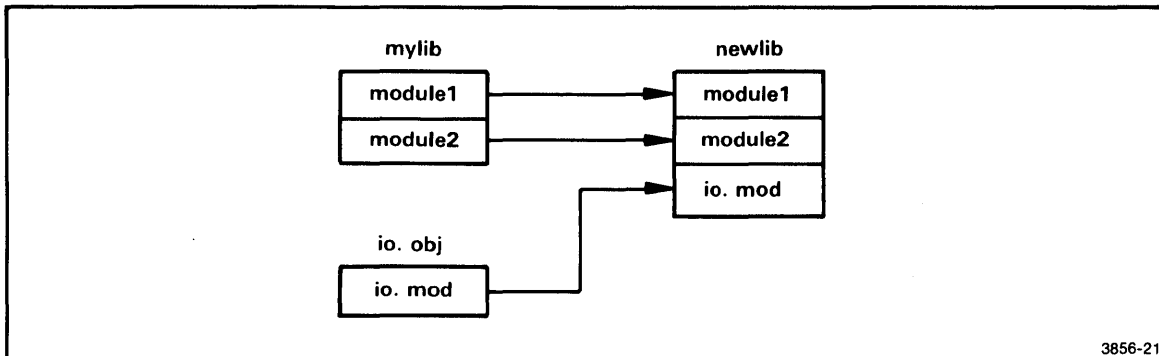


Fig. 6-2. -i command option example.

Display the Module List on System Terminal

```
libgen -o mylib -l
```

This LibGen command displays the module list of the library file **mylib** on the system terminal.

Output the Module List to a File

NOTE

If you are using any system other than the 8550 or 8560, you may not be able to redirect output, or the output redirection format may be different from what is shown here. Refer to the Host Specifics section of this manual for further information.

```
libgen -o mylib -l >mylist
```

This LibGen command designates file **mylist** to receive the module list of the library file **mylib**.

Display the Entire LibGen Listing on the System Terminal

```
libgen -o mylib -l -v
```

This LibGen command displays the entire LibGen listing on the system terminal. The LibGen listing includes the command log, module list, and summary of actions.

Replace Old Modules with New Modules

Assume that the file **fpadd** contains module **modadd**. To replace the module **modadd** contained in the library **mylib** with the one contained in the file **fpadd**, use the **-r** command option.

```
libgen -o mylib -n newlib -r fpadd
```

This LibGen command copies the old library file **mylib** into a new library file **newlib**. During the copying process, LibGen deletes the existing module **modadd**, then inserts the contents of object file **fpadd** in its place. See Fig. 6-3.

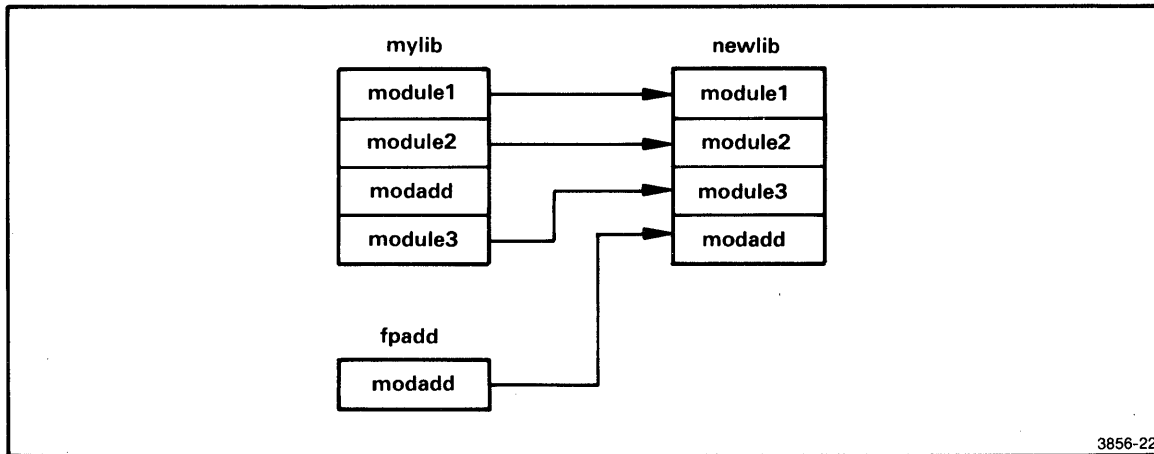


Fig. 6-3. -r command option example.

Copy Module to Object File

```
libgen -o mylib -x io.mod
```

This LibGen command copies the existing library module **io.mod** from the library file **mylib** to an object file with the same name (**io.mod**).

```
libgen -o mylib -x io.mod io.obj
```

This LibGen command copies the existing library module **io.mod** from the library file **mylib** to the object file **io.obj**.

Modify and List a Library File

```
libgen -o mylib -n newlib -i mod1 mod2 -l -d moda modb modc
```

This LibGen command copies the old library file **mylib** into a new library file **newlib**. During the copying process, LibGen inserts modules **mod1** and **mod2**, deletes modules **moda**, **modb**, and **modc**, then displays the module list of **newlib** on the system terminal.

Sample LibGen Command File

A LibGen command file contains a series of LibGen command options. For example, command file **mycommand** contains:

```
; LibGen command file: mycommand
-h Version 3.0
-l
-o mylib
-d mod1 mod2 mod3
-i myfile yourfile hisfile herfile itsfile *
  ourfile theirfile
-n newlib      ;end of mycommand
```

To invoke the command file **mycommand**, enter:

```
libgen -c mycommand
```

The command options in the command file **mycommand** are read and processed, until the end of file is reached.

```
libgen -c mycommand -v
```

This LibGen command invokes command file **mycommand**, also includes the command log and summary of actions in the LibGen listing.

LIBGEN EXECUTION

LibGen performs operations on library files by copying an old library file into a new one. Any changes specified by LibGen commands are made during the copying process. This process is illustrated in Fig. 6-4.

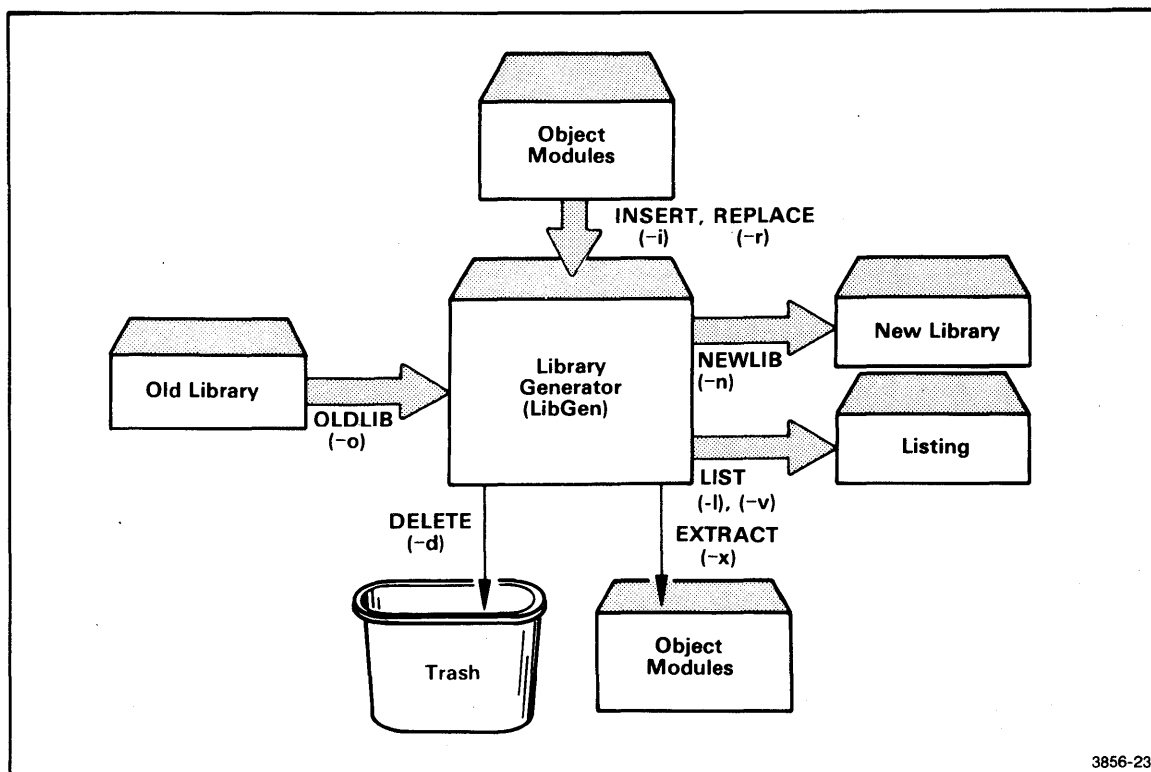


Fig. 6-4. LibGen information flow.

This figure illustrates the information flow into and out of the Library Generator (LibGen). LibGen takes information from the old library and designated object modules, and produces the new library, listing, and object files. The LibGen command options that designate the filespecs used for each file are shown along each data path.

Of course, you won't use all paths shown in Fig. 6-4 each time you invoke LibGen. For example, if you are creating a new library, then you wouldn't need to specify an old library. If you are examining an old library, then you wouldn't need to create a new library. If you do not need a listing, do not specify one.

You may enter the LibGen command options in any order. After you enter the entire **libgen** command line, LibGen processes the command options in the following order:

1. **-x** (extract)
2. **-d** (delete)
3. **-i** (insert)

The **-r** (replace) command option is processed as a combination of the **-d** (delete) and **-i** (insert) command options.

LIBGEN OUTPUT

LibGen produces three different types of output: the new library file, the listing, and the object file(s) (if specified with the **-x** command option).

The New Library File

The new library file contains all the object modules from the old library, plus any object modules that were inserted, minus any object modules that were deleted.

The Listing

The listing summarizes the operations that LibGen has performed. The listing consists of three parts:

1. a command log;
2. a new library module list; and
3. a summary of actions performed by LibGen.

Each of these listing parts is described in detail in the following paragraphs.

Error messages may also be generated by LibGen as a result of mistaken information or requests. LibGen error messages are described in the Error Messages section of this manual.

Command Log

The command log lists each LibGen command option used in the current invocation.

Module List

In this part of the listing, LibGen records the following information for each module in the library:

1. name of the module;
2. global symbols contained within the module;
3. external references (if any) used by the module; and
4. cross-references (the modules that contain the external references in the same library).

Global symbols within each module are divided into four categories:

- **Section name:** The name of a SECTION, COMMON, or RESERVE contained within the module.
- **Data area:** A global symbol which is declared as a data area that can be referenced by other modules.
- **Entry point:** A global symbol that represents a piece of code to be referenced by a subroutine call.

- **Constant:** A scalar value declared global.

These symbols are preceded in the listing with either a (S), (D), (E), or (C), indicating section name, data area, entry point, or constant, respectively.

Note that these global symbols are the factors that determine whether or not a module will be included at link time. For example, assume that module X in the library has a section named "P", an entry point named "P1", and a constant named "P9". At link time, if any one of the symbols "P", "P1", or "P9" has been referenced (through an unbound GLOBAL reference), and if this library had been given as linker input, then module X would be included as if it were one of the normal linker object modules.

Summary of Actions

The summary of actions describes the operations LibGen has performed during this execution.

LibGen actions include:

- generating a new library,
- deleting a module from the library,
- inserting a module into the library, and
- extracting (copying) a library module to an object file.

Section 7 PROGRAMMING EXAMPLES

	Page		Page
Introduction	7-1	Linking Overlays	7-30
Use of Conditional Assembly in Macros	7-2	Using the "@" Construct within a Macro	7-33
Save-and-Restore Macro	7-2	Delay Loop Macro	7-33
The SAVE Macro	7-3	Macro Invocation	7-34
The RESTORE Macro	7-4	The Assembler INCLUDE Directive	7-34
Sample Invocations	7-4	Including Constant Definitions	7-34
SVC Generation	7-5	Including COMMON Declarations	7-35
Creating Service Request Blocks	7-5	Including Macro Definitions	7-35
The SRB Macro	7-6	Authorship and Copyright Notices for Listings	7-36
Explanation of the SRB Macro	7-6		
Sample Invocations of the SRB Macro	7-9		
Generating Service Calls	7-9		
The SVC Macro	7-10		
Explanation of the SVC Macro	7-10		
Sample Invocation of the SVC Macro	7-10		
Creating Constant Values	7-10		
The CONSTANT Macro	7-11		
The VARIABLE Macro	7-13		
Macro Invocation	7-13		
Creating and Using a Subroutine Library	7-14		
The ADD Module	7-15		
Explanation of the ADD Module	7-15		
Entry Points	7-16		
The SUBTRACT Module	7-16		
Explanation of the SUBTRACT Module	7-17		
Entry Points	7-17		
Assembling the Modules	7-17		
Creating the Library	7-20		
Using the ADD Module from a Program	7-20		
The Mainline Add Program	7-21		
Explanation of the Mainline Add Program	7-21		
Assembling and Linking the Program	7-22		
Linking Explanation	7-24		
Using the SUBTRACT Module from a Program	7-25		
The Mainline Subtract Program	7-26		
Explanation of the Mainline Subtract Program	7-26		
Assembling and Linking the Program	7-26		
Linking Explanation	7-29		

ILLUSTRATIONS

Fig. No.		
7-1	Linking the add program to the library	7-25
7-2	Linking the subtract program to the library	7-30

Section 7

PROGRAMMING EXAMPLES

INTRODUCTION

This section contains examples of some typical uses of the assembler, linker, and library generator. These examples range from the basic (conditional assembly) to the more complex (creation and use of a floating-point library).

In order to get the most out of this section, you should have some familiarity with assembly language programming, and with the Tektronix 8500 Modular MDL B Series Assembler, Linker, and Library Generator. These examples are **not** intended to be used during your initial familiarization with these subsystems.

The examples in this section use 8086/8088 instructions, but similar instructions for other microprocessors may be substituted without changing the validity of the examples.

The following examples are included in this section:

- **Conditional assembly.** This example suggests ways of using the IF assembler directive to include or omit program segments, based on various conditions.
- **Save-and-restore macro.** This example uses a macro to perform a common programming operation: saving registers on the stack and restoring the registers from the stack.
- **SVC generation.** This example shows how the macro and conditional assembly features of the assembler can make it easier to generate service calls (SVCs).
- **Creating constant values.** This example uses an assembler macro to declare a constant value in a separate assembler section. You can use this technique to keep instructions, fixed data values, and variable data values separate, so that you could eventually place your program into ROM.
- **Creating and using a subroutine library.** This example shows how you can build a library (a skeleton floating-point package), and then use parts of that library at a later time. Relevant parts of the assembler, linker, and library generator are illustrated.
- **Linking overlays.** This example shows how you can use the linker to link overlay modules.
- **Using the "@" construct within macros.** This example shows typical uses of the "@" construct within macros.
- **The assembler INCLUDE directive.** This example shows some typical uses of the INCLUDE directive, such as providing common constant, COMMON, or macro declarations. It also shows how to provide a copyright or authorship notice for your listings.

USE OF CONDITIONAL ASSEMBLY IN MACROS

This example illustrates some uses of the IF-ELSE-ENDIF and IF-ENDIF constructs for conditional assembly in macros.

Conditional assembly is used primarily in macros. The main body of the program is usually structured such that, once it is written, few changes will need to be made. Macros, however, are designed to examine their parameters and make decisions which may vary with programming and run-time conditions.

One use of conditional assembly in macros is to assemble statements only upon the first invocation of the macro. For example, an error will occur if a string variable is defined more than once; the following structure may be used to check for previous definitions.

```
IF      \DEF(STRI)      ; If STRI has not been defined
STRING STRI(100)      ; Define STRI with length of 100 bytes
ENDIF
```

These instructions determine whether or not the string variable STRI has been defined previously. If it has not, the statement STRING STRI(100), which defines a string variable named STRI with a maximum length of 100 characters, is assembled.

Another use of conditional assembly in macros is to verify that a symbol has previously been defined.

```
MACRO  CALLPRINT
IF      \DEF(PRINT)    ; If PRINT has not been defined yet...
GLOBAL PRINT          ; Define PRINT as a global
ENDIF                    ; Continue with rest of macro
MOV     AX, "1"        ; Move first parameter to AX
LD      BX, "2"        ; Move second parameter to BX
CALLS  PRINT, PRINT
ENDM
```

The conditional block in this macro checks to see if PRINT has already been defined. If PRINT has not been defined, the statement GLOBAL PRINT is assembled. If PRINT has been defined previously, the statement in the conditional block (GLOBAL PRINT) is skipped.

SAVE-AND-RESTORE MACRO

This example uses two assembler macros to perform a common assembly language operation: saving and restoring microprocessor registers. The example uses the 8086/8088 instruction set; however, the techniques illustrated here can be applied to nearly all stack-oriented microprocessors.

The SAVE Macro

The SAVE macro saves one or more registers on the stack. The parameters of the macro invocation line designate the registers to be saved. The body of this macro examines those parameters and generates the appropriate 8086/8088 PUSH instructions.

```

MACRO   SAVE                ; line 1
IF     "#"                  ; line 2
SAVE$  SET    1              ; line 3
      REPEAT SAVE$ <= "#"  ; line 4
      PUSH   "SAVE$"       ; line 5
SAVE$  SET    SAVE$ + 1     ; line 6
      ENDR                    ; line 7
      ELSE                    ; line 8
      SAVE   AX, BX, CX, DX ; line 9
      ENDIF                  ; line 10
      ENDM                   ; line 11

```

Line 1 begins the macro definition, and gives the macro the name SAVE. This name is used in the program to invoke the macro.

Line 2 begins an IF..ELSE..ENDIF block. This IF statement has one operand: the construct "#". The assembler will replace this construct with the number of parameters present in the macro invocation line. If the parameter count is not zero, the assembler processes all statements between this IF statement and the corresponding ELSE statement (line 8). If the parameter count is zero (meaning that the macro was invoked with no parameters), the assembler processes the statements between the ELSE and ENDIF statements.

Lines 3 through 7 are processed if the macro invocation includes one or more parameters. Each parameter is the name of one 8086/8088 register. The macro generates one PUSH instruction for each parameter. A REPEAT..ENDR loop processes each parameter in turn.

Line 3 initializes the assembler variable SAVE\$ to 1. This assembler variable is incremented once for each parameter in the macro invocation line.

Line 4 is the beginning of the REPEAT..ENDR block. The statements in the block are repeated as long as the assembler variable SAVE\$ is not greater than the number of parameters passed to the macro ("#").

Line 5 generates an 8086/8088 PUSH instruction. The operand of the PUSH instruction is obtained from the current value of SAVE\$. For example, if SAVE\$ is 3, and the third parameter in the macro invocation is CX, this statement generates an 8086/8088 PUSH CX instruction.

Line 6 increments the value of the assembler variable SAVE\$.

Line 7 marks the end of the REPEAT..ENDR block. As long as the expression in the REPEAT statement is true (non-zero), the assembler will process the group of statements within the REPEAT..ENDR block.

Line 8 terminates the IF..ELSE block.

Line 9 is processed only when the IF condition (in line 2) is false ($\# = \text{zero}$). If SAVE is invoked with no parameters, the SAVE macro reinvokes itself with the four general purpose registers as parameters, thereby saving all four general purpose registers.

Line 10 terminates the IF..ELSE..ENDIF block.

Line 11 statement terminates the definition of the macro.

The RESTORE Macro

The RESTORE macro retrieves the registers that were saved on the stack.

```

MACRO  RESTORE
IF     "#"
RESTORE$ SET 1
      REPEAT RESTORE$ <= "#"
      POP   "RESTORE$"
RESTORE$ SET RESTORE$ + 1
      ENDR
      ELSE
      RESTORE DX, CX, BX, AX
      ENDF
      ENDM

```

The RESTORE macro is similar to the SAVE macro, with two changes:

1. The assembler variable is named RESTORE\$ in this macro.
2. The order of the registers in the default macro invocation (no parameters) is reversed. The stack operates in a last-in-first-out (LIFO) manner: the last register saved must be the first register restored.

Sample Invocations

The SAVE and RESTORE macros are most commonly used at the beginning and end of subroutines to insure that the subroutine does not destroy values in the registers needed by the calling routine. For example, if all general purpose registers are used in a subroutine, you can include the SAVE macro invocation (with no parameters) at the subroutine's beginning, and the RESTORE macro invocation (again, with no parameters) at the subroutine's end, like this:

```

SUBR   SAVE           ; Beginning of subroutine SUBR; save all registers
      ...
      ...           ; Body of the subroutine
      ...
      RESTORE        ; Restore all registers
      RET           ; 8086/8088 return-from-subroutine instruction

```

If some (but not all) registers are used in the subroutine, you can invoke SAVE and RESTORE with a list of those registers to be saved on the stack. Note that the order of the registers must be reversed when restoring them from the stack.

```

SUBR   SAVE   AX, BX           ; Save AX and BX
      ...
      ...                       ; Body of subroutine
      ...
      RESTORE BX, AX          ; Restore BX and AX
      RET                                ; Return from subroutine
    
```

SVC GENERATION

This example illustrates some of the features of the assembler that can make it easier for you to use service calls. Two examples are discussed; creating service request blocks (SRBs), and generating the microprocessor service call (SVC) instructions. These examples use the 8086/8088 instruction set, but similar techniques can be applied to most processors.

These examples assume you are familiar with the use of SVCs, as described in your system user's manual.

Creating Service Request Blocks

The first task in using an SVC is to create an appropriate SRB. The SRB consists of seven fields (12 bytes):

SRB Field Name	Bytes Used
Function	1
Channel	2
Status	3
Reserved byte	4
Byte count	5-6
Buffer length	7-8
Buffer address	9-12

The buffer (specified by the last 6 bytes) is used for I/O operations. This example uses a macro to set up an SRB. Based on the parameters passed to it, the macro determines such things as whether to generate an SRB location vector, what the names of the SRB components are, and what the size of the I/O buffer is. The following assembler source statements define a macro that performs these functions. A line-by-line description follows the listing.

The SRB Macro

```

        STRING  SRB$SEC(8), SRB$BUF(16)          ; line 1
        MACRO   SRB                               ; line 2
SRB$SEC SET    '""'                               ; line 3
        IF     DEF(SRB.SEC)                       ; line 4
        RESUME SRB.SEC                           ; line 5
        ELSE                                     ; line 6
        SECTION SRB.SEC                          ; line 7
        ENDIF                                     ; line 8
"1".FUN BLOCK 1                                  ; line 9
"1".CHN BLOCK 1                                  ; line 10
"1".STA BLOCK 1                                  ; line 11
"1".RES BLOCK 1                                  ; line 12
"1".CNT BLOCK 2                                  ; line 13
        IF     STRINGOF(3)=''                     ; line 14
"1".LEN BLOCK 2                                  ; line 15
"1".BU1 BLOCK 1                                  ; line 16
"1".BU2 BLOCK 1                                  ; line 17
"1".BU3 BLOCK 1                                  ; line 18
"1".BU4 BLOCK 1                                  ; line 19
        ELSE                                     ; line 20
        IF     STRINGOF(4)=''                     ; line 21
SRB$BUF SET    "'1'.BUF'                         ; line 22
        ELSE                                     ; line 23
SRB$BUF SET    STRINGOF(4)                       ; line 24
        ENDIF                                     ; line 25
"1".LEN WORD   "3"                               ; line 26
"1".BU1 BLOCK 1                                  ; line 27
"1".BU2 BLOCK 1                                  ; line 28
"1".BU3 BYTE   HI("SRB$BUF")                    ; line 29
"1".BU4 BYTE   LO("SRB$BUF")                    ; line 30
        IF     DEF(BUF.SEC)                       ; line 31
        RESUME BUF.SEC                           ; line 32
        ELSE                                     ; line 33
        SECTION BUF.SEC                          ; line 34
        ENDIF                                     ; line 35
"SRB$BUF" BLOCK "3"                              ; line 36
        ENDIF                                     ; line 37
        IF     STRINGOF(2)<>'                     ; line 38
        IF     DEF(SRB.VEC)                       ; line 39
        RESUME SRB.VEC                           ; line 40
        ELSE                                     ; line 41
        SECTION SRB.VEC, ABSOLUTE                 ; line 42
        ENDIF                                     ; line 43
        ORG   40H+4*("2"-1)                       ; line 44
        BYTE HI("1".FUN)                          ; line 45
        BYTE LO("1".FUN)                          ; line 46
        ENDIF                                     ; line 47
        RESUME "SRB$SEC"                          ; line 48
        ENDM                                       ; line 49

```

Explanation of the SRB Macro

The macro is invoked with the following parameters:

1. The first parameter is the name of the SRB. The name must be one to twelve characters long, and must be a valid symbol prefix. Many labels describing parts of the SRB and buffer are derived from this name.

2. The second parameter is optional; if present, it designates the SVC number (1 to 8) that is used with the SRB. If you provide this number, the macro creates the appropriate pointer to this SRB in the 40H to 5FH area of memory. If you omit this parameter, you must use other assembler statements to supply the pointer.
3. The third parameter is optional; if present, it designates the size of I/O buffer associated with the SRB. The last six bytes of the SRB are altered to describe the buffer's size and location, and a buffer of this size is created. If you omit the third parameter, the last six bytes of the SRB are left empty.
4. The fourth parameter is optional; if present, it selects the name of the buffer associated with this SRB. If you omit this parameter, the macro assigns a name derived from the SRB name. This parameter is ignored if the third parameter is not present.

Line 1 creates two assembler string variables: SRB\$SEC and SRB\$BUF. These variables are used within the body of the assembler macro to temporarily store data, so that it may be retrieved later in the macro. These variables are discussed further when they are used.

Line 2 defines the beginning of the macro, and gives the macro the name SRB.

Line 3 saves the current section name in the assembler variable SRB\$SEC. The current section name is saved so that it may be restored later; the remaining statements in this macro switch sections at least once.

Lines 4 through 8 switch the current section to SRB.SEC, so that later assembler statements can generate object bytes for an SRB. The IF statement determines if the section SRB.SEC was previously defined: if so, a simple RESUME statement is processed, to continue object code generation; if not, the section begins with a SECTION statement, as its first definition. This technique of using IF DEF(section-name) to conditionally resume a section is used again, starting at lines 31 and 39.

Lines 9 through 13 define the common part of the SRB. Each byte of the SRB is given a descriptive name (label). This label consists of the SRB name (given as the first parameter at invocation) followed by a four-character suffix. The suffix for each SRB byte indicates the function of that byte. For example, if the first parameter at macro invocation is QQ, then the five bytes generated by these five lines of code are: QQ.FUN (function), QQ.CHN (channel), QQ.STA (status), QQ.RES (reserved byte), and QQ.CNT (I/O count).

Lines 14 through 37 generate the last six bytes of the SRB, according to the third and fourth parameters, and create the buffer if necessary. Three possible combinations exist:

1. **No third parameter;** The last six bytes of the SRB are generated like the first five: labels are generated and space is allocated, but no values are inserted into the SRB bytes.
2. **Third parameter only;** The last six bytes of the SRB describe a buffer generated by this macro. The name of the buffer is derived from the name of the SRB, in the same way as the name of the SRB components.
3. **Both third and fourth parameters;** Again, the last six bytes of the SRB describe a buffer generated by this macro, but the name of the buffer is explicitly given by the fourth parameter.

Line 14 examines the third parameter: if it is absent, lines 15 through 19 are assembled; if present, lines 21 through 36 are assembled. In either case, only one block of statements is assembled.

Lines 15 through 19 generate the last six bytes of the SRB when the third parameter is absent. Again, the names of the six bytes are derived from the SRB name given in the macro invocation line. If the SRB name is QQ, for example, six bytes are generated: QQ.LEN (length of buffer—two bytes), QQ.BU1, QQ.BU2, QQ.BU3, and QQ.BU4 (four bytes for buffer address).

Lines 21 through 25 determine the name of the buffer. If the fourth parameter is present, it is used as the buffer name. If the fourth parameter is absent, the name of the buffer is created from the SRB name; for example, an SRB name of QQ produces a buffer name of QQ.BUF. In either case, the buffer name is assigned to the assembler string variable SRB\$BUF. This variable is used later in the macro.

Lines 26 through 30 generate the last six bytes of the SRB, using the given size and name of the buffer. As with the other bytes of the SRB, each of these bytes is given a label derived from the SRB name. For example, a SRB name of QQ generates the labels QQ.LEN, QQ.BU3, and QQ.BU4. However, unlike the other bytes of the SRB, these bytes are given values at assembly time. Because the location and size of the buffer are known, the correct values can be given to these bytes.

Lines 31 through 35 change the current section to BUF.SEC, using the method described previously (lines 4 through 8). Section BUF.SEC contains any I/O buffers generated by the macro.

Line 36 generates the I/O buffer. The name is defined in the assembler string variable SRB\$BUF. The size is taken from the third invocation parameter.

Lines 38 through 47 generate a pointer to the SRB in the SRB vector (fixed locations 40H to 5FH). These lines are assembled only if the second parameter is present.

Lines 39 through 43 define SRB.VEC as the current section. This section is absolute (non-relocatable), because the vectors must be in fixed locations in memory.

Line 44 generates an assembler ORG directive to place the pointer in the proper location. The operand of the ORG directive computes an address from the second parameter in the invocation; this parameter is a digit from 1 to 8.

Lines 45 and 46 generate a pointer to the SRB's first entry, the function byte.

Line 48 restores the current section to the section name that was saved upon entry into this macro.

Line 49 terminates the definition of the macro.

Sample Invocations of the SRB Macro

The SRB macro can be invoked in many different ways, depending on the needs of the situation. For example, in its simplest invocation,

```
SRB    QQQ
```

only an SRB is generated. The first parameter, QQQ, specifies the name of the SRB. The SRB consists entirely of BLOCK assembler directives; your program is expected to place values into the various bytes of the SRB.

The SRB macro can be invoked with an SVC number:

```
SRB    RRR, 4
```

The SRB macro automatically places the appropriate pointer to the SRB (named RRR) at locations 4CH to 4FH. Again, no part of the SRB is given a value at assembly time; your program must supply all values (including pointers to buffers) at program execution time.

If you wish to specify a buffer, you must include the third and fourth parameters. For example,

```
SRB    SSS, , 128, BUFFER
```

specifies a BUFFER that is 128 bytes long. The macro creates the buffer and places values describing the location and length of the buffer into the last 6 bytes of the SRB.

If you do not require a specific buffer name, you may omit the fourth parameter. The buffer name will be derived from the SRB name. You still specify the third parameter, to tell the macro the length of buffer to be created. For example, the macro invocation

```
SRB    TTT, , 64
```

creates a 64-byte buffer named TTT.BUF.

You can create the buffer and SRB pointer simultaneously by including all four parameters in the SRB macro. For example, the invocation

```
SRB    UUU, 3, 80, MYBUF
```

creates an SRB named UUU, a pointer to the SRB at addresses 48H to 4BH (the SVC 3 vector location), and an 80-byte buffer named MYBUF.

Generating Service Calls

The task of generating a service call consists of placing three microprocessor-dependent instructions in your program. The first instruction is usually a data transfer instruction; the second and third are no-operation instructions. For an 8086/8088 microprocessor, the OUT and NOP instructions are used for SVCs.

You can use an assembler macro to assist you in creating the OUT/NOP instruction sequence. The following listing presents a sample macro. A line-by-line description follows the listing.

The SVC Macro

```
MACRO   SVC                               ; line 1
IF      STRINGOF(1)=''                    ; line 2
WARNING ; Missing SVC Number              ; line 3
ELSE                                          ; line 4
OUT     OFFF8H-"1", AL                     ; line 5
NOP                                           ; line 6
NOP                                           ; line 7
ENDIF                                       ; line 8
ENDM                                         ; line 9
```

Explanation of the SVC Macro

This macro is invoked with one parameter, the SVC number: a single digit between 1 and 8.

Line 1 defines the name of the SVC-generation macro.

Lines 2 through 8 form an IF..ELSE..ENDIF block. If the first parameter is absent, line 3 is processed. If the first parameter is present, lines 5, 6 and 7 are processed.

Line 3 (processed only if no first parameter is given) generates an error message. This message indicates that the required parameter has not been given in this invocation of the macro. The error message appears on the listing and the system terminal.

Lines 5, 6 and 7 generate an 8086/8088 service call instruction sequence. Line 5 generates an OUT instruction; the address of the OUT instruction is computed from the first macro parameter. Lines 6 and 7 generate the NOP (no-operation) instructions to allow time for the service call to be processed.

Line 9 terminates the macro definition.

Sample Invocation of the SVC Macro

The SVC macro is invoked with one parameter, the SVC number. For example,

```
SVC     4
```

generates the proper instruction sequence for SVC 4. If the parameter is omitted, an error message is generated.

CREATING CONSTANT VALUES

This example illustrates the use of a macro to declare a constant value in a separate assembler section. In this example, two versions of the macro are shown: one to define values to be stored in ROM, and the other to define values to be stored in RAM. By using these two macros, you can store constants in either ROM or RAM from anywhere within your program.

Here's how the macro works: first, it switches from the current section to an alternate section. Then, it generates the object code for the statements specified. Finally, it switches back to the original section. By using statements with data storage directives (such as ASCII, BYTE, BLOCK, and WORD), you can store values in the alternate section.

The macro may be invoked by one of two methods:

- **Method 1.** The statement lines to be assembled in the alternate section are passed as parameters in the operand field of the macro invocation.
- **Method 2.** The statement lines to be assembled in the alternate section are a sequence of lines following the macro invocation. The macro invocation has no parameters in the operand field. The invocation of a second macro terminates the sequence of lines and resumes the original section.

Sample invocations are presented later in this example.

The CONSTANT Macro

This version of the macro stores values in a section ROM.CODE, which can be assigned to ROM memory at link time.

```

        STRING    CON$SAVE,CON$SEC ; line 1
CON$SEC SET      'ROM.CODE'      ; line 2

        MACRO    CONSTANT          ; line 3
CON$SAVE SET      "%"             ; line 4
        IF      DEF("CON$SEC")    ; line 5
        RESUME  "CON$SEC"         ; line 6
        ELSE    ; line 7
        SECTION "CON$SEC"        ; line 8
        ENDIF   ; line 9
        IF      "#"               ; line 10
CON$CNT SET      1                ; line 11
        REPEAT  CON$CNT <= "#"    ; line 12
"CON$CNT"
CON$CNT SET      CON$CNT + 1      ; line 14
        ENDR   ; line 15
        ENDCONSTANT              ; line 16
        ENDIF   ; line 17
        ENDM   ; line 18

        MACRO    ENDCONSTANT      ; line 19
        RESUME  "CON$SAVE"        ; line 20
        ENDM   ; line 21

```

Line 1 creates two string assembler variables, CON\$SAV and CON\$SEC. These variables are used within the body of the macro to temporarily store data.

Line 2 assigns the character string 'ROM.CODE' to the variable CON\$SEC. The variable is used for the name of the section in which the constants are stored.

Line 3 defines the beginning of the macro and gives it the name CONSTANT.

Line 4 saves the current section name in the variable `CON$SAVE` so that the section may be resumed later.

Lines 5 through 9 switch the current section to the section `ROM.CODE` (the value of the variable `CON$SEC`). The `IF` statement determines whether or not the section `ROM.CODE` was previously defined (started): if so, the `RESUME` statement (line 6) continues the section definition; if not, the `SECTION` statement (line 8) begins the section definition.

Line 10 tests for the presence of a parameter. The assembler replaces the construct `"#"` with the number of parameters in the macro invocation line. If the parameter count is non-zero, the assembler processes lines 11 through 16. Otherwise, the assembler skips to line 18.

Line 11 initializes the assembler variable `CON$CNT` to designate the first parameter. This variable is incremented later (line 14) for each parameter.

Lines 12 through 15 form a conditional repeat block. In this block, the invocation parameters are processed within the macro. The first time the repeat loop is processed, the value of `CON$CNT` is 1, and the construct `"CON$CNT"` (in line 13) is replaced by the first parameter. As `CON$CNT` is incremented (line 14), each successive parameter is processed, until the value of `CON$CNT` exceeds the number of parameters passed (`"#"` in line 12).

Line 16 invokes the macro `ENDCONSTANT`, which is defined in lines 19 to 21.

If `"#"` was zero in line 10, the assembler proceeds to line 18 (the first statement following the `ENDIF`). This statement terminates the macro. The assembler then processes the statement lines following the invocation of the `CONSTANT` macro. These statements provide data for section `ROM.CODE`. The statement lines will continue to be processed within section `ROM.CODE` until macro `ENDCONSTANT` is invoked.

Lines 19 through 21 define macro `ENDCONSTANT`. The macro `ENDCONSTANT` switches back to the section name that was saved at the beginning of this macro (line 4).

The VARIABLE Macro

A similar macro can be created to store variables in RAM. The section RAM.CODE can be assigned to RAM memory at link time.

```

        STRING  VAR$SAVE, VAR$SEC
VAR$SEC SET     'RAM.CODE'

        MACRO   VARIABLE
VAR$SAVE SET     ""
        IF     DEF("VAR$SEC")
RESUME    "VAR$SEC"
        ELSE
SECTION   "VAR$SEC"
        ENDIF
        IF     ""
VAR$CNT  SET     1
        REPEAT VAR$CNT <= ""
"VAR$CNT"
VAR$CNT  SET     VAR$CNT + 1
        ENDR
ENDVARIABLE
        ENDIF
        ENDM

```

The ENDVARIABLE macro definition is:

```

        MACRO   ENDVARIABLE
RESUME    "VAR$SAVE"
        ENDM

```

Macro Invocation

Assume that you want to store a character string in a section of ROM memory and call a routine to print that character string. (This example assumes that your program supplies a subroutine PRINT in a section named OUTPUT. The subroutine prints each successive character pointed to by the BX register until a return character is encountered.) The following invocation of the macro CONSTANT could be used to store the message to be printed.

```

SECTION PRINCON
LEA     BX, MES1
CONSTANT MES1 ASCII 'HELLO THERE', [ BYTE 13]

```

1st parameter
2nd parameter

```

CALLS   PRINT, OUTPUT

```

The first line declares section PRINCON.

The second line is an 8086/8088 instruction that loads the BX register with a pointer to MES1.

The third line invokes the macro `CONSTANT` with two parameters. The first parameter is an assembler statement that stores the ASCII representation of the character string 'HELLO THERE' and has the location `MES1`. The second parameter [`BYTE 13`] generates one byte of data with the value 13 (the ASCII return character). The space in the first position of the parameter causes the `BYTE` to be treated as an assembler directive, and not as a label. These two lines are processed within the section `ROM.CODE`. The macro then switches back to the section `PRINCON`.

The last line of this example, `CALLS PRINT, OUTPUT` invokes the subroutine `PRINT`.

If you invoke the macro `CONSTANT` without parameters, it simply switches to section `ROM.CODE`. Any assembler statements between the invocation of `CONSTANT` (without parameters) and a matching invocation of `ENDCONSTANT` are generated into section `ROM.CODE`. For example, the following assembler statements produce identical results to the previous example:

```

SECTION    PRINCON
LEA       BX,MES1
CONSTANT
MES1     ASCII    'HELLO THERE'
BYTE     13
ENDCONSTANT
CALLS    PRINT, OUTPUT

```

In this invocation, the invocation of macro `ENDCONSTANT` terminates the alternate section and resumes the original section.

With the use of macro `VARIABLE`, you could establish a data block in a section destined for `RAM`. In this example, the symbol `DATA.TAB` points to a block of 512 bytes. The macro can be invoked with either this sequence of statement lines:

```

LEA      BX,DATA.TAB
VARIABLE DATA.TAB BLOCK 512
CALL     PROCESS

```

or this sequence:

```

LEA      BX,DATA.TAB
VARIABLE
DATA.TAB BLOCK 512
ENDVARIABLE
CALL     PROCESS

```

CREATING AND USING A SUBROUTINE LIBRARY

This example shows you how to create a library using the assembler and library generator, and how to write programs that use selected modules from the library.

The example develops a portion of a floating-point package. The floating-point package uses processor instructions to manipulate floating-point numbers such as 10000. or π (3.14159...). For this example, assume that any floating-point number can be stored in eight consecutive bytes. (The method of storage is not relevant to this example.)

To keep things simple, only two primitive floating-point operations are shown: addition and subtraction. Modules that perform these two operations are the nucleus of the library. Later, other modules, such as multiplication, could be added to the library.

In this example, the addition and subtraction modules are written as subroutines. They pass and return data using a predefined COMMON section: a floating-point accumulator.

This example, then, consists of seven major tasks:

1. Develop the library ADD module.
2. Develop the library SUBTRACT module.
3. Assemble the modules.
4. Create the floating-point library from the two object modules using the library generator.
5. Develop a sample mainline program that uses the library module ADD.
6. Assemble and link the sample mainline program.
7. Develop, assemble, and link a parallel mainline program that uses the library module SUBTRACT.

The ADD Module

The following assembler source statements present a 'skeleton' of the library ADD module. The actual microprocessor instructions to perform the addition are not included, but are represented by assembler BLOCK directives of comparable length. A line-by-line description of the source module follows the listing.

```

          LIST      LINE(80)           ; line 1
          NAME      FP$ADD             ; line 2
          GLOBAL    FP.ADD, FP.AD2     ; line 3
          COMMON    FP$ACC             ; line 4
SRC1      BLOCK    8                   ; line 5
SRC2      BLOCK    8                   ; line 6
DEST      BLOCK    8                   ; line 7
          SECTION  FP_ADD             ; line 8
FP.ADD    BLOCK    40                  ; line 9
FP.AD2    BLOCK    350                 ; line 10
          END                      ; line 11

```

Explanation of the ADD Module

Line 1 specifies that the assembler listing contain a maximum of 80 characters per line. Any line longer than this will be truncated in the listing.

Line 2 declares the name of the object module. This name is essential in all LibGen references to this particular library element. The name (FP\$ADD) indicates the module's function (floating-point addition).

Line 3 designates FP.ADD and FP.AD2 as global symbols. Both of these symbols are defined in this module. These symbols are entry points into the subroutine; they are used by other modules to select this library module at link time.

Lines 4 through 7 define the structure of the floating-point accumulator. This COMMON section is named FP\$ACC (floating-point accumulator). The accumulator provides space for three floating-point numbers: two operands (SRC1 and SRC2) and a result (DEST).

Lines 8 through 10 define the assembler section containing the instructions that perform the addition. The BLOCK directives represent the approximate number of bytes consumed by the instructions. Two entry points are defined in this section: FP.ADD and FP.AD2.

Line 11 designates the end of this assembler module.

Entry Points

This library module defines two entry points:

- **FP.ADD**—Your program can call this subroutine at FP.ADD to add SRC1 to DEST, leaving the result in DEST. This entry point is useful when you are maintaining a running total. To simplify the discussion, assume that the routine beginning at FP.ADD simply copies the contents of DEST to SRC2, then falls through to the routine at FP.AD2.
- **FP.AD2**—Your program can call this subroutine at FP.AD2 to add SRC1 to SRC2, leaving the result in DEST. This entry point is used when you do not wish to incur the additional overhead of the first entry point.

The SUBTRACT Module

The SUBTRACT module, as represented here, is very similar to the ADD module. The assembler statements present a 'skeleton' of this SUBTRACT module. A line-by-line description of the source module follows the listing.

```

LIST      LINE(80)           ; line 1
NAME      FP$SUB             ; line 2
GLOBAL    FP.SUB, FP.SU2     ; line 3
GLOBAL    FP.AD2, FP_ADD     ; line 4
COMMON    FP$ACC             ; line 5
SORC1     BLOCK 8            ; line 6
SORC2     BLOCK 8            ; line 7
DST       BLOCK 8            ; line 8
          SECTION FP_SUB     ; line 9
FP.SUB    BLOCK 70           ; line 10
FP.SU2    BLOCK 30           ; line 11
          CALLS  FP.AD2, FP_ADD ; line 12
          BLOCK 35           ; line 13
          END                ; line 14
    
```

Explanation of the SUBTRACT Module

Line 1 specifies that the assembler listing contain a maximum of 80 characters per line. Any line longer than this will be truncated in the listing.

Line 2 declares the name of the object module: FP\$SUB (floating-point subtraction).

Line 3 designates FP.SUB and FP.SU2 as global symbols. These address symbols form entry points into this routine.

Line 4 declares FP.AD2 as a global symbol. Unlike the other global symbols, FP.AD2 is defined in another module (the ADD module). When the SUBTRACT module is linked with a program, the linker notes the FP.AD2 symbol and attempts to locate a definition for it in another module.

Lines 5 through 8 define the structure of the floating-point accumulator. The COMMON section is named FP\$ACC, as before. However, the components of FP\$ACC are given different names in this module: the operands are named SORC1 and SORC2, while the destination is named DST. This module illustrates how two modules can refer to the same portions of memory with independently selected names.

Lines 9 through 13 define the assembler section that contains the instructions that perform the subtraction. Two entry points are defined here: FP.SUB and FP.SU2.

Line 14 designates the end of this assembler routine.

Entry Points

This library module defines two entry points:

- **FP.SUB**—Your program can call this subroutine at FP.SUB to subtract SORC1 from DST, leaving the result in DST. The contents of DST are copied to SORC2 and execution continues at FP.SU2.
- **FP.SU2**—Your program can call the subroutine at FP.SU2 to subtract SORC1 from SORC2, leaving the result in DST. The subtraction is accomplished by changing the sign of SORC1, and calling FP.AD2. (The 8086/8088 instruction at line 12 is a call to a subroutine, and returns to the address following the instruction.)

Assembling the Modules

You may use the operating system editor to enter these two modules into their respective assembler source files. Place the ADD module in a file named **fpa.asm**, and the SUBTRACT module in a file named **fps.asm**. After entering the programs, you may assemble them to generate the necessary object modules for the library.

Enter the following command to assemble the source file **fpa.asm** into the object file **fpa.obj**. The listing is output to the file **fpa.asml**, so that you may examine it.

```
asm fpa.obj fpa.asml fpa.asm [assembler invocation]
```

Now look at **fpa.asml**:

Tektronix ASM 8086/8088
Vxx.xx-xx (xxxx)

Page 1
dd-mmm-yy/xx:xx:xx

```

1          LIST      LINE(80)          ; line 1
2          NAME      FP$ADD            ; line 2
3          GLOBAL    FP.ADD, FP.AD2    ; line 3
4          COMMON    FP$ACC            ; line 4
5 00000000      8      SRC1      BLOCK  8      ; line 5
6 00000008      8      SRC2      BLOCK  8      ; line 6
7 00000010      8      DEST      BLOCK  8      ; line 7
8          SECTION   FP_ADD            ; line 8
9 00000000      28     FP.ADD      BLOCK  40     ; line 9
10 00000028     15E    FP.AD2     BLOCK  350    ; line 10
11          END                                ; line 11
    
```

Tektronix ASM 8086/8088 SYMBOL TABLE
Vxx.xx-xx (xxxx)

Page 2
dd-mmm-yy/xx:xx:xx

Section = %FPAOBJ, Aligned to 00000010, Size = EMPTY

Common Section = FP\$ACC, Aligned to 00000010, Size = 00000018

```

DEST-----00000010      SRC1-----00000000
SRC2-----00000008
    
```

Section = FP_ADD, Aligned to 00000010, Size = 00000186

```

FP.AD2-----00000028 G   FP.ADD-----00000000 G
    
```

```

11 Lines Read
11 Lines Processed
0 Errors
    
```

Now assemble **fpa.asm** into **fpa.obj**:

asm fpa.obj fpa.asml fpa.asm [assembler invocation]

The listing file contains:

Tektronix ASM 8086/8088
Vxx.xx-xx (xxxx)

Page 1
dd-mmm-yy/xx:xx:xx

```

1          LIST      LINE(80)          ; line 1
2          NAME      FP$SUB            ; line 2
3          GLOBAL    FP.SUB, FP.SU2    ; line 3
4          GLOBAL    FP.AD2, FP.ADD    ; line 4
5          COMMON    FP$ACC            ; line 5
6 00000000      8      SORC1  BLOCK  8      ; line 6
7 00000008      8      SORC2  BLOCK  8      ; line 7
8 00000010      8      DST    BLOCK  8      ; line 8
9          SECTION   FP_SUB            ; line 9
10 00000000     46      FP.SUB  BLOCK  70     ; line 10
11 00000046     1E      FP.SU2  BLOCK  30     ; line 11
12 00000064 9A000000 R    CALLS   FP.AD2, FP.ADD ; line 12
           00
13 00000069     23      BLOCK  35          ; line 13
14          END                                ; line 14
    
```

Tektronix ASM 8086/8088 SYMBOL TABLE
Vxx.xx-xx (xxxx)

Page 2
dd-mmm-yy/xx:xx:xx

Section = %FPSOBJ, Aligned to 00000010, Size = EMPTY

Common Section = FP\$ACC, Aligned to 00000010, Size = 00000018

```

DST-----00000010      SORC1-----00000000
SORC2-----00000008
    
```

Section = FP_SUB, Aligned to 00000010, Size = 0000008C

```

FP.SU2-----00000046 G  FP.SUB-----00000000 G
    
```

Unbound Globals

```

FP.AD2-----00000000      FP_ADD-----00000000
    
```

14 Lines Read
14 Lines Processed
0 Errors

Creating the Library

Now you can use the library generator (LibGen) to create the floating-point library. LibGen is discussed in the Library Generator section of this manual.

Enter the underlined characters to create the floating-point library **fp.lib** from the two object modules.

```
libgen -l -v -n fp.lib -i fps.obj fpa.obj >libgen.lst           [LibGen invocation]
```

Look at the listing in **libgen.lst**.

```
Tektronix Library Generator      Version x.x.x  COMMAND LOG                Page  1
```

```
-n fp.lib
-i fpa.obj
-i fpa.obj
```

```
Tektronix Library Generator      Version x.x.x  SUMMARY OF ACTION          Page  2
```

```
New Library Generated:
fp.lib
```

```
Module: FP$SUB      INSERTED
      from: fps.obj
***libgen : 8(W)Duplicate symbol name: FP$ACC
Module: FP$ADD      INSERTED
      from: fpa.obj
```

```
Tektronix Library Generator      Version x.x.x  MODULE LISTING             Page  3
```

```
New Library Generated:
fp.lib
```

```
Module: FP$ADD
  Int Symbols: (S)FP_ADD;          (D)FP.AD2;          (D)FP.ADD;
               (S)%FPAOBJ;
```

```
Module: FP$SUB
  Int Symbols: (S)FP_SUB;          (D)FP.SU2;          (S)FP.SUB;
               (S)FP$ACC;         (S)%FPSOBJ;
```

```
Ext Symbols: (D)FP_ADD;          (D)FP.AD2;
Ext References:
              FP$ADD
```

```
***END OF LISTING***
```

Using the ADD Module from a Program

The information stored in the library can be used by a mainline program that references one of the library module's global entry points. The following mainline program uses the FP\$ADD module of the library; a line-by-line annotation follows the listing.

The Mainline Add Program

```

LIST      LINE(80)           ; line 1
NAME      MAIN.ADD           ; line 2
GLOBAL    FP.ADD, FP_ADD     ; line 3
COMMON    FP$ACC             ; line 4
S1        BLOCK 8             ; line 5
S2        BLOCK 8             ; line 6
DESTN     BLOCK 8             ; line 7
SECTION   MAIN               ; line 8
ENTRY     BLOCK 40           ; line 9
          CALLS FP.ADD,FP_ADD ; line 10
MORE      BLOCK 50           ; line 11
END       ENTRY              ; line 12

```

Explanation of the Mainline Add Program

Line 1 specifies that the assembler listing contain a maximum of 80 characters per line. Any line longer than this will be truncated in the listing.

Line 2 declares the name of the object module to be MAIN.ADD.

Line 3 declares the symbol FP.ADD as a global symbol. This symbol is not defined in this object module; therefore, the symbol is called an 'unbound' global. At link time, the linker will attempt to locate a definition for FP.ADD; the library **fp.lib** (created earlier) will provide this definition.

Lines 4 through 7 define the structure of the floating-point accumulator. In this module, the two source fields and the destination field are called S1, S2, and DESTN.

Line 8 begins the definition of the main section (called MAIN). All object bytes generated after this directive are in the MAIN section.

Line 9 sets aside memory space for processor instructions; these instructions load values into S1 and DESTN for processing. In a functional program, this BLOCK directive would be replaced with microprocessor instructions, such as data transfer instructions or I/O operations.

Line 10 is an 8086/8088 instruction that calls the subroutine FP.ADD (contained in the floating-point library). The contents of S1 are added to the contents of DESTN, and the subroutine returns to the memory location following the CALLS instruction.

Line 11 represents more microprocessor instructions following the invocation of the ADD routine. These instructions might perform some type of output to display the results of the addition.

Line 12 defines the end of this source module. The symbol ENTRY is designated as the starting address of the instructions. From this value, the linker will determine the transfer address of the module.

Assembling and Linking the Program

You may use your operating system editor to enter the mainline add program. Place the mainline add program in a file named `mna.asm`. After entering the mainline add program, you can assemble and link it using the following command entries:

```
asm mna.obj mna.asml mna.asm      [assembler invocation]
```

Now look at `mna.asml` :

```
Tektronix ASM 8086/8088
Vxx.xx-xx (xxxx)
```

```
Page 1
dd-mmm-yy/xx:xx:xx
```

```

1          LIST      LINE(80)          ; line 1
2          NAME      MAIN.ADD          ; line 2
3          GLOBAL    FP.ADD, FP_ADD    ; line 3
4          COMMON    FP$ACC            ; line 4
5 00000000      8      S1      BLOCK    8          ; line 5
6 00000008      8      S2      BLOCK    8          ; line 6
7 00000010      8      DESTN   BLOCK    8          ; line 7
8          SECTION   MAIN              ; line 8
9 00000000      28     ENTRY    BLOCK    40         ; line 9
10 00000028 9A000000 R      CALLS    FP.ADD, FP_ADD ; line 10
           00
11 0000002D      32     MORE    BLOCK    50         ; line 11
12          0          END      ENTRY          ; line 12
```

```
Tektronix ASM 8086/8088 SYMBOL TABLE
Vxx.xx-xx (xxxx)
```

```
Page 2
dd-mmm-yy/xx:xx:xx
```

```
Section = %MNAOBJ, Aligned to 00000010, Size = EMPTY
```

```
Common Section = FP$ACC, Aligned to 00000010, Size = 00000018
```

```
DESTN-----00000010      S1-----00000000
S2-----00000008
```

```
Section = MAIN, Aligned to 00000010, Size = 0000005F
```

```
ENTRY-----00000000      MORE-----0000002D
```

Unbound Globals

```
FP.ADD-----00000000      FP_ADD-----00000000
```

```
12 Lines Read
12 Lines Processed
0 Errors
```

Now enter the following command to link the mainline object file (**mna.obj**) with the necessary modules from the floating-point library (**fp.lib**).

```
link -l -0 mna.obj fp.lib -o mna.load >mna.lnk1 [linker invocation]
```

Now look at **mna.lnk1**:

```
Tektronix      8086/8088 Linker Vxx.xx-xx (xxxx)      Page   1
For mna.load
```

MODULE AND FILE MAP:

```
LINK FILES:
MAIN.ADD      mna.obj
fp.lib        fp.lib
```

```
Tektronix      8086/8088 Linker Vxx.xx-xx (xxxx)      Page   2
For mna.load
```

MEMORY AND SECTION MAP:

```
NONAME:
      0 -   FFFFF

FP_ADD      0-   185      186 SECTION ALIGNED FP$ADD
MAIN        190-  1EE      5F SECTION ALIGNED MAIN.ADD
FP$ACC      1F0-  207      18 COMMON  ALIGNED MAIN.ADD
```

```
Tektronix      8086/8088 Linker Vxx.xx-xx (xxxx)      Page   3
For mna.load
```

MODULE AND SECTION MAP:

MODULE(S) IN LINK FILE(S):

```
MODULE: FP$ADD

SECTION: FP$ACC      1F0-  207      18 COMMON  ALIGNED

SECTION: FP_ADD      0-   185      186 SECTION ALIGNED
FP.ADD2 ----- 28   FP.ADD ----- 0

MODULE: MAIN.ADD

SECTION: FP$ACC      1F0-  207      18 COMMON  ALIGNED

SECTION: MAIN        190-  1EE      5F SECTION ALIGNED
```

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 4
 For mna.load

GLOBAL SYMBOL LISTING:

ENDREL	208	*****
FP\$ACC	1F0	MAIN.ADD
FP.AD2	28	FP\$ADD
FP.ADD	0	FP\$ADD
FP_ADD	0	FP\$ADD
MAIN	190	MAIN.ADD

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 5
 For mna.load

STATISTICS:

Number of warning errors: 0
 Number of errors: 0

Load file is not relinkable
 Load file is not useable for symbolic debugging

Linking Explanation

The library module containing the floating-point addition routine is automatically linked in with the mainline program. The linker determined that two global symbols (FP.ADD and FP_ADD) had not been given a value by any of the non-library modules. The linker then scanned the library and found that module FP\$ADD provided values for these global symbols. The linker included module FP\$ADD in the load module. This process is illustrated in Fig. 7-1.

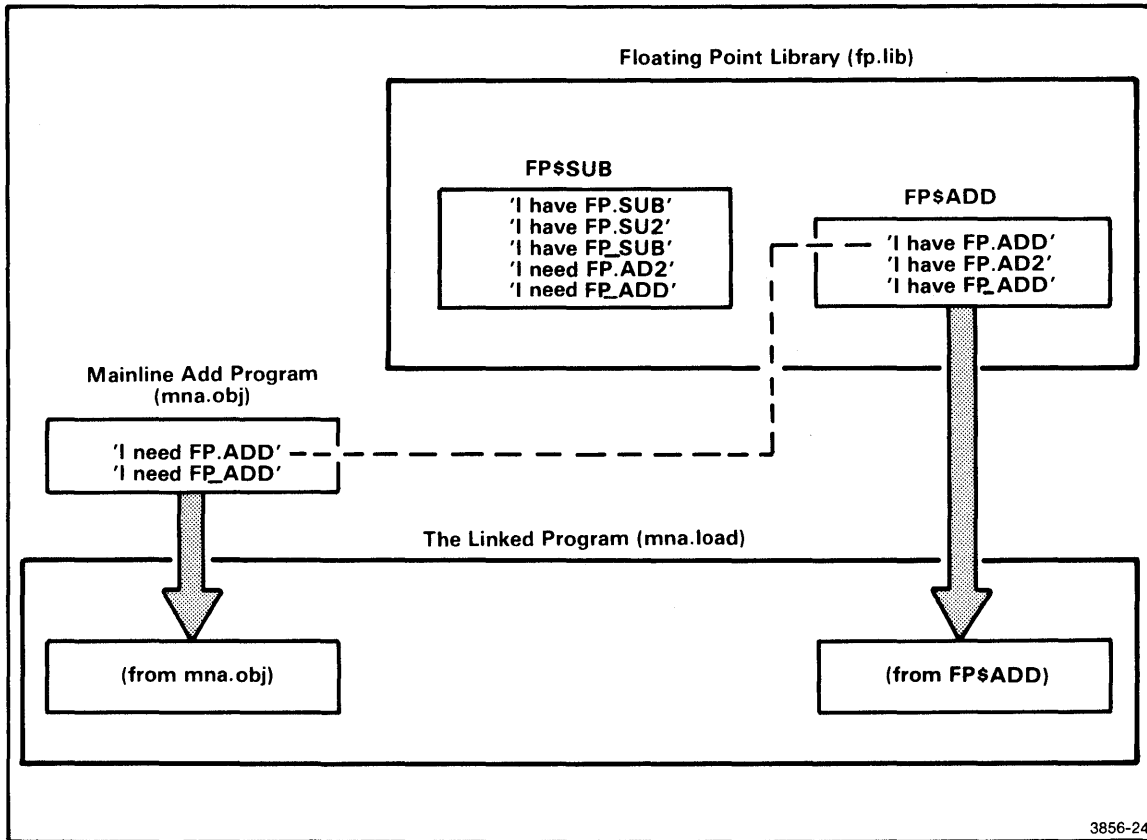


Fig. 7-1. Linking the add program to the library.

In this example, **mna.obj** needs a definition for its unbound global symbols, **FP.ADD** and **FP_ADD**. The linker examines the contents of the library **fp.lib**, and locates module **FP\$ADD**, which provides definitions for **FP.ADD** and **FP_ADD**. Both **mna.obj** and module **FP\$ADD** are then included in the final load file. Since **FP\$SUB** does not provide definitions for any unbound globals, it is not included in the final load file.

Using the SUBTRACT Module from a Program

Let's modify the mainline program to invoke the subtract routine. In this way, we can watch the linker extract one module from the library to resolve a reference in the mainline program, and another module from the library to resolve a reference in the first library module.

The Mainline Subtract Program

```

LIST      LINE(80)           ; line 1
NAME      MAIN.SUB          ; line 2
GLOBAL    FP.SUB, FP_SUB    ; line 3
COMMON    FP$ACC            ; line 4
S1        BLOCK 8           ; line 5
S2        BLOCK 8           ; line 6
DESTN     BLOCK 8           ; line 7
SECTION   MAIN              ; line 8
ENTRY     BLOCK 45          ; line 9
CALLS     FP.SUB, FP_SUB    ; line 10
MORE      BLOCK 35          ; line 11
END       ENTRY             ; line 12
    
```

Explanation of the Mainline Subtract Program

The mainline subtract program is similar to the mainline add program, with the following exceptions:

1. The name of the module in line 2 is MAIN.SUB, not MAIN.ADD.
2. The global symbols requested in lines 3 and 10 are FP.SUB and FP_SUB, not FP.ADD and FP_ADD.
3. The size of the code representations in lines 9 and 11 has been altered, to show the relocatability of the library sections.

Assembling and Linking the Program

You may use your operating system editor to enter the mainline subtract program. Place the mainline subtract program in a file named **mns.asm**. The mainline subtract program can be assembled and linked using the following command entries:

```
asm mns.obj mns.asml mns.asm [assembler invocation]
```

Look at **mns.asml**.

Tektronix ASM 8086/8088
Vxx.xx-xx (xxxx)

Page 1
dd-yyy-yy/xx:xx:xx

```

1          LIST      LINE(80)           ; line 1
2          NAME      MAIN.SUB          ; line 2
3          GLOBAL    FP.SUB, FP_SUB    ; line 3
4          COMMON    FP$ACC            ; line 4
5 00000000      8      S1        BLOCK 8           ; line 5
6 00000008      8      S2        BLOCK 8           ; line 6
7 00000010      8      DESTN     BLOCK 8           ; line 7
8          SECTION   MAIN              ; line 8
9 00000000      2D      ENTRY     BLOCK 45          ; line 9
10 0000002D 9A000000 R  CALLS     FP.SUB, FP_SUB    ; line 10
           00
11 00000032      23      MORE      BLOCK 35          ; line 11
12          0          END       ENTRY             ; line 12
    
```

Tektronix ASM 8086/8088 SYMBOL TABLE
 Vxx.xx-xx (xxxx)

Page 2
 dd-mmm-yy/xx:xx:xx

Section = %MNSOBJ, Aligned to 00000010, Size = EMPTY

Common Section = FP\$ACC, Aligned to 00000010, Size = 00000018

DESTN-----00000010 S1-----00000000
 S2-----00000008

Section = MAIN, Aligned to 00000010, Size = 0000005F

ENTRY-----00000000 MORE-----0000002D

Unbound Globals

FP.SUB-----00000000 FP_SUB-----00000000

12 Lines Read
 12 Lines Processed
 0 Errors

Now link the mainline object file (**mns.obj**) with the necessary object modules from the floating-point library (**fp.lib**) using the following command line:

link -l -o mns.obj fp.lib -o mns.load >mns.lnk1 [linker invocation]

Look at **mns.lnk1**.

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx)
 For mns.load

Page 1

MODULE AND FILE MAP:

LINK FILES:
 MAIN.SUB mns.obj
 fp.lib fp.lib

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 2
 For mns.load

MEMORY AND SECTION MAP:

NONAME:

0 - FFFFF

FP_SUB	0-	8B	8C SECTION ALIGNED	FP\$SUB
MAIN	90-	E4	55 SECTION ALIGNED	MAIN.SUB
FP\$ACC	FO-	107	18 COMMON ALIGNED	MAIN.SUB
FP_ADD	110-	295	186 SECTION ALIGNED	FP\$ADD

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx) Page 3
 For mns.load

MODULE AND SECTION MAP:

MODULE(S) IN LINK FILE(S):

MODULE: FP\$ADD

SECTION: FP\$ACC FO- 107 18 COMMON ALIGNED

SECTION: FP_ADD 110- 295 186 SECTION ALIGNED

FP.AD2 ----- 138 FP.ADD ----- 110

MODULE: FP\$SUB

SECTION: FP\$ACC FO- 107 18 COMMON ALIGNED

SECTION: FP_SUB 0- 8B 8C SECTION ALIGNED

FP.SU2 ----- 46 FP.SUB ----- 0

MODULE: MAIN.SUB

SECTION: FP\$ACC FO- 107 18 COMMON ALIGNED

SECTION: MAIN 90- E4 55 SECTION ALIGNED

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx)
For mns.load

Page 4

GLOBAL SYMBOL LISTING:

ENDREL	296	*****
FP\$ACC	F0	MAIN.SUB
FP.AD2	138	FP\$ADD
FP.ADD	110	FP\$ADD
FP.SU2	46	FP\$SUB
FP.SUB	0	FP\$SUB
FP_ADD	110	FP\$ADD
FP_SUB	0	FP\$SUB
MAIN	90	MAIN.SUB

Tektronix 8086/8088 Linker Vxx.xx-xx (xxxx)
For mns.load

Page 5

STATISTICS:

Number of warning errors: 0
Number of errors: 0

Load file is not relinkable
Load file is not useable for symbolic debugging

Linking Explanation

The mainline subtract program has two unbound global references, FP.SUB and FP_SUB. The library module FP\$SUB has bound global definitions for FP.SUB and FP_SUB. The linker brings module FP\$SUB into the final load module. However, FP\$SUB itself contains references to the unbound global symbols, FP.AD2 and FP_ADD. The definitions for these unbound global symbols are found in the FP\$ADD library module. The linker must include both modules from the library to satisfy all requests for global symbols. This process is illustrated in Fig. 7-2.

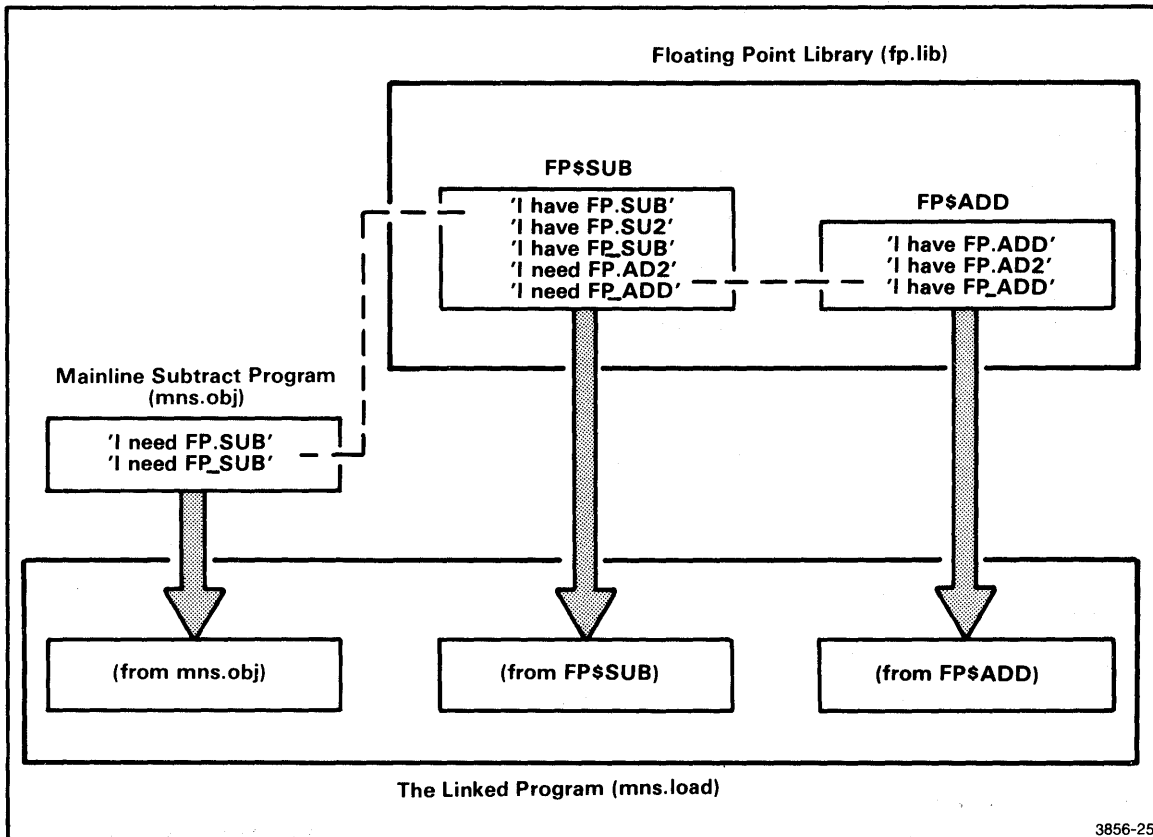


Fig. 7-2. Linking the subtract program to the library.

In this example, *mns.obj* requests definition for the unbound global symbols, *FP.SUB* and *FP.SUB*. The linker scans the library and locates definitions for *FP.SUB* and *FP.SUB* in module *FP\$SUB*. However, *FP\$SUB* itself contains references to the unbound global symbols, *FP.AD2* and *FP_ADD*. The linker continues to scan the library and finds definitions for these symbols in library module *FP\$ADD*. Thus, the final load file contains all three modules (mainline *mns.obj*, and *FP\$SUB* and *FP\$ADD* from the library) linked together.

LINKING OVERLAYS

This example shows how to use the linker to link overlays. Overlays are normally used when there is not enough core memory to contain an entire program. The program is broken up into segments in such a way that there is a minimum of overlaying required. There must be a root segment that does not get overlaid. The root segment calls in the overlay modules. In order for the root to be able to communicate with the overlays, the unbound global references in each must be resolved. In addition, each overlay must be relocated to an address that is just above the root segment. The linker performs both of these functions.

To illustrate this process we will create an overlay version of the floating-point subtract program presented in the previous subsection. The mainline subtract program will call the subtract module as before, but the subtract module will not call the add module. Instead, when the

subtract module is done, it will return control to the mainline subtract program which will overlay the subtract module with the add module and call the add module at the appropriate entry point.

The new source modules look like this:

mna.asm

```

LIST      LINE(80)                ; line 1
NAME      MAIN.SUB                 ; line 2
GLOBAL   FP.SUB, FP_SUB           ; line 3
GLOBAL   FP.AD2, FP_ADD           ; line 4
COMMON   FP$ACC                   ; line 5
S1       BLOCK 8                   ; line 6
S2       BLOCK 8                   ; line 7
DESTN    BLOCK 8                   ; line 8
SECTION  MAIN                     ; line 9
ENTRY    BLOCK 45                  ; line 10
CALLS    FP.SUB, FP_SUB           ; line 11
LOADOVL  BLOCK 40                  ; line 12
CALLS    FP.AD2, FP_ADD           ; line 13
MORE     BLOCK 35                  ; line 14
END      ENTRY                     ; line 15

```

The following changes were made:

- line 4 declares the add module's entry points as unbound globals.
- line 12 represents the block of statements that load the overlay into memory. In the development stage, the loading may be accomplished with a service call. In your final microprocessor-based product, the method used to load the overlays is dependent upon the prototype's operating system.
- line 13 calls the add module.

fps.asm

```

LIST      DBG                      ; line 1
NAME      FP$SUB                   ; line 2
GLOBAL   FP.SUB, FP.SU2           ; line 3
COMMON   FP$ACC                   ; line 4
SORC1    BLOCK 8                   ; line 5
SORC2    BLOCK 8                   ; line 6
DST      BLOCK 8                   ; line 7
SECTION  FP_SUB                   ; line 8
FP.SUB   BLOCK 70                  ; line 9
FP.SU2   BLOCK 30                  ; line 10
END      ; line 11

```

This module no longer calls the add module; thus, the entry points of the add module need not be declared global in this module.

fpa.asm

```

LIST      DBG                      ; line 1
NAME      FP$ADD                   ; line 2
GLOBAL    FP.ADD, FP.AD2          ; line 3
COMMON    FP$ACC                   ; line 4
SRC1      BLOCK 8                  ; line 5
SRC2      BLOCK 8                  ; line 6
DEST      BLOCK 8                  ; line 7
          SECTION FP_ADD           ; line 8
FP.ADD    BLOCK 40                 ; line 9
FP.AD2    BLOCK 350               ; line 10
END                          ; line 11
    
```

There are no changes in the add module.

After entering and assembling these modules, the following linker commands will resolve all global references and relocate the overlay properly.

```

link -r -O mns.obj -o mns.root
    
```

} This command relocates the root so that the linker will know where to place the overlays in subsequent commands.


```

Tektronix Linker Vxx.xx-xx (xxxx)
link:111 (E) Unresolved global reference FP.SUB at 2D
link:111 (E) Unresolved global reference FP.SUB at 2D
link:111 (E) Unresolved global reference FP.SUB at 2E
link:111 (E) Unresolved global reference FP.AD2 at 55
link:111 (E) Unresolved global reference FP_ADD at 55
link:111 (E) Unresolved global reference FP_ADD at 56
Listing file not generated
    
```

These error messages indicate the location of the unresolved global references. These references will be resolved in the last linker invocation in the overlay procedure.

```

link -r -s mns.root -O fps.obj -o fps.ovl
    
```

} This command relocates fps.obj to the first empty space above the root.

```

Tektronix Linker Vxx.xx-xx (xxxx)
link:119 (W) Transfer address undefined
Listing file not generated
    
```

```

link -r -s mns.root -O fpa.obj -o fpa.ovl
    
```

} This command relocates fpa.obj to the first empty space above the root.

```

Tektronix Linker Vxx.xx-xx (xxxx)
link:119 (W) Transfer address undefined
Listing file not generated
    
```

```

link -s fps.ovl -s fpa.ovl -O mns.obj -o mns.load
    
```

} This command resolves all of the unbound global references in the root (mns.obj).

```

Tektronix Linker Vxx.xx-xx (xxxx)
Listing file not generated
    
```

USING THE "@" CONSTRUCT WITHIN A MACRO

This example illustrates the use of the "@" (at) construct in a macro. Each time a macro is invoked, any "@" construct appearing in the macro body is replaced with a unique eight-character hexadecimal value. When this value is appended to a one-to-eight character symbol within the macro body, a unique 9-to-16 character label is created. This construct allows you to use a label symbol within a macro. Even though the macro is invoked more than once, the label symbol is unique for each invocation.

The example shown here is a delay loop that uses the "@" construct for a label symbol. Even though this macro is invoked more than once, DEL1"@" will be unique each time the macro is invoked.

The number of delay loops (1 to OFFFH) is passed to the macro as the single parameter.

This example uses the 8086/8088 instruction set, but similar techniques can be applied to most processors.

Delay Loop Macro

```

                MACRO      DELAY      ; line 1
                PUSH      CX          ; line 2
                MOV       CX, #"1"    ; line 3
DEL1"@"        LOOP      DEL1"@"     ; line 4
                POP       CX          ; line 5
                ENDM          ; line 6

```

Line 1 defines the beginning of the macro and gives it the name DELAY.

Line 2 is an 8086/8088 assembly language instruction that stores the value of the CX register on the stack in order to be able to restore the CX register to its previous value after the delay.

Line 3 moves the value of the parameter (number of delay loops) to the CX register.

Line 4 is an 8086/8088 assembly language instruction that decrements the CX register until it is zero.

Line 5 retrieves the previous value of the CX register.

Line 6 marks the end the of macro definition.

Macro Invocation

```
DELAY 10H ; SHORT DELAY
```

```
DELAY OFFFHH ; LONG DELAY
```

In this example, the first time DELAY is invoked, the number of delay loops is 10H. The second time it is invoked, the number of delay loops is OFFFHH. Each time the macro is invoked, DEL"@ represents a different address.

THE ASSEMBLER INCLUDE DIRECTIVE

This example illustrates some uses of the INCLUDE directive. The INCLUDE directive causes the assembler to process statements from the specified file as though they were a part of your source file.

Frequently used blocks of code and macro definitions may be stored in files. These statements may be included in programs when needed, by simply entering the INCLUDE directive and the filespec of the file. The contents of the file are then assembled into the object module.

This example illustrates four ways in which the INCLUDE directive is typically used: (1) defining constants, (2) defining COMMON sections, (3) defining macros, and (4) providing authorship notices in your listings.

Including Constant Definitions

If you're using the same set of constants for a number of programs, you may store them in a file. You can INCLUDE them in your program, where they'll be processed with your source statements at assembly time. This feature can save you a great deal of time. For example, a file named `cnst.asm` contains the constant definition block listed below.

```
ROWS EQU 20 ; Defines the number of rows
COLS EQU 15 ; Defines the number of columns
```

The main program, which uses this block of constants, is shown below.

```
NAME MAINPRO
INCLUDE 'cnst.asm' ; Constant definitions
MOV BX,ROWS ; Number of rows to BX
MOV CX,COLS ; Number of columns to CX
.
.
TABLE BLOCK ROWS*COLS ; Allocates space for a 300-byte table.
```

When the program MAINPRO is assembled, the constant definitions are included and the program looks like this:

```

NAME      MAINPRO
INCLUDE   'cnst.asm'
ROWS     EQU      20      ; Defines the number of rows
COLS     EQU      15      ; Defines the number of columns
.
.
MOV      BX,ROWS      ; Number of rows to BX
MOV      CX,COLS      ; Number of columns to CX
.
.
TABLE   BLOCK   ROWS*COLS ; Allocates space for a 300-byte table

```

Including COMMON Declarations

A group of COMMON statements is usually used in more than one program. You may store these statements in a file and include them in the various programs that require the same COMMON declarations.

A file named **comm.asm** contains the COMMON declarations for the program MAINPRO.

```

COMMON   CUSTOMER      ; Defines a COMMON section named CUSTOMER
CNAME    BLOCK   30      ; Reserves 30 bytes for CNAME
ADDRESS  BLOCK   30      ; Reserves 30 bytes for ADDRESS
CITY     BLOCK   16      ; Reserves 16 bytes for CITY
STATE    BLOCK   2       ; Reserves 2 bytes for STATE

```

MAINPRO is the program which uses the COMMON declarations from the file **comm.asm**.

```

NAME      MAINPRO
INCLUDE   'comm.asm'    ; Defines the COMMON section
.
.

```

When MAINPRO is assembled, the object module will contain the COMMON declarations as follows:

```

NAME      MAINPRO
INCLUDE   'comm.asm'
COMMON   CUSTOMER      ; Defines a COMMON section named CUSTOMER
CNAME    BLOCK   30      ; Reserves 30 bytes for CNAME
ADDRESS  BLOCK   30      ; Reserves 30 bytes for ADDRESS
CITY     BLOCK   16      ; Reserves 16 bytes for CITY
STATE    BLOCK   2       ; Reserves 2 bytes for STATE
.
.

```

Including Macro Definitions

A frequently used macro may be defined in a file and included in your program with the INCLUDE directive.

In this example, file **mabc.asm** contains the macro definition to be included in the program MAINPRO. The BYTE directive has a parameter which will be given when the macro is invoked.

```
MACRO   ABC           ; Beginning of macro definition
BYTE    "1"          ; Generate a byte of the first parameter
WORD    40            ; Generate a word containing the value 40
ENDM     ; End of macro definition
```

MAINPRO, the program which includes the file **mabc.asm** for its macro definition, is listed below.

```
NAME    MAINPRO
INCLUDE 'mabc.asm' ; INCLUDEs the definition for macro ABC
.
.
ABC     5           ; Invokes ABC with a parameter of 5
.
.
ABC     15          ; Invokes ABC with a parameter of 15
.
.
```

Once the macro ABC has been included in MAINPRO, each invocation of macro ABC causes the macro to be expanded at assembly time.

Authorship and Copyright Notices for Listings

The INCLUDE directive may be used to print authorship and copyright notices on program listings.

Let's say that a file named `cpyr.asm` contains the heading information that you wish to place on each program listing.

```

;*****
;*
;*          COPYRIGHT (C) 1981 BY
;*
;*****
;*
;*          *****
;*          *          *          *
;*          *          *          *
;*          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*          *          *          *          *          *          *          *          *          *
;*****
;*****
;*
;*          TEKTRONIX, INCORPORATED, BEAVERTON, OREGON 97077
;*
;*          ALL RIGHTS RESERVED
;*
;*****
;*****
;*
;*          AUTHOR: KEN DEDATE
;*
;*****
;*****

```

Using a single statement, INCLUDE 'cpyr.asm', your assembler listing will take the following format.

```

NAME      MAINPRO
INCLUDE  'cpyr.asm'
;*****
;*
;*          COPYRIGHT (C) 1981 BY
;*
;*****
;*
;*          *****
;*          *          *          *
;*          *  ***   *  *  ****  ****  ****  ****  *  *  *
;*          * * *   * * *   *   *   *   *   *   *   *   *   *   *
;*          * ****   ***   *   *   *   *   *   *   *   *   *   *   *
;*          * *          * *   *   *   *   *   *   *   *   *   *   *
;*          * *****   *   *** *   **** *   * * *   *
;*
;*          COMMITTED TO EXCELLENCE
;*
;*****
;*
;*          TEKTRONIX, INCORPORATED, BEAVERTON, OREGON 97077
;*
;*          ALL RIGHTS RESERVED
;*
;*****
;*
;*          AUTHOR: KEN DEDATE
;*
;*****

```

Section 8

HOST SPECIFICS

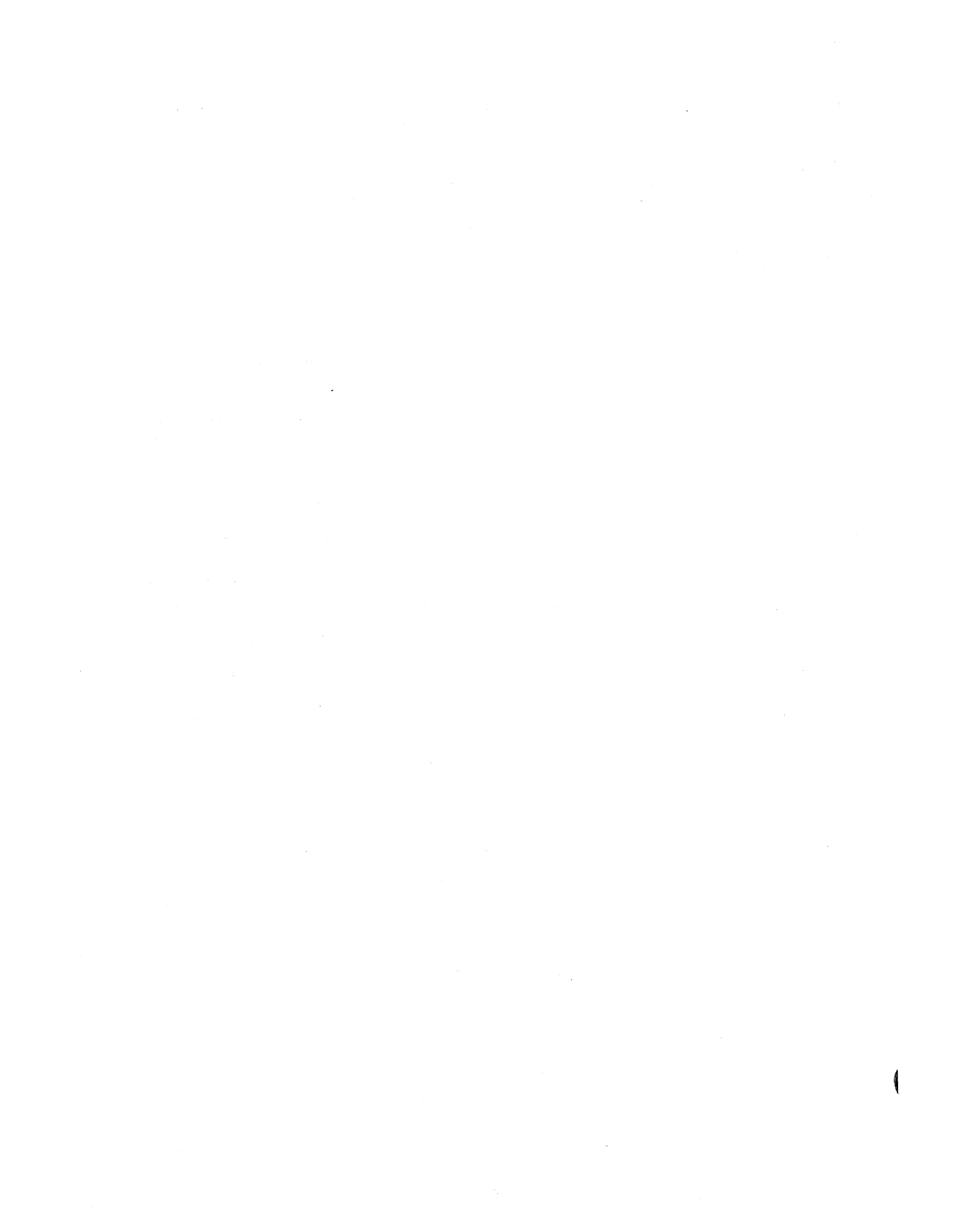
System-specific information is contained in the Host Specifics supplement for each host system. Each supplement is designed as a subsection to this manual.

The Host Specifics supplements are numbered as if they were separate sections of the manual. For example, the 8550 supplement is labeled "Section 8A", and the first illustration is numbered "Fig. 8A-1". Similarly, other supplements are labeled Section 8B, 8C, etc. Figures, pages, and tables are numbered accordingly.

Each subsection presents the following information:

- **Operating System Features.** Describes the filespecs, filenames, and other operating system features that are related to assembler operation.
- **Installation.** Shows you how to install the software for your specific assembler.
- **Assembler Invocation.** Describes how to invoke the assembler with your operating system.
- **Linker Invocation.** Describes how to invoke the linker with your operating system.
- **LibGen Invocation.** Describes how to invoke the library generator with your operating system.
- **Demonstration Run.** Shows how to enter and assemble a simple program and subroutine, and how to prepare the resulting object modules for loading into memory.

Each supplement has its own table of contents.



Section 10

TECHNICAL NOTES

NOTE 1: DIFFERENCES BETWEEN THE A SERIES AND B SERIES ASSEMBLERS

This technical note is intended for users of the TEKTRONIX 8300AXX (A series) and the 8002A assemblers. In this note, all references to "A series assembler" apply equally to the 8002A assembler.

This note describes the differences between the A series assembler and the TEKTRONIX 8500 MDL B series assembler, as described in this manual.

This note lists only the differences between the microprocessor-independent parts of the two assemblers. Differences between the microprocessor-dependent parts are under the heading "Irregularities" in the corresponding Assembler Specifics section of this manual.

String Substitution Delimiter. In the A series assembler, the delimiter denoting the insertion of strings, macro arguments, and current section names is a single quote ('). In the B series assembler, this delimiter is a double quote (").

String Delimiter. In the A series assembler, the delimiter surrounding string literals is a double quote ("). In the B series assembler, this delimiter is a single quote (').

Symbol Length. The A series assembler uses only the first 8 characters of a symbol, and discards the rest of the symbol. (AVERYLONGSYMBOL is equivalent to AVERYLONGNAME.) The B series assembler uses the first 16 characters of a symbol. (AVERYLONGSYMBOL is different from AVERYLONGNAME.)

Value Size. The A series assembler performs all arithmetic with 16-bit integers; values are limited to the range of -32768 to 32767 (or 0 to 65535 for addresses). The B series assembler performs arithmetic with 32-bit integers, yielding a range of values from -2147483648 to 2147483647 (or 0 to 4294967295 for addresses).

Default String Length. In the A series assembler, all strings declared with the STRING directive have a default length of 8 characters. In the B series assembler, the default string length is 16 characters.

Macro and REPEAT Block Storage. The A series assembler stores macro and REPEAT block text in core. The B series assembler rereads (from the source file) the macro or REPEAT block text upon each expansion.

Page Size. In the A series assembler, the listing page size is 55 lines per page, at either 72 or 132 characters per line (selected by the LIST TRM directive). In the B series assembler, these limits are completely user-selectable (see the LIST directive).

Conversion of Numbers to Strings. In the A series assembler, when a string value is needed and a numeric value is supplied, the assembler converts the numeric value into a fixed-format 6-character string padded with leading zeros. In the B series assembler, the conversion creates a variable-length literal decimal representation, with no leading-zero padding.

LONG, ADDRESS, and FLOAT Directives. The LONG, ADDRESS, and FLOAT data storage directives are available only in the B series assembler. (See the Assembler Directives section of this manual for further information.)

ASCII Directive. In the A series assembler, the ASCII directive accepts only a series of string expressions. In the B series assembler, the ASCII directive accepts both string and numeric expressions.

BITS and STRINGOF Functions. The BITS and STRINGOF functions are present only in the B series assembler. (See the Assembler section of this manual for further information.)

CLASS and ALIGN Attributes. The CLASS and ALIGN attributes on SECTIONs, COMMONs, and RESERVEs are present only in the B series assembler. (See the Linker section of this manual for further information.)

Nested INCLUDE Files. The B series assembler allows an INCLUDE file to contain an INCLUDE directive. The A series assembler does not allow nesting of INCLUDE files.

ELSEIF Directive. The B series assembler provides the ELSEIF directive to allow more flexibility in composing alternate source code blocks. The A series assembler does not provide this feature.

TIMES Option and EXITR Directive. In the B series assembler, the TIMES option (on the REPEAT directive) and the EXITR directive allow you to have more precise control in repeated source text. The A series assembler does not provide these features.

Repeat Blocks Outside of Macros. The A series assembler requires you to place repeated source text inside macro definition blocks. The B series assembler allows you to place this repeated source text inside or outside macro definition blocks.

Nested Brackets in Macro Parameter Lists. The B series assembler allows brackets [] to be nested (similar to parentheses in a numeric expression). The A series assembler does not allow nesting; the first right bracket is paired with the first left bracket, regardless of any additional left brackets encountered.

No Labels on Some Directives. The A series assembler allows a label on nearly every directive, including those that generate no object code. The B series assembler does not allow labels on several directives. See the Label Field discussion in The Assembler section of this manual for a list of directives that may not have labels.

Label on a Line by Itself. The B series assembler allows a label to stand by itself on a separate line (equivalent to "label EQU \$"). The A series assembler does not allow this.

Cross Reference Listing. The B series assembler provides a cross reference listing of symbols. The A series assembler does not provide this feature.

Reserved Words. The words ADDRESS, ALIGN, BITS, CLASS, ELSEIF, EXITR, FLOAT, LONG, STRINGOF, TIMES, and XREF are reserved in the B series assembler, but not in the A series assembler.

Line Number Count. The B series assembler increments the line number counter for each line processed, including those that are assembled because of macro expansion or repetition. The A series assembler increments the line number counter only for each source line read from the source file or INCLUDE file.

Current Listing Flags. The B series assembler saves the current listing flags (set by the LIST and NOLIST directives) when it begins to process a macro invocation or an INCLUDE file, and restores the flags at the end of processing the macro or INCLUDE file. The A series assembler performs this save-and-restore operation only for macro expansion.

Object Code Format. The object code formats of the A series assembler and B series assembler are totally incompatible.

Section 11 TABLES

**Table
No.**

11-1	Source Module Character Set	11-1
11-2	Assembler Directives	11-3
11-3	ASCII-Binary-Hexadecimal-Decimal Conversion	11-5
11-4	Decimal-Hexadecimal-Binary Equivalents	11-6
11-5	Hexadecimal Addition.....	11-7
11-6	Hexadecimal Multiplication.....	11-7

Section 11

TABLES

Table 11-1
Source Module Character Set

Symbols	Definition
A..Z	letters used in symbols, strings, functions, hexadecimal constants (A-F only), and other constants as the radix (H, O, Q, and B); lowercase characters (other than in strings and comments) are interpreted as the corresponding uppercase characters
0..9	numbers used in symbols and constants
\$	used in symbols, and to represent assembler location counter contents
.	used in symbols and floating point constants
_(underscore)	used in symbols
;	precedes a comment
,(comma)	delimiter for operand items
'(single quote)	string delimiter
:	string concatenation operator
"	string substitution delimiter
#	total number of arguments passed to current macro expansion
[]	treat everything within brackets as a single macro argument
@	provide unique labels for each macro expansion
%	replaced by name of current section via string substitution
*	binary arithmetic operation, multiplication
/	binary arithmetic operation, division
+	unary or binary arithmetic operator, addition
-	unary or binary arithmetic operator, subtraction

Table 11-1 (cont.)

Symbols	Definition
()	override precedence of operators
\	unary logical operator, NOT
&	binary logical operator, AND
!	binary logical operator, inclusive OR
!!	binary logical operator, exclusive OR
SPACE	field delimiter
TAB	field delimiter
RETURN	field and line delimiter
^	allow following special character to have literal meaning
^^	allow the second up-arrow character to have literal meaning
=	relational operator, equal
<>	relational operator, not equal
>	relational operator, greater than
<	relational operator, less than
>=	relational operator, greater than or equal
<=	relational operator, less than or equal

**Table 11-2
Assembler Directives**

Directive	Operation
ADDRESS	initializes memory with data in address format
ASCII	initializes memory with data in ASCII format
BLOCK	reserves a data block
BYTE	initializes memory with 8-bit value(s)
COMMON	declares program section, assigns name, defines type to be common
ELSE	turns on assembly if it has been turned off by an IF statement
ELSEIF	turns on assembly if it has been turned off by an IF statement and the associated expression is evaluated true (nonzero)
END	marks the end of an assembly source module
ENDIF	marks the end of an IF block
ENDM	marks the end of a macro
ENDR	marks the end of a REPEAT block
EQU	assigns a value to a symbol(s)
EXITM	terminates macro expansion before the ENDM
EXITR	terminates repeat process before the ENDR
FLOAT	initializes memory with data in floating point format
GLOBAL	declares global symbol
IF	turns off assembly if the associated expression is evaluated false (zero or undefined)
INCLUDE	inserts text from another source file
LIST	turns on assembler listing options
LONG	initializes memory with 32-bit value(s)
MACRO	defines the beginning of a macro source block
NAME	declares object module name

Table 11-2 (cont.)

Directive	Operation
NOLIST	turns off assembler listing options
ORG	assigns an address to the assembler location counter
PAGE	advances listing to a new page
REPEAT	causes source statements to be assembled repeatedly
RESERVE	reserves memory space and defines a section
RESUME	resumes the definition of a section
SECTION	declares a program section, assigns name
SET	assigns or reassigns a value to a variable
SPACE	inserts blank line(s) in listing
STITLE	creates a listing page subtitle
STRING	declares a string variable
TITLE	creates a listing page title
WARNING	displays a warning message
WORD	initializes memory with 16-bit value(s)

Table 11-3
ASCII-Binary-Hexadecimal-Decimal Conversion

B T S				07 06 05	04 03 02 01	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
				CONTROL				SYMBOLS				UPPERCASE				LOWERCASE						
0	0	0	0	NUL	10	16	20	32	30	40	40	04	50	0F	60	06	70	112				
1	0	0	0	SOH	11	17	21	33	31	40	41	05	51	01	61	07	71	113				
0	0	1	0	STX	2	18	22	34	32	40	42	06	52	02	62	08	72	114				
0	0	1	1	ETX	3	19	23	35	33	40	43	07	53	03	63	09	73	115				
0	1	0	0	EOT	4	20	24	36	34	40	44	08	54	04	64	0A	74	116				
0	1	0	1	ENQ	5	21	25	37	35	40	45	09	55	05	65	0B	75	117				
0	1	1	0	ACK	6	22	26	38	36	40	46	0A	56	06	66	0C	76	118				
0	1	1	1	BEL BELL	7	23	27	39	37	40	47	0B	57	07	67	0D	77	119				
1	0	0	0	BS BACK SPACE	8	24	28	40	38	40	48	0C	58	08	68	0E	78	120				
1	0	0	1	HT	9	25	29	41	39	40	49	0D	59	09	69	0F	79	121				
1	0	1	0	LF	10	26	2A	42	3A	40	4A	0E	5A	0A	6A	10	7A	122				
1	0	1	1	VT	11	27	2B	43	3B	40	4B	0F	5B	0B	6B	11	7B	123				
1	1	0	0	FF	12	28	2C	44	3C	40	4C	10	5C	0C	6C	12	7C	124				
1	1	0	1	CR RETURN	13	29	2D	45	3D	40	4D	11	5D	0D	6D	13	7D	125				
1	1	1	0	SO	14	30	2E	46	3E	40	4E	12	5E	0E	6E	14	7E	126				
1	1	1	1	SI	15	31	2F	47	3F	40	4F	13	5F	0F	6F	15	7F	127			DEL RUBOUT	

Table 11-4
Decimal-Hexadecimal-Binary Equivalents

Decimal	Hexadecimal	Binary 8-bit Code	Decimal	Hexadecimal	Binary 8-bit Code	Decimal	Hexadecimal	Binary 8-bit Code	Decimal	Hexadecimal	Binary 8-bit Code
0	00	0000 0000	64	40	0100 0000	128	80	1000 0000	192	C0	1100 0000
1	01	0000 0001	65	41	0100 0001	129	81	1000 0001	193	C1	1100 0001
2	02	0000 0010	66	42	0100 0010	130	82	1000 0010	194	C2	1100 0010
3	03	0000 0011	67	43	0100 0011	131	83	1000 0011	195	C3	1100 0011
4	04	0000 0100	68	44	0100 0100	132	84	1000 0100	196	C4	1100 0100
5	05	0000 0101	69	45	0100 0101	133	85	1000 0101	197	C5	1100 0101
6	06	0000 0110	70	46	0100 0110	134	86	1000 0110	198	C6	1100 0110
7	07	0000 0111	71	47	0100 0111	135	87	1000 0111	199	C7	1100 0111
8	08	0000 1000	72	48	0100 1000	136	88	1000 1000	200	C8	1100 1000
9	09	0000 1001	73	49	0100 1001	137	89	1000 1001	201	C9	1100 1001
10	0A	0000 1010	74	4A	0100 1010	138	8A	1000 1010	202	CA	1100 1010
11	0B	0000 1011	75	4B	0100 1011	139	8B	1000 1011	203	CB	1100 1011
12	0C	0000 1100	76	4C	0100 1100	140	8C	1000 1100	204	CC	1100 1100
13	0D	0000 1101	77	4D	0100 1101	141	8D	1000 1101	205	CD	1100 1101
14	0E	0000 1110	78	4E	0100 1110	142	8E	1000 1110	206	CE	1100 1110
15	0F	0000 1111	79	4F	0100 1111	143	8F	1000 1111	207	CF	1100 1111
16	10	0001 0000	80	50	0101 0000	144	90	1001 0000	208	D0	1101 0000
17	11	0001 0001	81	51	0101 0001	145	91	1001 0001	209	D1	1101 0001
18	12	0001 0010	82	52	0101 0010	146	92	1001 0010	210	D2	1101 0010
19	13	0001 0011	83	53	0101 0011	147	93	1001 0011	211	D3	1101 0011
20	14	0001 0100	84	54	0101 0100	148	94	1001 0100	212	D4	1101 0100
21	15	0001 0101	85	55	0101 0101	149	95	1001 0101	213	D5	1101 0101
22	16	0001 0110	86	56	0101 0110	150	96	1001 0110	214	D6	1101 0110
23	17	0001 0111	87	57	0101 0111	151	97	1001 0111	215	D7	1101 0111
24	18	0001 1000	88	58	0101 1000	152	98	1001 1000	216	D8	1101 1000
25	19	0001 1001	89	59	0101 1001	153	99	1001 1001	217	D9	1101 1001
26	1A	0001 1010	90	5A	0101 1010	154	9A	1001 1010	218	DA	1101 1010
27	1B	0001 1011	91	5B	0101 1011	155	9B	1001 1011	219	DB	1101 1011
28	1C	0001 1100	92	5C	0101 1100	156	9C	1001 1100	220	DC	1101 1100
29	1D	0001 1101	93	5D	0101 1101	157	9D	1001 1101	221	DD	1101 1101
30	1E	0001 1110	94	5E	0101 1110	158	9E	1001 1110	222	DE	1101 1110
31	1F	0001 1111	95	5F	0101 1111	159	9F	1001 1111	223	DF	1101 1111
32	20	0010 0000	96	60	0110 0000	160	A0	1010 0000	224	E0	1110 0000
33	21	0010 0001	97	61	0110 0001	161	A1	1010 0001	225	E1	1110 0001
34	22	0010 0010	98	62	0110 0010	162	A2	1010 0010	226	E2	1110 0010
35	23	0010 0011	99	63	0110 0011	163	A3	1010 0011	227	E3	1110 0011
36	24	0010 0100	100	64	0110 0100	164	A4	1010 0100	228	E4	1110 0100
37	25	0010 0101	101	65	0110 0101	165	A5	1010 0101	229	E5	1110 0101
38	26	0010 0110	102	66	0110 0110	166	A6	1010 0110	230	E6	1110 0110
39	27	0010 0111	103	67	0110 0111	167	A7	1010 0111	231	E7	1110 0111
40	28	0010 1000	104	68	0110 1000	168	A8	1010 1000	232	E8	1110 1000
41	29	0010 1001	105	69	0110 1001	169	A9	1010 1001	233	E9	1110 1001
42	2A	0010 1010	106	6A	0110 1010	170	AA	1010 1010	234	EA	1110 1010
43	2B	0010 1011	107	6B	0110 1011	171	AB	1010 1011	235	EB	1110 1011
44	2C	0010 1100	108	6C	0110 1100	172	AC	1010 1100	236	EC	1110 1100
45	2D	0010 1101	109	6D	0110 1101	173	AD	1010 1101	237	ED	1110 1101
46	2E	0010 1110	110	6E	0110 1110	174	AE	1010 1110	238	EE	1110 1110
47	2F	0010 1111	111	6F	0110 1111	175	AF	1010 1111	239	EF	1110 1111
48	30	0011 0000	112	70	0111 0000	176	B0	1011 0000	240	F0	1111 0000
49	31	0011 0001	113	71	0111 0001	177	B1	1011 0001	241	F1	1111 0001
50	32	0011 0010	114	72	0111 0010	178	B2	1011 0010	242	F2	1111 0010
51	33	0011 0011	115	73	0111 0011	179	B3	1011 0011	243	F3	1111 0011
52	34	0011 0100	116	74	0111 0100	180	B4	1011 0100	244	F4	1111 0100
53	35	0011 0101	117	75	0111 0101	181	B5	1011 0101	245	F5	1111 0101
54	36	0011 0110	118	76	0111 0110	182	B6	1011 0110	246	F6	1111 0110
55	37	0011 0111	119	77	0111 0111	183	B7	1011 0111	247	F7	1111 0111
56	38	0011 1000	120	78	0111 1000	184	B8	1011 1000	248	F8	1111 1000
57	39	0011 1001	121	79	0111 1001	185	B9	1011 1001	249	F9	1111 1001
58	3A	0011 1010	122	7A	0111 1010	186	BA	1011 1010	250	FA	1111 1010
59	3B	0011 1011	123	7B	0111 1011	187	BB	1011 1011	251	FB	1111 1011
60	3C	0011 1100	124	7C	0111 1100	188	BC	1011 1100	252	FC	1111 1100
61	3D	0011 1101	125	7D	0111 1101	189	BD	1011 1101	253	FD	1111 1101
62	3E	0011 1110	126	7E	0111 1110	190	BE	1011 1110	254	FE	1111 1110
63	3F	0011 1111	127	7F	0111 1111	191	BF	1011 1111	255	FF	1111 1111

**Table 11-5
Hexadecimal Addition**

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Example $0F + 8 = 17$ (hexadecimal).

**Table 11-6
Hexadecimal Multiplication**

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Example $9 * 8 = 48$ (hexadecimal).

Section 12 ERROR MESSAGES

	Page
Introduction	12-1
Assembler Errors	12-1
Linker Errors	12-10
LibGen Errors	12-15



Section 12

ERROR MESSAGES

INTRODUCTION

This section describes three categories of error messages: those originating from the assembler, those originating from the linker, and those originating from the library generator. Each group of error messages is separately listed in numeric order. Each error message is followed by a description of the problem.

Each error message appears in the following format:

```
*** source:nnn (severity-code) message-text
```

The three asterisks ******* are present in the listing, but do not appear on the terminal.

source indicates the source of the error: ASM, LINK, or LIBGEN. However, if the assembler, linker, or library generator is invoked using another name, that name will be substituted.

nnn is the error number.

The **severity-code** indicates the severity of the error, and may be any one of the following:

- W—warning
- E—error
- S—serious
- F—fatal

The **message-text** briefly describes the nature of the error.

For example:

```
*** ASM: 4 (E) Missing closing ' or "
```

This example shows that the error message was issued by the assembler (ASM), that it is a non-fatal error (E), and that a quotation mark was omitted.

Assembler Errors

Assembler errors in the range 0–199 are described in this section. Assembler errors in the range 200–299 are microprocessor-dependent. Refer to the Assembler Specifics section of this manual for a list of microprocessor-dependent errors.

All assembler errors are listed on the terminal **and** in the listing file.

The **severity-code** may be one of the following:

W (warning)—The error should have no effect on assembled program's execution.

E (error)—The assembled program probably won't execute properly.

F (fatal)—The error directly affects the assembler's execution. The assembler closes all open channels and returns to the operating system.

In some error messages, a symbol in single quotes is used to represent a filename, digit-string, or other information that will be included in the actual message. For example:

ASM: 44 (E) Illegal character in 'digit-string'

This symbol is replaced with a string
of digits when the message is generated.

- ASM: 0 (F) Internal assembler error 'number'.** Serious problems have been detected. Contact your Tektronix service representative if this error persists.
- ASM: 1 (W)** This is a user-defined error. See the **WARNING** directive in the **Assembler Directives** section of this manual for more information.
- ASM: 2 (E) Nonprinting character; ignored.** A character that is not in the assembler character set has been used outside of single quotes. Refer to the 'Source Module Character Set' in the **Tables** section of this manual for a list of valid characters.
- ASM: 3 (E) Label symbol is reserved; line ignored.** The symbol in the label field is a reserved word. See the **Assembler Specifics** section of this manual for a list of reserved words.
- ASM: 4 (E) Missing closing ' or ".** An opening quotation mark is not matched by a closing quotation mark.
- ASM: 5 (W) Label not allowed; accepted.** A symbol is not allowed in the label field of this directive. The symbol is assigned the current value of the location counter. See the **Assembler Directives** section of this manual for a list of directives that may have labels.
- ASM: 6 (E) Label phase error.** There is a difference between pass 1 and pass 2 in the value of a label; or, a label is encountered on pass 2 that was not encountered on pass 1. This could be the result of string substitution and/or conditional assembly.
- ASM: 7 (E) Symbol 'symbol' previously defined; ignored.** This error occurs if the specified symbol has already been defined. The symbol retains the value given the first time it was encountered.
- ASM: 8 (E) String too long for conversion; truncated.** The length of a string used as a number exceeds four characters.

-
- ASM: 9 (E) Illegal escape sequence.** An escape (^) is the last character before the end-of-line.
- ASM: 10 (E) Source line too long; truncated.** The source line is too long. Maximum length is 127 characters plus RETURN (maximum of 131 after text substitution).
- ASM: 11 (E) Bad substitution construct; line ignored.** The substitution construct (symbol inside double quotes) is null or erroneous. This could be caused by misspelling the symbol within quotes.
- ASM: 12 (E) Negative parameter pointer.** A substitution construct or a STRINGOF function contains a negative number.
- ASM: 13 (E) No macro active.** Either a STRINGOF function, a number within double quotes, a #, or a @ is encountered outside of a macro.
- ASM: 14 (E) Conditional assembly nesting error.** There is an overlap of IF and/or REPEAT blocks or an ENDIF directive is encountered when no IF block is active.
- ASM: 15 (E) Misplaced ELSE; ignored.** Either an ELSE directive is outside an IF-ENDIF block, or more than one ELSE directive is within an IF-ENDIF block.
- ASM: 16 (E) Misplaced ELSEIF; ignored.** An ELSEIF directive is outside an IF-ENDIF block.
- ASM: 17 (E) BYTE value too large; truncated.** The value entered exceeds one byte. The allowable range is -128 to +255. The value is truncated to fall within this range.
- ASM: 18 (E) WORD value too large; truncated.** The value entered exceeds one word. The allowable range is -32768 to +65535. The value is truncated to fall within this range.
- ASM: 19 (E) BITS selection too large; truncated.** The size of the receiving field is smaller than the bit string specified and the bit source (first parameter in BITS function) is relocatable.
- ASM: 20 (W) Scalar value required.** An address value has been used where a scalar is required.
- ASM: 21 (E) Undefined operand.** No value has yet been assigned to a symbol used in an expression.
- ASM: 22 (E) Negative string length.** A declaration in the STRING directive specifies a maximum length that is negative.
- ASM: 23 (E) Symbol value phase error.** There is a difference between pass 1 and pass 2 in the value of a symbol. This message may occur if a SET directive has a forward reference in conjunction with conditional assembly.
- ASM: 24 (E) Illegal operation on unbound global.** An unbound global has been encountered in the operand field of a SET directive.

-
- ASM: 25 (E) String value truncated.** More characters are assigned to a string than were specified in the STRING directive that defined it.
- ASM: 26 (E) Illegal use of class name.** The only legal use of a class name in an assembler source program is to assign its value with a SECTION, COMMON, or RESERVE directive.
- ASM: 27 (W) Cannot open error message text file.** The error message text file is missing or otherwise inaccessible.
- ASM: 28 (E) String value required.** A numeric value appears where a string is required. Concatenation, SEG or NCHR functions, and TITLE or STITLE directives all require string operands.
- ASM: 29 (E) Expression too complex; line ignored.** Parentheses in expression are nested more than 3 levels deep.
- ASM: 30 (E) Illegal address comparison.** An attempt has been made to subtract or compare addresses based on different sections.
- ASM: 31 (E) Division by zero.** A division or a MOD operation attempted to use zero as a divisor.
- ASM: 32 (E) Illegal scalar-address.** An attempt has been made to subtract an address from a scalar value.
- ASM: 33 (E) Illegal address + address.** An attempt has been made to add two addresses.
- ASM: 34 (E) BITS index out of range.** The second parameter in the BITS function is negative or greater than 31.
- ASM: 35 (E) BITS length out of range.** The sum of the second and third parameters is greater than 32.
- ASM: 36 (E) Illegal operation on complex result.** Arithmetic is being performed on a value that will not be evaluated until link time (relocatable).
- ASM: 37 (E) Cannot close file 'filename'.** The assembler is unable to close the specified file.
- ASM: 38 (E) Null expression for INCLUDE file name.** Either there is no filename in the operand field of an INCLUDE directive or the string variable used contains the null string.
- ASM: 39 (E) Cannot open file 'filename'.** The assembler is unable to open the specified file. Make sure that the filename is spelled correctly and is accessible by you.
- ASM: 40 (E) File 'filename' read error.** The assembler is unable to read the specified file. Make sure that the filename is spelled correctly and is readable by you. This message may result from a hardware error.

-
- ASM: 41 (E) Illegal use of an unbound global in a SET or EQU.** An attempt has been made to take the difference of two unbound globals or to apply the BITS function to an unbound global.
- ASM: 42 (E) Invalid line size; default used.** The expression in the LINE option of the LIST directive is less than 1 or greater than 132. The line size is set to its default value. (72 characters if the output device is the system terminal; 132 otherwise)
- ASM: 43 (E) Invalid page size; default used.** The expression in the PAGE option of the LIST directive is less than 4 or undefined. The page size is set to its default value. (65536)
- ASM: 44 (E) Illegal character in 'digit-string'; ignored.** An illegal digit or radix has been used. See The Assembler section of this manual for information about numeric constant notation.
- ASM: 45 (E) Subtitle truncated.** The expression in the operand field of the STITLE directive exceeds 35 characters. The excess characters are ignored.
- ASM: 46 (E) Title truncated.** The expression in the operand field of the TITLE directive exceeds 31 characters. The excess characters are ignored.
- ASM: 47 (E) ENDM without matching MACRO; ignored.** An ENDM directive is encountered outside a macro definition.
- ASM: 48 (E) EXITM outside macro expansion; ignored.** An EXITM directive is encountered outside a macro definition.
- ASM: 49 (E) Macro nesting error.** A macro definition overlaps with an IF block or another macro.
- ASM: 50 (E) Unmatched [.** An opening bracket is not matched by a closing bracket.
- ASM: 51 (E) Garbage follows closing].** The information following a bracketed macro parameter has been ignored. For example, [BC]DE results in a parameter of BC (and generates this error message). Refer to the Macros section of this manual for further information on macro parameter conventions.
- ASM: 52 (E) End of file encountered during macro definition.** The end-of-file or an END directive is encountered during a macro definition. This message may result from omitting an ENDM directive.
- ASM: 53 (E) MACRO encountered during macro definition or expansion.** A MACRO directive occurs within a macro definition block. The MACRO directive is ignored.
- ASM: 54 (E) Assembly too complex.** The nesting complexity of IF blocks, REPEAT blocks, macro expansions, and/or INCLUDE files, in conjunction with the number of symbols, exceeds the capacity of the assembler.

- ASM: 55 (E) Syntax error: 'illegal-expression'.** A statement does not conform to the required syntax. Refer to the Assembler Directives section of this manual for the correct syntax.
- ASM: 56 (E) Section phase error.** The specified section relocation option or the length of a section differs between pass 1 and pass 2. This error may occur if a SET directive has a forward reference in conjunction with conditional assembly.
- ASM: 57 (E) Object module name already defined; ignored.** The NAME directive has been used more than once.
- ASM: 58 (E) Invalid conditional expression.** The conditional expression in the REPEAT directive does not yield a scalar.
- ASM: 59 (E) Invalid iteration expression.** The iteration expression in the REPEAT directive is not a scalar.
- ASM: 60 (E) Negative iteration count.** The iteration expression in the REPEAT directive is negative.
- ASM: 61 (E) Iteration count exceeded.** An attempt has been made to assemble a REPEAT block more than the number of times specified in the second parameter of the REPEAT directive. If that parameter is not specified, the error message is displayed after 256 repeat cycles are completed.
- ASM: 62 (E) ENDR without matching REPEAT; ignored.** An ENDR directive is encountered outside a REPEAT block.
- ASM: 63 (E) REPEAT-ENDR improperly nested.** A REPEAT block overlaps with an IF block, macro expansion or INCLUDE file.
- ASM: 64 (E) EXITR outside REPEAT-ENDR; ignored.** An EXITR directive is encountered outside a REPEAT block.
- ASM: 65 (E) EXITR improperly nested.** An EXITR is encountered, in a macro or an INCLUDE file, that matches a REPEAT outside the macro or INCLUDE file.
- ASM: 66 (E) ENDR encountered before end of REPEAT block; ignored.** Text substitution and/or conditional assembly has placed an ENDR directive within a REPEAT block.
- ASM: 67 (E) Undefined BLOCK size.** The operand of a BLOCK directive is either undefined or is a forward reference. This error may occur if a misspelled or undefined symbol appears in a BLOCK directive, or if these directives reference a symbol that has not yet been assigned a value.
- ASM: 68 (E) Negative BLOCK size.** The operand of the BLOCK directive is negative.
- ASM: 69 (E) Invalid alignment expression.** The alignment expression used is not a positive scalar.

-
- ASM: 70 (E) Too many global symbols.** The number of global symbols exceeded 253. This number includes all section, common, reserve, class, and global names. The current global declaration is not accepted by the assembler.
- ASM: 71 (E) END encountered in INCLUDE file; ignored.** An END directive is present within an INCLUDE file. The entire INCLUDE file is included.
- ASM: 72 (E) Invalid transfer address expression.** The label used for a transfer address on an END directive is undefined.
- ASM: 73 (E) Global phase error on 'symbol'.** There is a difference in globals between pass 1 and pass 2. This message may be caused by declaring the same global twice or by declaring a global through string substitution.
- ASM: 74 (E) Class phase error.** A class name was encountered on pass 2 that was not encountered on pass 1. This could be caused by declaring a class name through string substitution.
- ASM: 75 (E) Undefined origin expression.** The operand of an ORG directive is either undefined or a forward reference. This error may occur if a misspelled or undefined symbol appears in an ORG directive, or if these directives reference a symbol that has not yet been assigned a value.
- ASM: 76 (E) Origin outside current section.** The expression used with an ORG directive is not an address within the current section. This error may occur if a misspelled or invalid symbol is used within an ORG expression or if a SECTION or RESUME statement is missing.
- ASM: 77 (E) Invalid RESERVE expression.** The value specified as the length of the RESERVE section is not a positive scalar.
- ASM: 78 (E) Undefined section name on RESUME.** The resumed section has not been previously defined with a SECTION or COMMON directive. This error may occur if the parameters of the SECTION or COMMON directives are misspelled or contain invalid characters.
- ASM: 79 (E) Symbol not a section name.** The symbol in the operand field of the RESERVE directive is not a section name.
- ASM: 80 (E) RESUME of RESERVE section.** A RESUME directive has been used with a RESERVE section name. It is illegal to RESUME a RESERVE section.
- ASM: 81 (F) Symbol table overflow.** The assembler symbol table is filled. This error occurs when too many symbols have been used. The source module must be divided into smaller pieces to permit assembly.
- ASM: 82 (F) Cannot use class on an absolute section.** A class name is specified for an absolute section.

-
- ASM: 83 (F) Usage: asm [obj] [lst] src.** No input file was specified upon assembler invocation.
- ASM: 84 (F) Not enough memory.** An internal work area of the assembler is full. This error may occur if the source program is too big, or if system memory has been removed or otherwise made unavailable to the assembler.
- ASM: 85 (F) Cannot open listing file.** The assembler is unable to open the specified file. Make sure that the filename is spelled correctly and is accessible by you.
- ASM: 86 (F) Cannot open object file.** The assembler is unable to open the specified file. Make sure that the filename is spelled correctly and is accessible by you.
- ASM: 87 (F) Source file used as object or listing file.** The source file has been named as object or listing file. This would cause the source file to be overwritten. Control is returned to the operating system.
- ASM: 88 (E) Object and listing file names not unique.** The object filename and the list filename are the same.
- ASM: 89 (E) Address operand required.** An address value is required.
- ASM: 91 (E) Nesting error.** An IF or REPEAT block is still open when end-of-file is encountered. This message may result from omitting an ENDIF or ENDR directive.
- ASM: 92 (E) Improperly terminated source line; accepted.** The last line in the source file is not terminated with a RETURN.
- ASM: 94 (E) ENDM encountered before end of MACRO text; ignored.** A text substitution has placed an ENDM directive within an macro.
- ASM: 95 (E) Assembler overlay not found.** Part of the assembler is missing or has been renamed.
- ASM: 97 (E) Macro definition phase error.** The macro has been defined in the second (but not the first) pass of the assembler. This error may be caused by conditional assembly and/or text substitution.
- ASM: 98 (E) String length phase error.** The declared length in the STRING directive differs between the first and second assembler passes. This error may be caused by conditional assembly and/or text substitution.
- ASM: 99 (E) Bad floating point syntax 'float-expression'; ignored.** The expression in the operand field of a FLOAT directive is not in proper format. See the Assembler Directives section of this manual for information concerning the proper format.
- ASM: 100(E) File 'filename' write error.** The assembler is unable to write to the specified file. Make sure that the filename is spelled correctly and writable by you.

- ASM: 101(E) SEG index is zero.** The starting position parameter of the SEG function is zero.
- ASM: 102(E) Negative SEG argument.** A parameter of the SEG function is negative.
- ASM: 103(E) Section larger than address space.** The section contains more bytes than the microprocessor is capable of addressing.
- ASM: 105(W) 'float-expression' too large; set to infinity.** The operand used in the FLOAT directive is too large. See the Assembler Directives section of this manual for information about what is stored.
- ASM: 106(W) 'float-expression' too small; denormalized.** The operand used in the FLOAT directive is too small. See the Assembler Directives section of this manual for information about what is stored.
- ASM: 107(E) Undefined opcode 'illegal-opcode'.** The assembler is unable to recognize or process a symbol or character in the operation field of a statement. This error may occur if an instruction is misspelled or if an invalid delimiter follows a label.
- ASM: 108(E) Symbol 'symbol' previously defined; value changed.** A symbol has been redefined. This error occurs if the same symbol is equated to two different values (with EQU directives). The symbol assumes the value given in the latest EQU directive.
- ASM: 109(E) Unable to restore previous input.** The assembler is unable to reopen the original source file after an INCLUDE file is read. The source file was changed while the INCLUDE file was being read.
- ASM: 110(E) Memory wraparound.** The program counter exceeds FFFFFFFF (4,294,967,295 decimal).
- ASM: 111(E) Section larger than page size.** A section with the relocation type INPAGE exceeds one page in length. Assembly continues. The program probably won't link correctly. See the Assembler Specifics section of this manual for the page size of your microprocessor.
- ASM: 112(E) Section larger than segment size.** The section is larger than the segment size of the target microprocessor. Assembly continues. The program probably won't link correctly. See the Assembler Specifics section of this manual for the segment size of your microprocessor.
- ASM: 113(E) Invalid alignment.** The data or instruction is out of alignment. Some microprocessors require instructions and/or data to begin on certain boundaries (for example, word boundaries). See the Assembler Specifics section of this manual for the alignment requirements for your microprocessor.

Linker Errors

There are two types of linker errors:

1. **Command processing errors** are errors detected while the linker is interpreting the invocation command. These errors are numbered in the range 1–99.
2. **Linking errors** are errors that are encountered during the linking process. These errors are numbered in the range 100–199.

All linker errors are listed on the standard error device (usually the terminal). They may also be listed in the listing file if the `-l` command option has been processed before the error is detected **and** standard output has been redirected. If the `-l` command option is used without standard output being redirected, any error messages will appear twice on the terminal.

The **severity-code** may be one of the following:

- W** (warning)—The error should have no effect on the linked program's execution.
- E** (error)—The linked program probably won't execute properly.
- S** (serious)—The linked program will not execute properly. The load file is not generated.
- F** (fatal)—The error directly affects the linker's execution. The linker closes all open channels, deletes the load file, and returns to the operating system.

In some error messages, a symbol in single quotes is used to represent a filename, digit-string, or other information that will be included in the actual message. For example:

LINK:112 (E) Section 'name' not found in files.

This symbol is replaced with the name of a section when the message is generated.

- LINK: 1 (F)** **Usage: link -<CDLOcdlmorstx> [object files].** The command was not recognized. This error occurs when an illegal command option is used.
- LINK: 2 (F)** **Invalid filename.** A filename containing non-printing ASCII characters has been encountered in the command line.
- LINK: 3 (F)** **Cannot open command file.** The specified command file cannot be opened. This usually occurs if the file doesn't exist or the name was mistyped.
- LINK: 4 (W)** **Syntax error.** The syntax of the command is invalid. See the Linker section of this manual for information about command syntax.
- LINK: 5 (W)** **Extraneous information ignored.** Extra characters were included on a command line after the required parameter(s). The extra characters are ignored.
- LINK: 6 (W)** **Invalid name.** An invalid symbol name has been found in a command line. See the Linker section of this manual for information about forming valid symbol names.

-
- LINK: 7 (W)** **Invalid number.** An invalid value has been found in the **-D**, **-t**, or **-x** command option.
- LINK: 8 (W)** **Invalid range.** One of the range specifications in the **-m** or **-L** command options is invalid. The starting address must be lower than the ending address.
- LINK: 9 (W)** **Invalid address.** One of the address specifications in either the **-m** or **-L** command option is invalid.
- LINK: 10 (W)** **Undefined memory.** A logical memory name specified in the **BASE** or **RANGE** specification of the **-L** command option was not defined in a previous **-m** command option.
- LINK: 11 (W)** **Memory range undefined.** A **-m** command option was given after a **-L** command option.
- LINK: 12 (W)** **Memory definition overlap.** Two logical memory areas have overlapped.
- LINK: 13 (W)** **Invalid option.** An invalid parameter option was specified. For example, an invalid relocation type may have been specified in a **-L** command option.
- LINK: 14 (W)** **Section already in a class.** A section appears in two **-C** command options. The section belongs to the class specified in the first **-C** command option.
- LINK: 15 (W)** **Class or section located twice.** Two **-L** command options for the same section or class have been encountered. The second **-L** for the section is ignored.
- LINK: 16 (W)** **Relocation type redefined.** Two **-t** command options of different relocation types for the same section or class have been encountered. The second **-t** command option is ignored.
- LINK: 17 (W)** **No continuation line provided.** A command continuation indicator (*****) in the command file is followed by an end-of-file.
- LINK: 18 (W)** **Missing link files.** No input files were specified. Always specify at least one input file when you invoke the linker.
- LINK: 19 (W)** **Memory previously defined.** The same logical memory name is specified more than once in **-m** command options.
- LINK: 20 (W)** **Missing command file.** The command file was not specified in the **-c** command option.
- LINK: 21 (W)** **Section already located to memory.** The section named in the **-C** command option was already located to an area of memory.
- LINK: 22 (W)** **Map undefined.** A memory name is used in a **-L** command option when no **-m** command option has been processed.

-
- LINK: 23 (W)** **Bad base.** A global symbol in a **-D** command option is based on a logical memory name or a class name.
- LINK: 24 (W)** **The following command line is too long.** The command line is longer than 80 characters (including RETURN). Only the first 79 characters are processed.
- LINK: 25 (W)** **Missing class or section name.** No class or section name is given in a **-t** command option.
- LINK: 26 (W)** **Missing class name.** No class name is given in the **-C** command option.
- LINK: 27 (W)** **Missing define parameter.** No name to define is given in the **-D** command option.
- LINK: 28 (W)** **Missing locate parameter.** No parameter is given in the **-L** command option.
- LINK: 29 (W)** **Missing load file.** No filename is given in the **-o** command option.
- LINK: 30 (W)** **Missing map definition.** No memory name is given in the **-m** command option.
- LINK: 31 (W)** **Missing type parameter.** No parameter is given in the **-t** command option.
- LINK: 32 (W)** **Missing transfer address.** No address is given in the **-x** command option.
- LINK: 33 (F)** **Input and output filenames not unique.** The input and output filenames are the same.
- LINK: 34 (W)** **Name previously defined.** A symbolic name has been given conflicting definitions during command processing. The first definition is used.
- LINK:100 (S)** **Name 'name1' in section 'name2' previously defined.** An attempt has been made to redefine a global symbol. This occurs when two modules both define a global symbol of the same name. The first symbol value encountered is used. If the global symbol is a section name, the linker will only include the first section encountered in the load module.
- LINK:101 (F)** **Internal linker error.** Serious problems have been detected. Contact your Tektronix service representative if this problem persists.
- LINK:102 (F)** **Memory overflow.** The linker has run out of memory during allocation of one of its internal data structures. Linking is terminated. The total number of global symbols, modules, or input files must be reduced in order to link in the memory available.
- LINK:103 (F)** **Operating system error.** An error has occurred in the operating system. Consult your system user's manual for information about this problem.

- LINK:104 (F)** File 'filespec' read error. The operating system reported an error when trying to read from an input file. Make sure that the filespec is spelled correctly and is readable by you. This error may result from a hardware problem.
- LINK:105 (F)** Checksum error in 'filespec'. A checksum is incorrect, indicating an error during a read. Try again.
- LINK:106 (F)** Invalid object format for 'filespec'. The object file is not in valid format.
- LINK:107 (F)** No object files. No object files were specified. Always specify at least one input file when you invoke the linker.
- LINK:108 (E)** Cannot open 'filespec'. The linker is unable to open the specified file. Make sure the filespec is spelled correctly and the file is accessible by you.
- LINK:109 (E)** File 'filespec' write error. The linker is unable to write to the specified file. Make sure the filespec is spelled correctly and the file is writable by you. This error may result from a hardware problem.
- LINK:110 (E)** No memory allocated to 'name'. The specified section is too big to relocate in the available contiguous memory in the range specified.
- LINK:111 (E)** Unresolved global reference 'name' at 'address'. An undefined global symbol was referenced at the specified address. This error occurs when a global symbol is used in one module but never defined. The unresolved reference is zero-filled in the load file.
- LINK:112 (E)** Section 'name' not found in files. A section specified in a command was not found in any object file.
- LINK:113 (E)** Section 'name' does not fit in logical memory. A section in a symbol file does not fit entirely within a logical memory.
- LINK:114 (E)** Absolute section 'name' 'name' conflicts with -L switch. An attempt has been made to relocate an absolute section. The command is ignored.
- LINK:115 (E)** Truncation error at 'address'. A value calculated during relocation was truncated when replaced in the load module because the receiving field was smaller than the calculated value.
- LINK:116 (S)** Section 'name' too big. The length of a RESERVE section is greater than the addressing space of the microprocessor. The section is not included in the load module. This error may occur when the combined length of several RESERVE sections is too large.
- LINK:117 (S)** Name 'name1' in file 'name2' previously defined. A symbolic name has been given conflicting definitions during command processing. The first definition is used.

-
- LINK:118 (W)** **Transfer address undefined.** No transfer address was specified. The linker outputs a transfer address of 0 in the load module.
- LINK:119 (W)** **Processor changed from 'name1' to 'name2'.** The current input module has been generated for a different microprocessor than the previous object modules. Be aware that differences between microprocessors may cause incompatibility during linking (such as page size, alignment).
- LINK:120 (W)** **Data 'name' typing error.** Type definitions of the named global are different in the referencing and referenced modules.
- LINK:121 (W)** **Subroutine 'name' typing error.** Type definitions of the parameters or the function types are not matched in the referencing and referenced modules.
- LINK:122 (W)** **Section 'name1' already linked to 'name2'.** The class name in the linker command conflicts with the class name specified in the object file. The name given in linker the command supersedes that given in the object file.
- LINK:123 (W)** **Relocation type conflict for 'name'.** There is a conflict in the relocation type for the COMMON or RESERVE section among the declarations in different modules. The first declaration encountered is used.
- LINK:124 (W)** **Number of globals in 'name' does not match.** The linker detects a discrepancy in the number of global symbols. This could happen if some global symbols were not processed due to an earlier error, such as a multiple definition.
- LINK:125 (W)** **Reserved name 'name' used incorrectly.** ENDREL, a reserved name, was used as something other than as a global symbol.
- LINK:126 (W)** **Invalid type 'name' found.** A type name from the compiler had been used previously as something other than a type.
- LINK:127 (W)** **Relocation type redefined for section 'name'.** The relocation type specified in the -t command option differs from that specified in the assembler source file. The relocation type specified in the -t linker command option is used.
- LINK:128 (E)** **Absolute or symbol file section 'name' cannot be relocated.** An absolute or symbol file section name was used in a -t, -C, or -L command option.
- LINK:129 (E)** **Symbol not linked.** An attempt has been made to use a section name which belongs to an empty section.

LibGen Errors

All LibGen errors are listed on the standard error device (usually the terminal). They may also be listed in the listing file if the `-l` command option has been processed before the error is detected and standard output has been redirected. If the `-l` command option is used without standard output being redirected, any error messages will appear twice on the terminal.

The **severity-code** may be one of the following:

W (warning)—The error should have no effect on program's execution.

E (error)—The program probably won't execute properly.

F (fatal)—The error directly affects LibGen's execution. LibGen closes all channels and returns to the operating system.

In the following discussion, 'oldlib' refers to the filespec associated with the `-o` command option and 'newlib' refers to the filespec associated with the `-n` command option.

In some error messages, a symbol in single quotes is used to represent a filename, digit-string, or other information that will be included in the actual message. For example:

LIBGEN : 10(F) Invalid file name 'filespec'

This symbol is replaced with the filespec of the file when the message is generated.

LIBGEN : 1(F) Both oldlib and newlib are missing. An attempt has been made to modify, list, or create a new library without providing an existing library filespec or valid new library filespec. Always provide the filespec of the library to be modified, listed, or created.

LIBGEN : 2(F) Cannot find end block for module in file 'filespec'. 'filespec' is not a valid object file. Verify that you have specified the correct filespecs in your `-i` and `-r` command options.

LIBGEN : 3(F) Cannot find end block for module 'module-name' of library 'oldlib'. The specified module contained in library file 'oldlib' has an invalid object format. You must recreate the module file.

LIBGEN : 4(F) Cannot find library end block in library 'filespec'. LibGen cannot find the end of the library block when modifying or listing an existing library. You must recreate the library file.

LIBGEN : 5(F) Checksum error in file 'filespec'. The checksum value calculated by LibGen does not equal the value stated by the module file or library file. The module format is incorrect. You must recreate the module file or library file.

LIBGEN : 6(E) Command line too long. A command line in the command file has more than 80 characters. Only the first 79 characters are processed.

- LIBGEN : 7 (E)** **Duplicate module name:** 'module-name'. The specified module to be inserted into the library is already defined in the library. The new module is not inserted and LibGen processing continues. When creating a library, be sure to give each object module a unique name with the assembler NAME directive.
- LIBGEN : 8 (W)** **Duplicate symbol name:** 'symbol-name'. Two or more global symbols within the library file have the same name. This condition does not affect the performance of the linker when selecting modules, but makes future modification and maintenance of the library difficult. When creating a library, be sure to give each symbol a unique global name.
- LIBGEN : 9 (F)** **Cannot open file 'filespec'.** 'filespec' does not exist. Verify that the file exists and that the file is accessible by you.
- LIBGEN : 10(F)** **Invalid file name:** 'filespec'. 'filespec' contains invalid character(s). Verify that the filespec contains only legal characters.
- LIBGEN : 11(F)** **Invalid object format for file 'filespec'.** 'filespec' is not a valid object file. Verify that you have specified the correct filespecs in your `-i` and `-r` command options.
- LIBGEN : 12(F)** **I/O error on file 'filespec'.** The operating system reported an I/O error while accessing the specified file.
- LIBGEN : 13(W)** **Library file is replaced by 'filespec'.** The `-o` command option is entered more than once in the same LibGen invocation line. Only the last `-o` command option is recognized.
- LIBGEN : 14(F)** **Memory overflow.** LibGen has run out of memory during allocation of one of its internal data structures. The total number of global symbols, modules, or input files must be reduced.
- LIBGEN : 15(E)** **Module not found in library:** 'module-name'. The module specified in a `-d`, `-x`, or `-r` command option was not found in the old library. In the case of the `-d` or `-x` options, the command option is ignored. In the case of the `-r` option, the module is still placed in the library.
- LIBGEN : 16(F)** **Newlib filespec missing.** An attempt has been made to modify a library without providing a valid filespec for the output library file. Always provide a valid output filespec when modifying an existing library.
- LIBGEN : 17(E)** **No continuation line provided.** A command continuation indicator (`*`) in the command file is followed by an end-of-file.

-
- LIBGEN : 18(F)** **Oldlib filespec missing.** An attempt has been made to modify or list an existing library file without providing a valid filespec for the old library. Always provide a valid input filespec when modifying or listing an existing library.
- LIBGEN : 19(F)** **'Oldlib' not a library.** 'Oldlib' is not a library file. Verify that you have specified proper filespec in the `-o` command option.
- LIBGEN : 20(W)** **'Symbol-name' has type conflict.** The specified symbol contained in an existing library or new library has more than one type definition.
- LIBGEN : 21(F)** **Syntax error: command.** The command option does not conform to the proper syntax for that command option. Refer to the Library Generator section of this manual for proper command option syntax.
- LIBGEN : 22(W)** **Type conflict with 'nnn' external symbol(s).** The library contains the specified number of external symbol(s) that have type conflicts. To locate the symbol(s), look for warning messages contained in the listing.
- LIBGEN : 23(F)** **'filespec' write error.** LibGen is unable to write to the specified file. Make sure that the filespec is spelled correctly and that you have write permission to the file.
- LIBGEN : 24(F)** **'filespec' read error.** LibGen is unable to read the specified file. Make sure the filespec is spelled correctly and you have permission to read the file.
- LIBGEN : 25(F)** **Usage: LIBGEN -<cdhilnorvx> [modname] [filespec].** LibGen must be invoked with the legal command options. Legal options are `-c`, `-d`, `-h`, `-i`, `-l`, `-n`, `-o`, `-r`, `-v`, and `-x`. Any other command option is illegal.

Section 13

GLOSSARY

Absolute. Having a specified location in memory; not relocatable. An absolute address specifies the actual location of a byte in memory.

Actual Parameter. See **Parameter**.

Address. A number or symbol that specifies a byte in memory. A 32-bit address has a value in the range 0 to FFFFFFFF (hexadecimal).

Assembler. A system program that translates assembly language programs into machine language.

Assembly Language. A microprocessor-specific programming language that allows the symbolic representation of any processor operation. Each operation is coded as one assembly language statement.

Base. The base of a section of object code is the location of the first byte in the section.

Binary. The base 2 numbering system. A binary digit, or bit, has the value 0 or 1. A binary constant in an assembly language program requires the suffix B or b. For example, the decimal number 29 may be written as 11101B or 11101b.

Bound Global. See **Global**.

Byte-Relocatable. See **Relocatable**.

Carriage Return. See **Return**.

Code. To translate a step-by-step procedure into a series of assembly language statements. This series of statements constitutes an assembly language program. The statements of such a program are called **source code**. The machine instructions produced by assembling the source code (usually done by the assembler) are called **object code**. Object code represents the assembly language program in a form that the microprocessor understands.

Command File. A file containing commands to be processed by the operating system or by a system program such as the linker or library generator.

Comment. A source program line, or part of a line, that is ignored by the assembler. Comments are used for program documentation. A semicolon (;) signifies that the rest of the line is a comment. Comments may also be embedded within Linker and LibGen command files (if the semicolon is not the first character in the command line, a space must precede it).

Common. A section of memory that may be shared by any number of subprograms. The assembler directive `COMMON` declares a common section. The linker assigns the same area of memory to all common sections with the same name.

Concatenation. Connecting end-to-end. For example, the concatenation `'FLIP':'FLOP'` yields the string `'FLIPFLOP'`. The colon (`:`) is the concatenation operator used in assembly language programs.

Conditional Assembly. A feature of the TEKTRONIX Assembler that allows a block of source code to be assembled a certain number of times or not at all, depending on conditions defined earlier in the source module.

Constant. A value expressed in literal form rather than as a symbol. A **numeric constant** is written as a string of digits, optionally followed by a letter that indicates the radix (for example, `29`, `11101B`, `35O`, `1DH`). A **string constant** is written as a character string enclosed in single quotes (for example, `'TEXT'`, `'P.O. Box 500'`, `''`). See also **Floating Point Constant**.

Data Item. A byte or sequence of bytes of object code that contains data other than machine instructions. A data item is defined by an `ADDRESS`, `ASCII`, `BYTE`, `FLOAT`, `LONG`, or `WORD` directive.

Default Value. A predefined value for a parameter, used when no value for the parameter is explicitly specified.

Defined Symbol. A symbol that has been assigned a value.

Directive. An assembly language statement that does not represent a machine instruction but does provide special information to the assembler. Also called a pseudo-operation, pseudo-instruction, or quasi-instruction.

Expression. A formula that contains symbols, constants, or functions related by operators, and yields a numeric or string value when evaluated. Symbols, constants, and functions are themselves trivial expressions.

Floating Point Constant. A constant, specified by the `FLOAT` assembler directive, that is stored in 32 or 64 bits of memory, according to the proposed IEEE Floating Point Formats.

Formal Parameter. See **Parameter**.

Forward Reference. Use of a symbol, in the current assembly language statement, that has not yet been defined by any of the preceding statements; that is, the symbol is defined in a succeeding statement in the current source module.

Function, Assembler. A predefined function that may be used in assembly language expressions. An assembler function has the form `func(expr)`, where `func` is the name of the function and `expr` is one or more expressions separated by commas.

Global (or Global Symbol). A symbol that is assigned a value in one module and referenced in another. A **bound** global is defined in the current module by its use as a label. An **unbound** global is undefined in the current module; its value must be supplied by another module or by the linker command **-D**.

Hexadecimal. The base 16 numbering system. Hexadecimal digits include the digits 0 through 9, and the letters A through F to represent the decimal values 10 through 15. A hexadecimal constant in an assembly language program requires the suffix H or h, and must begin with a decimal digit (to distinguish it from a symbol). For example, the decimal number 29 may be written as 1DH or 1Dh. The decimal number 15 may be written as 0FH or 0Fh (but not FH or Fh).

Inpage-Relocatable. See **Relocatable**.

Instruction. A **machine** instruction is a sequence of bytes that directs a microprocessor to perform an elementary operation such as load, store, add, or branch. An **assembly language** instruction is an alphanumeric representation of a machine instruction. The assembler translates an assembly language instruction (source code) into the corresponding machine instruction (object code).

Label. A symbol, located in the label field of a source line in an assembly language program, that represents an address, variable, or constant.

Library. A collection of object modules that usually contains commonly-used subroutines. You may include calls to library routines in your source program; the linker includes the necessary object modules in the load file.

Library Generator (LibGen). A system program used to create and maintain libraries of object modules.

Linker. A system program that combines object modules into a single executable load file.

Listing. A file or printout that summarizes the actions of a program such as the assembler, linker, or library generator.

Local. Not global. In an assembly language program, a local symbol is referenced only by statements in the same source module.

Location Counter. An internal counter maintained by the assembler that marks the location, relative to the beginning of the section, of the next machine instruction to be assembled. A symbol in the label field of an assembly language statement is usually assigned the current value of the location counter.

Machine Instruction. See **Instruction**.

Machine Language. The binary language of a microprocessor. A high-level or assembly language program must be translated into machine instructions before the microprocessor can execute the program. Relocatable machine language produced by the assembler may require adjustment by the linker in order for the instructions to execute properly.

Macro. A frequently-used group of assembler statements that are inserted into the program at assembly time wherever the macro is invoked.

Macro Definition. A group of assembler statements that define a macro. A macro definition begins with a MACRO directive and ends with an ENDM directive. Statements in the macro definition may contain formal parameters, which are replaced with actual parameters wherever the macro is invoked.

Macro Expansion. The process of replacing a macro invocation with the macro definition block it invokes.

Macro Invocation. An assembler statement containing the name of a macro in the operation field and, optionally, a list of actual parameters in the operand field.

Mnemonic. A symbol that represents a machine instruction. Usually the symbol is an abbreviation that suggests the machine operation to be performed. For example, the 8086/8088 mnemonic MOV represents a machine instruction that moves data from a register or memory location into another register or memory location.

Module. A program unit that is complete for purposes of assembling, linking, or loading. It may be combined with other modules to produce a complete program. An **object module** contains all the object code produced in a single assembler run. A **source module** is a set of assembly language statements (ending with an END directive or an end-of-file) that produces an object module when assembled.

Nest. (1) To include a block of assembly language statements inside another block of statements. (2) To include a subexpression within an expression.

Null String. An empty character string ("").

Object Code. Machine language produced by the assembler from source statements. An **object module** contains one or more sections of object code, plus special information used by the linker, library generator, or the operating system command that loads the object code into memory. An **object file** is a file that contains an object module.

Octal. The base 8 numbering system. The eight octal digits are 0 through 7. An octal constant in an assembly language program requires any of the suffix letters O, o, Q, or q. For example, the decimal number 29 may be written as 35O, 35o, 35Q, or 35q.

Operand. A number or other value on which an operation is performed. The expression $X + 3$ performs an add operation on the operands X and 3. The 8086/8088 assembly language statement NEG BX performs a two's complement operation on the byte addressed by the operand BX.

Operator. A character or symbol that represents an operation to be performed on one or more operands. Operators used in assembly language programs are:

*	/	+	-	MOD		(arithmetic)
\	&	!	!!	SHL	SHR	(bit manipulation)
=	<	<=	>	>=	<>	(relational)
:						(string concatenation)

Page. A subdivision of memory. Page size is processor-dependent and reflects addressing considerations. For example, in a 64K memory with 256-byte pages, the high-order byte of a 16-bit address selects one of the 256 pages, and the low-order byte of the address selects a byte within that page.

Page-Relocatable. See **Relocatable**.

Parameter. In an operating system command, a parameter is a name or number that follows the command word and tells something about how the command is to be executed.

In an assembler macro, a parameter is a value that remains undefined until the macro is invoked. A **formal parameter** is a place holder in a macro definition block; the first formal parameter is written as "1", the second as "2", and so on. An **actual parameter** is a character string in a macro invocation that replaces each occurrence of the corresponding formal parameter in the macro block. **Parameter** may refer to either a formal parameter or an actual parameter.

Program Memory. The memory used as a substitute for prototype memory in the early stages of prototype development (emulation modes 0 and 1). User programs run in program memory, as do the assembler, linker, library generator, and certain other system programs.

Relocatable. A relocatable section is a section whose location in memory is not determined until link time. A **page-relocatable** section must begin on a page boundary; an **inpage-relocatable** section may not cross page boundaries; a **byte-relocatable** section may be positioned anywhere in memory; an **aligned-relocatable** section may start at any address that is an integer multiple of a specified value; an **absolute** section must start at a specified address. (However, depending on the microprocessor, a byte-relocatable section may have to start on a certain boundary, such as a word boundary. See the Assembler Specifics section of this manual for more information.)

Reserved Word. A predefined symbol that has a special meaning to the assembler and may not be used as a label or section name. Reserved words include mnemonics, register names, and assembler directives and functions.

Return. The RETURN character is also called CR or carriage return. This character marks the end of a command or an assembly language statement. See the Host Specifics section of this manual for the ASCII code value representing the carriage return.

Scalar. A 32-bit signed numeric value not used as an address. A scalar takes a value in the range -2147483648 to +2147483647.

Section. A section of **object code** is a block of contiguous bytes, and is the fundamental, indivisible unit handled by the linker. A section of **source code** comprises the statements that will produce a section of object code when they are assembled. Each section of source code begins with a SECTION, COMMON, or RESERVE directive.

Source Code. Program statements written in assembly language. A **source module** is a set of source statements (ending with an END directive or an end-of-file) that produces an object module when assembled. A **source file** is a file containing all or part of a source module.

String. A sequence of ASCII characters. A string enclosed in single quotes (for example, 'ELEPHANT') is called a **string constant**.

Symbol. A string of 1 to 16 characters beginning with a letter and containing only letters, digits, periods, underscores, or dollar signs. **Predefined symbols** include assembler directives and functions, mnemonics, and register names. **User-defined symbols** represent addresses, data items, variables, macros, or sections.

Transfer Address. The address of the first machine instruction to be executed in a load file. A transfer address may be specified in the END statement of a source module or in the linker command option -x.

Unbound Global. See **Global**.

Variable. In an assembly language program, a symbol whose value may be altered during assembly time. A variable is defined and redefined by the use of the SET directive.

Section 14

INDEX

A

Absolute, defined, 13-1
 Absolute relocation type, 3-8, 3-53
 Actual parameter, defined, 13-5
 Address, defined, 13-1
 ADDRESS directive, 3-3
 Address values, 2-10
 comparisons, 2-20
 Addressing modes, section 9
 ALIGN, relocation type, 3-8, 3-53, 5-14, 5-17
 Allocation of sections, 5-18
 Arithmetic operators, 2-16
 ASCII codes, table of, 11-5
 ASCII directive, 3-5
 ASM command, 2-2
 Assembler:
 defined, 13-1
 execution, 2-36
 features, 1-4
 input, 2-2
 invoking the, 2-1
 macro. See Macro
 object module, 2-37
 output, 2-37
 variable, 2-11
 Assembler directives, section 5
 defined, 13-2
 labels with, 3-2
 list of, 11-3
 Assembler listing, 2-37
 explanation of, 2-38, 2-45
 example, 2-40
 headings, 3-59, 3-62
 options, 3-31
 statistics, 2-39, 2-53
 Assembler specifics, section 9
 Assembly:
 conditional:
 blocks, 3-26, 3-46
 defined, 13-2
 example of, 7-2
 Assembly language, defined, 13-1
 Assembly language instructions:
 defined, 13-3
 notational conventions for, section 9

B

BASE function, 2-22
 Base, defined, 13-1
 Binary, defined, 13-1
 BITS function, 2-24

BLOCK directive, 3-6
 Bound global:
 description, 3-24
 defined, 13-3
 BYTE directive, 3-7
 Byte-relocatable, defined, 13-5

C

Carriage return, 2-3, 13-5
 Characters, special:
 @ (at sign), 4-4, 7-33
 \$ (dollar sign), 2-10
 % (percent sign), 2-13, 4-5
 # (pound sign), 4-5
 ^ (up arrow), 2-7, 2-11, 4-6, 4-9
 disabling significance of, 2-7, 2-11, 4-6, 4-9
 Class name, 3-8, 3-50, 3-53, 5-16
 CND (listing option), 3-32
 Code, defined, 13-1
 Command file, 5-8, 5-15, 6-3, 6-4
 defined, 13-1
 Command file invocation:
 LibGen, 6-3, 6-4
 linker, 5-8, 5-15
 Command name, 1-6
 Comment, defined, 13-1
 Comment field, 2-7
 Common, defined, 13-2
 COMMON directive, 3-8
 Common section, 3-8
 CON listing option, 3-33
 Concatenation:
 defined, 13-2
 string, 2-21
 Conditional assembly, 3-26, 3-46
 defined, 13-2
 examples, 7-2
 Constant:
 defined, 13-2
 numeric, 2-10
 string, 2-11
 Constant values, example of creating, 7-10
 Conversions, 2-12
 <CR> (carriage return), 2-3
 defined, 13-5
 Cross-reference listing:
 controlling display of, 3-33
 description, 2-39
 example of, 2-42
 explanation of, 2-51
 Current section name, 4-5

D

Data item, defined, 13-2
 DBG listing option, 3-33
 Decimal-hexadecimal-binary equivalents, table of, 8-6
 DEF function, 2-26
 Default object module name, 3-38
 Default section, 2-52, 3-54
 Default value, defined, 13-2
 Defined symbol, defined, 13-2
 Differences between Series A assembler and Series B assembler, section 10
 Directive, defined, 13-2
 See also each directive by name.

E

ELSE directive, 3-12, 3-26
 ELSEIF directive, 3-13, 3-26
 END, directive, 3-14
 ENDF directive, 3-15
 ENDM directive, 3-16, 4-6
 ENDOF function, 2-27
 ENDR directive, 3-17
 ENDREL, 2-9, 5-19
 Entry point, 3-24
 EQU directive, 3-18
 Error messages:
 assembler, 12-1
 LibGen, 12-15
 linker, 12-12
 processor-specific, section 9
 user-defined, 3-64
 Escape character (^), 2-7, 2-11, 4-6, 4-9
 Execution, assembler, 2-36
 EXITM directive, 3-19, 4-6
 EXITR directive, 3-20
 Expression, 2-13, 3-27, 13-2

F

Field:
 comment, 2-7
 defined, 2-3
 label, 2-4
 operand, 2-6
 operation, 2-5
 File naming, section 8
 FLOAT directive, 3-21
 Floating point constant, 13-2

Floating point values, 2-10
 Formal parameter, defined, 13-5
 Forward reference:
 defined, 13-2
 use of, 2-36
 Functions, assembler, 2-21
 defined, 13-2
 BASE, 2-22
 BITS, 2-24
 DEF, 2-26
 ENDOF, 2-27
 HI, 2-28
 LO, 2-29
 NCHR, 2-30
 SCALAR, 2-31
 SEG, 2-32
 STRING, 2-34
 STRINGOF, 2-35

G

Global:
 bound, 3-24
 defined, 13-3
 unbound, 3-24
 GLOBAL directive, 3-24
 Global symbols list, 5-26

H

Hexadecimal, defined, 13-3
 Hexadecimal addition, table of, 11-7
 Hexadecimal multiplication, table of, 11-7
 HI function, 2-28
 Host Specifics, section 8

I

IF directive, 3-26
 IF...ENDIF block, 3-26
 INCLUDE directive, 3-30
 using the, 7-34
 INPAGE relocation type, 3-8, 3-50, 3-53, 5-14, 5-17
 Inpage-relocatable, defined, 13-5
 Input,
 assembler, 2-2
 linker, 5-5
 Installation, assembler software, section 8
 Instruction, defined, 13-3
 Instruction set, processor, section 9
 Internal symbol list, 3-33, 3-34, 5-9

L

Label, defined, 13-3

Label field, 2-4

Label generation, unique ('@'), 4-4

Labels with assembler directives, 3-2

LibGen:

- command file, 6-3
- command options, 6-2, 6-3, 6-4
 - c, 6-4
 - d, 6-4
 - h, 6-4
 - i, 6-4
 - l, 6-4
 - n, 6-5
 - o, 6-5
 - r, 6-5
 - v, 6-5
 - x, 6-5
- defined, 13-3
- error messages, 12-15
- execution of, 6-9
- features, 1-5
- invocation, 6-1
 - command file, 6-3
- library file, 6-3, 6-4, 6-5, 6-6, 6-10
- listing, 6-10
 - command log, 6-10
 - module list, 6-11
 - summary of actions, 6-11
- output, 6-10

Library:

- creating a subroutine, 7-14
- defined, 13-3

Library file:

- as LibGen output, 6-3, 6-4, 6-5, 6-6, 6-10
- linking a, 5-19

Library generator. See LibGen

Library module:

- adding a new, 6-4, 6-6
- deleting a, 6-4, 6-6
- extracting a, 6-5, 6-8
- replacing a, 6-5, 6-7

LINE listing option, 3-33

Linker:

- command file, 5-8
- command options, 5-3, 5-5
 - C, 5-6
 - D, 5-7
 - L, 5-7
 - O, 5-8
 - c, 5-8
 - d, 5-9
 - l, 5-9
 - m, 5-12
 - o, 5-13
 - r, 5-13
 - s, 5-14
 - t, 5-14
 - x, 5-15
- completion condition, 5-20
- defined, 13-3
- error messages, 12-10
- execution, 5-16
- features, 1-5
- invocation, 5-1
 - command file, 5-8
- listing file, 5-21
- maps,
 - memory and section, 5-24
 - module and file, 5-23
 - module and section, 5-24
- output, 5-21
- PASCAL typechecking, 5-20
- statistics, 5-27

Linker listing, 5-21

- command log, 5-22
- global symbol, 5-26
- memory and section, 5-24
- module and file, 5-23
- module and section, 5-24

Linking to a library file, 5-19

Linking to an address range, 5-7

LIST directive, 3-31

Listing:

- assembler:
 - control of, 3-31, 3-39
 - example of, 2-40
 - explanation of, 2-45
- defined, 13-3
- headings for assembler, 3-59, 3-62
- LibGen, 6-10
- linker, 5-21
- source, 2-38
- See also Assembler listing

LO function, 2-29

Local, defined, 13-3

Location counter:

- defined, 13-3
- described, 2-10
- setting the, 3-40

Logical operators, 2-17

LONG directive, 3-36

M

Machine instruction, defined, 13-3

Machine language, defined, 13-3

Macro:

- body, 4-3
- definition, 4-2
 - defined, 13-4
- defined, 4-1, 13-4
- expansion:
 - defined, 4-1, 4-2, 13-4
 - display of statements in, 3-32
- invocation, defined, 4-2, 4-7, 13-4
- name, 4-2
- operators, 4-3
- parameter:
 - accessing, 4-3, 4-7
 - brackets, 4-7
 - conventions, 4-7
 - determining number of, 4-5
 - null, 4-9
 - single quote character, 4-8
 - unique label generation, 4-4, 7-33

MACRO directive, 3-37, 4-2

ME listing option, 3-32

MEG listing option, 3-32

Memory, reserving an area of, 3-6, 3-50

Memory map, 5-24

Mnemonic, defined, 13-4

Mnemonics, processor, section 9

MOD (modulus) operator, 2-16

Module:

- defined, 13-4
- object, 2-37

Module map, 5-23, 5-24

N

NAME directive, 3-38
 NCHR function, 2-30
 Nest, defined, 13-4
 Nesting conventions for assembly language
 statements, 3-27, 3-46
 NOLIST directive, 3-39
 NONAME, 3-38
 Null string, defined, 2-11, 13-4
 Numeric values, 2-9
 Numeric variable, 2-11

O

Object code defined, 13-4
 Object file, defined, 13-4
 Object module:
 defined, 13-4
 description of, 2-37
 name of, 3-38
 Octal, defined, 13-4
 Operand defined, 13-4
 Operand field, 2-6
 Operation field, 2-5
 Operators, 2-15
 arithmetic, 2-16
 defined, 13-5
 hierarchy of, 2-14
 logical, 2-17
 relational, 2-19
 string, 2-21
 table of, 2-14, 2-15
 ORG directive, 3-40
 Overlay, linking, 7-30

P

PAGE:
 directive, 3-45
 listing option, 3-33
 relocation type, 3-8, 3-50, 3-53, 5-14, 5-17
 Page (of memory), defined, 13-5
 Page size, processor, section 9
 Page-relocatable, defined, 13-5
 Parameter, defined, 1-6, 13-5
 Parameter count (macro), 3-56, 4-5
 Passes, assembler, 2-36
 Program memory, defined, 13-5

R

Register names, section 9
 Relational operators, 2-19
 comparison table, 2-20
 Relocatable, defined, 13-5
 Relocatable address, 2-10
 Relocation indicator, 2-38
 example of, 2-47
 Relocation of sections, 5-18
 Relocation type 3-8, 3-50, 3-53, 5-14, 5-17
 REPEAT directive, 3-46
 REPEAT...ENDR block, 3-46
 RESERVE directive, 3-50
 Reserve section, 3-50
 Reserved words, 2-9, section 9
 defined, 13-5
 RESUME directive, 3-52
 Return character, 2-3
 defined, 13-5

S

Scalar, defined, 13-5
 SCALAR function, 2-31
 Scalar values, 2-10
 comparisons, 2-19
 Section:
 allocation of, 5-18
 attributes, 5-16
 default, 2-52, 3-54
 defined, 13-6
 SECTION directive, 3-53
 Section name, determining current, 4-5
 Section type, 5-16
 SEG function, 2-32
 Semicolon (comment), 2-7
 Service call (SVC) generation, example of, 7-5
 Service request blocks, example of creating, 7-5
 SET directive, 3-55
 SHL (shift left) operator, 2-16
 SHR (shift right) operator, 2-17
 Source code, defined, 13-6
 Source file, defined, 13-6
 Source listing:
 control of, 3-31
 description, 2-38
 example of, 2-40
 explanation of, 2-45

Source module, defined, 13-6
 Source module character set, 11-1
 Source program, example of, 2-44
 SPACE directive, 3-58
 Stack:
 allocating memory for, 3-51
 saving register values on, 7-2
 Statement fields, 2-3
 Statements, 2-2
 STITLE directive, 3-59
 String, defined, 13-6
 String constant, 2-11
 String conversions, 2-12
 STRING directive, 2-11, 3-61
 STRING function, 2-34
 String operator, 2-21
 String values, 2-11
 comparisons, 2-20
 String variable, 2-11
 STRINGOF function, 2-35
 Subroutine library, example of creating and using
 a, 7-14
 SVC generation, example of, 7-5
 SYM listing option, 3-33
 Symbol, 2-7
 assigning value to, 2-7, 2-8
 constructing, 2-8
 defined, 13-6
 defining, 2-8
 predefined, 2-9
 user-defined, 2-8
 Symbol table:
 description, 2-39
 controlling display of, 3-33
 example of, 2-43
 explanation of, 2-52
 Syntax notation, 1-5
 for assembler directives, 3-1

T

Text substitution, 2-12
 current section name, 2-13, 3-57, 4-5
 macro parameters, 3-56, 4-3
 parameter count 3-56, 4-5
 unique label generation, 4-4
 Text substitution indicator, 2-38
 example of, 2-49
 TIMES REPEAT option, 3-46
 TITLE directive, 3-62
 Transfer address, defined, 3-14, 5-15, 13-6
 Two passes of the assembler, 2-36
 Type conversion, 3-55

U

Unbound global, defined, 3-24, 13-3
 Unique label generation, 4-4
 example of, 7-33
 User-defined error messages, 3-64

V

Variable:
 defined, 13-6
 numeric, 2-11, 3-55
 string, 2-11, 3-55

W

WARNING directive, 3-64
 WORD directive, 3-65

X

XREF listing option, 3-33

DESCRIPTION

TEXT CORRECTIONS

Page 12-16 For LIBGEN error message 9, replace the text paragraph with the following information:

Cannot open file 'filespec'. Verify that 'filespec' exists and that you can access the file.

If you are using an 8550 Microcomputer Development Lab, you may be trying to use too many I/O channels (a maximum of 8 are allowed). The console and printer each require an I/O channel. Each of the following operations requires a separate I/O channel: redirection (>), insertion (-i), deletion (-d), extraction (-x), old library (-o), and new library (-n). Also, the Library Generator requires an I/O channel for the temporary file that it generates.

Date: 2-25-82

Change Reference: C2/282

Product: 8500 MDL: Assembler Core Users Manual
for B Series Assemblers

Manual Part No.: 070-3856-00

DESCRIPTION

TEXT CORRECTION

Page 12-17 **Delete** LibGen error message number 22(W) and the corresponding text paragraph.

