Tektronix®
COMMITTED TO EXCELLENCE

This manual supports the following TEKTRONIX products:

8550 Options	Products
1A	8300A01
1B	8300A02
1C	8300A04
1D	8300A05
1E	8300A07
1F	8300A09
1G	8300A10
1H	8300A14
1J	8300A15
1K	8300A20
1L	8300A26
1M	8300A28

This manual supports the following software modules:

TEKTRONIX Assembler Version 4.X
TEKTRONIX Linker Version 4.X
TEKTRONIX LibGen Version 2.X

These software modules are compatible with:

DOS/50 Version 1.X

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

8500
MICROCOMPUTER DEVELOPMENT LAB SERIES
ASSEMBLER CORE USERS MANUAL
A Series Assemblers

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

070-3575-01
Product Group 61

Serial Number ____________

First Printing JUL 1980
Revised MAY 1983

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, cr (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.

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.

TABLE OF CONTENTS

Section 1
LEARNING GUIDE

ILLUSTRATIONS

Section 1
LEARNING GUIDE

INTRODUCTION

This Learning Guide gives an overview of features and functions of the TEKTRONIX Assembler, Linker, and Library Generator. It also presents a simple demonstration for hands-on experience. The Learning Guide is divided into the following topics:

- **About This Manual**. 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.
- **Installation.** Shows how to install the software for your 8080A/8085A assembler from the installation disk to your DOS/50 Version 2 system disk.
- **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.
- **For Continued Learning**. Helps you decide where to go next in this manual to accomplish your own tasks.

ABOUT THIS MANUAL

NOTE

*This manual supports DOS/50 Version 2. If you **must** use DOS/50 Version 1, then you must convert this manual to support DOS/50 Version 1. Refer to the discussion, "Using This Manual with DOS/50 Version 1", at the end of this section.*

The TEKTRONIX Assembler, Linker, and Library Generator are fundamentally the same for every microprocessor supported. Each assembler recognizes a different instruction set, different registers, and different addressing modes; however, you may use the same assembler directives, operand expressions, symbols, constants, and advanced programming features with any assembler provided.

The Assembler Specifics section of this manual gives the instruction set and other processor-dependent information for your microprocessor. The information in the rest of this manual applies to all microprocessors supported. Once you have used one version of the TEKTRONIX Assembler, Linker, and Library Generator, you can program for any microprocessor supported, as soon as you learn that microprocessor's instruction set.

The Demonstration Run in this Learning Guide demonstrates the 8080A/8085A assembler. Demonstration Runs for microprocessors other than the 8080A/8085A are found in the Assembler Specifics section.

Examples in the Demonstration Run and elsewhere in this manual were created using DOS/50 Version 2, the operating system of the 8550 Microcomputer Development Lab.

Programming examples in the Learning Guide and Assembler Specifics sections are specific to each microprocessor. All examples in other sections of this manual are completely processor-independent. Some examples use 8080A instructions, but similar instructions for any other microprocessor may be substituted without changing the validity of any example.

SYSTEM OVERVIEW

Figure 1-1 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 **linker** collects object modules from specified files and determines where in memory each section of object code will reside. The **load file** produced by the linker contains the executable program, which you may copy into memory using the operating system LO command. (Under certain conditions you may load object modules without linking them. See the Assembler Introduction section of this manual.)

Commonly used subroutines can be developed and assembled separately. Their object code can then be stored with other useful 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.

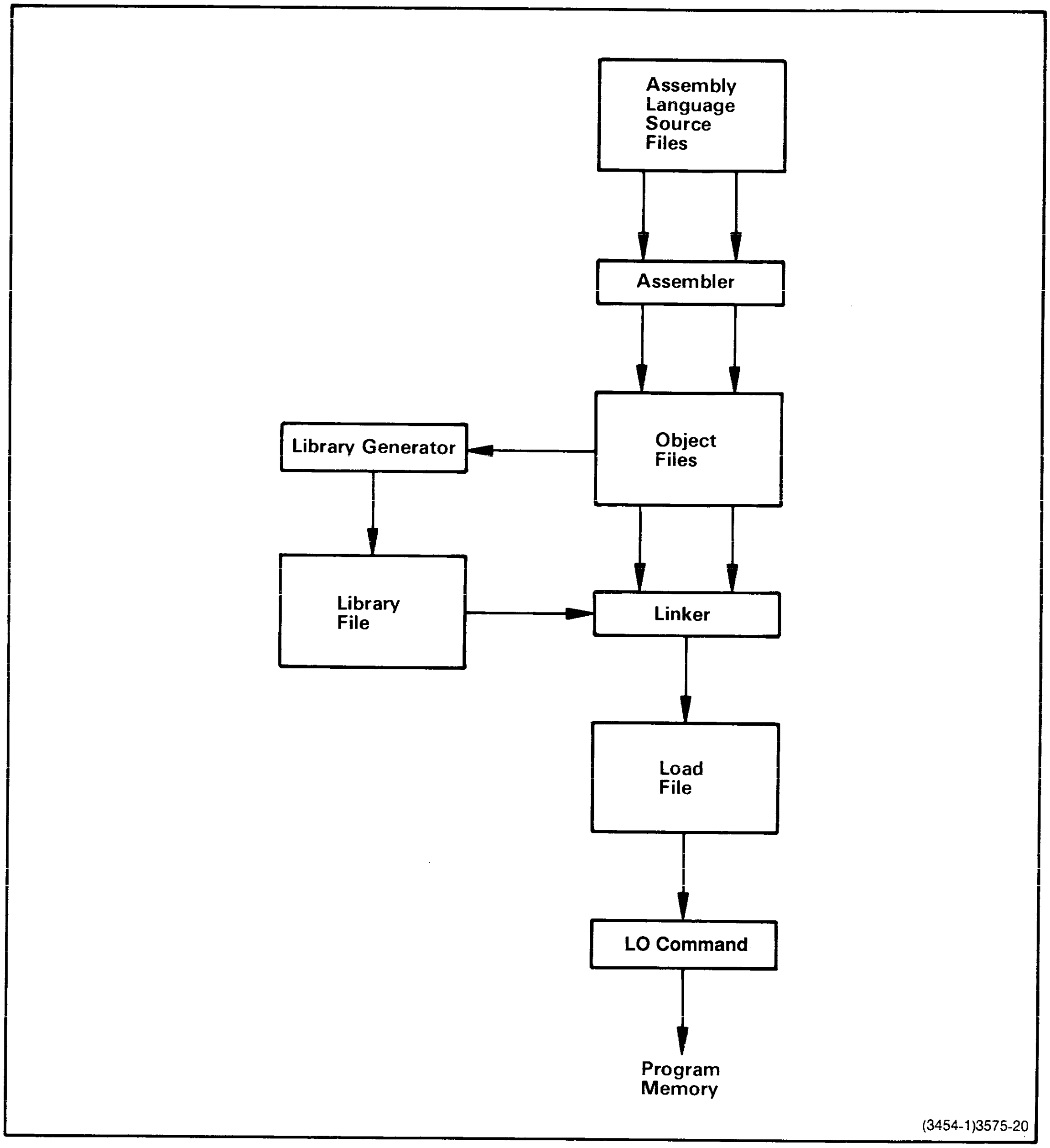


Fig. 1-1. Assembler programming process.

The assembler translates assembly language programs (source code) into relocatable machine language (object code). Commonly 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 operating system LO command copies object code from load files into program memory.

ASSEMBLER FEATURES

Here are some important features of the 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 quotes.
- Each error message contains a brief description of the error, plus an error number that helps you refer to this manual for more information. You may also write your own error messages for use in conditional assembly.
- The assembler listing shows your source code, and the object code, error messages, and symbol table 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 attributes at link time:
  - —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 specify simple linking operations with a single LINK command line. Special or complex operations can be specified with linker commands entered from the system terminal or from a command file.
- Each error message contains a brief description of the error and gives the severity of the error (WARNING, ERROR, or FATAL ERROR). When you enter an illegal command, the linker indicates which word or parameter is erroneous.
- The linker listing gives a detailed account of linker activity, showing the commands executed, local and global symbols, memory maps, and statistics.

LIBRARY GENERATOR FEATURES

Here are some important features of the library generator:

- You may create libraries of up to 100 modules from any number of object files.
- You may modify libraries by inserting, deleting, or replacing object modules.
- You may extract individual object modules into files.
- You may enter library generator commands from the system terminal or from a command file.
- Each error message contains a brief description of the error. When you enter an illegal command, the library generator indicates which word or parameter is erroneous.
- The library generator listing shows the commands executed, global symbols, and a summary of library generator activity.

INSTALLATION

This procedure describes how to copy the software for the 8080A/8085A assembler from the installation disc to your DOS/50 system disc. To complete this installation procedure you need the following items:

- an 8550 Microcomputer Development Lab;
- a DOS/50 Version 2 system disk with a write-enable tab over the write protect slot; and
- an 8080A/8085A assembler software installation disc.

You will need about 5 minutes to complete this installation procedure.

Start Up and Set the Date

Turn on your 8550 system, place your system disc in drive 0, and shut the drive door. When you see the > prompt on your system terminal, place your installation disc in drive 1 and shut the drive door.

Use the DAT command to set the date and time. For example, if it is 10:55 AM on Nov. 2, 1980, type:

```
> DAT 02-NOV-80/10:55:00 AM <CR>
```

The system will use this information when it sets the CREATION TIME attribute of each file copied to your system disc.

Check the Number of Free Files and Blocks

This procedure adds about 5 files, consisting of 100 blocks, to your system disc. Enter the L command to check the number of free files and free blocks on your system disc.

```
> L <CR>
FILENAME
```

list of files in your system volume directory

```
Files used    aaa
Free files    bbb  ◄─── must be at least 5
Free blocks   ccc  ◄─── must be at least 100
Bad blocks      0
```

If you have fewer than 5 free files or 100 free blocks, you must delete some files from your system disc or get a new system disc and start the installation procedure over again.

Install the Software (DOS/50 Version 2 Procedure)

To install the assembler software on your DOS/50 Version 2 system disk, simply type the filespec of the installation command file:

```
> /VOL/ASM.8080/INSTALL2 <CR>
```

If this command does not work, you must create the INSTALL2 installation command file:

[Step 1: Copy file **INSTALL** to **INSTALL2.**]

```
> COP /VOL/ASM.8080/INSTALL /VOL/ASM.8080/INSTALL2 <CR>
```

[Step 2: Use the command file converter (cfcv) program to convert INSTALL2 into the proper DOS/50 Version 2 format. Before converting, the cfcv program copies INSTALL2 to INSTALL2# for backup purposes.]

```
> cfcv /VOL/ASM.8080/INSTALL2 <CR>
```

[Step 3: Remove **INSTALL2#** from the directory.]

```
> DEL /VOL/ASM.8080/INSTALL2# <CR>
Delete /VOL/ASM.8080/INSTALL2#        ?y <CR>
```

[Step 4: Invoke the DOS/50 Editor, and prepend the **t,ON** command line to **INSTALL2**.]

```
> EDIT /VOL/ASM.8080/INSTALL2 <CR>

** EDIT VERSION x.x
*GET 99 <CR>
** EOF
*BEGIN <CR>
*During this installation procedure, one or more of the
*INPUT <CR>
INPUT:
t,ON <CR>
<CR>
*FILE <CR>
** END OF TEXT
** EOF

>
```

You may now install the assembler software on your **DOS/50 Version 2** disk by typing the filespec of the installation command file:

```
> /VOL/ASM.8080/INSTALL2 <CR>
```

Now proceed to the heading, "Operating System Response".

Install the Software (DOS/50 Version 1 Procedure)

To install the assembler software on a **DOS/50 Version 1** disk, type:

```
> /VOL/ASM.8080/INSTALL <CR>
```

Now proceed to the following discussion, "Operating System Response".

Operating System Response

The operating system responds with the following message:

```
* During this installation procedure, one or more of the
* following messages may appear.  IGNORE THESE MESSAGES:
*
*       Error 6E - Directory alteration invalid
*       Error 7E - Error in command execution
*       Error 1D - File not found
*
* If any OTHER error message appears, see your
* Users Manual for further instructions.
*
* If no other error message appears, you'll receive a
* message when the installation procedure is complete.
*
t,OFF
```

The installation process generates a number of error messages that do not affect the success of the installation. However, if any message **OTHER** than 6E, 7E, or 1D appears: check that you are using the correct discs, that a write-enable tab is present on your system disc, and that there are at least 100 free blocks and 5 free files on your system disc; then, begin the installation procedure again. If the error message appears again, copy down the error and contact your local Tektronix Field Office.

"t,OFF" is the first command in the installation command file and is displayed before it is executed. This command suppresses subsequent output to your system terminal (except error messages) until the installation command file finishes executing.

Within about 5 minutes, the installation command file will finish and your system terminal will display the following message:

```
*
* Your installation has been successfully completed.
>
```

Your software is now installed, and you can:

- remove your discs and turn off your 8550 system, or
- install more software on your system disc, or
- continue with the 8080A/8085A Assembler Demonstration Run that follows in this section. If you do so, you do not have to restart the system or reset the date.

You can install more than one assembler on your DOS/50 system disc. To tell the operating system which software to use at the time of assembly, use the SEL command. For example:

```
> SEL 8080 <CR>
```

selects the 8080A software.

NOTE

The 8080A/8085A assembler supports both the 8080A and 8085A emulators. SEL 8080 or SEL 8085 will specify the 8080A/8085A assembler software.

DEMONSTRATION RUN

Introduction

This Demonstration Run shows you how to enter, modify, assemble, link, and load a simple program and subroutine. This demonstration uses the 8080A/8085A assembler. If you have an assembler other than the 8080A/8085A, refer to the Assembler Specifics section of this manual for a Demonstration Run that is parallel to this one.

The purpose of this demonstration is to give you the basic information and experience you will need to begin using the assembler, linker, and library generator.

For your convenience, the sample program and subroutine are short and trivial. Only a few features of the assembler and linker are demonstrated, and the library generator is not discussed.

This Demonstration Run uses the following conventions:

1. Underlined—Underlined characters in a command line must be entered from your system terminal. Those characters not underlined are system output.
2. <CR>—Each command line ends with an end-of-line character. The end-of-line character for the 8550 is a carriage return (ASCII code 13). When a carriage return is to be entered, the symbol <CR> is used.

Preparation

To do this Demonstration Run you should have a basic understanding of the 8550 Microcomputer Development Lab and the DOS/50 Editor. If you need to review how the 8550 and its editor work, refer to the Learning Guide in the 8550 System Users Manual (DOS/50 Version 2) and the 8550 Editor Users Manual.

You will need about 60 minutes to complete this Demonstration Run.

Start up your 8550 system. Make sure your system disc has the 8080A/8085A assembler software installed and a write-enable tab over the write-protect slot. Insert the system disc into drive 0.

Use the DAT command to set the date and time. For example, if it is 11:05 AM on Nov. 3, 1980, enter:

```
> DAT 03-NOV-80/11:05:00 AM <CR>
```

Next, enter the USER command to establish ME as the owner of the files you will create:

```
> USER,,ME <CR>
```

Use the SEL command to tell DOS/50 to use the 8080A/8085A assembler software:

```
> SEL 8080 <CR>
```

(Notice that the A in 8080A is not included in the SEL command line.)

SEL 8085 also specifies the 8080A/8085A assembler software.

Examine your disc directory to make sure you have at least 25 blocks available for the files created during this demonstration:

```
> L <CR>
FILENAME
```

list of files in your system volume directory

```
Files used    aaa
Free files    bbb  ←——  must be at least 20
Free blocks   ccc  ←——  must be at least 25
Bad blocks     0
```

If there are **not** at least 25 blocks or 20 free files available on your system disc, you must make some room by copying some of your non-system files to another disc.

Enter the following commands to create an empty directory ASM.DEMO and make it the current directory:

```
> CREATE ASM.DEMO <CR>

> USER ASM.DEMO <CR>
```

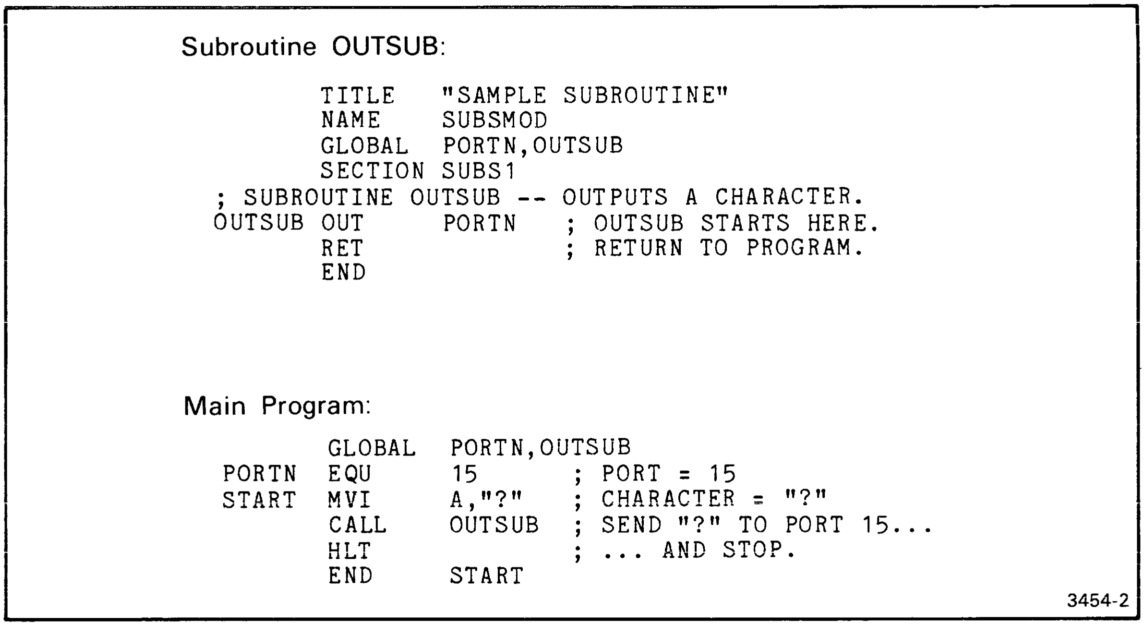
Examine the Sample Subroutine and Main Program

Figure 1-2 lists the subroutine and program you will enter, assemble, link, and load in this Demonstration Run.

The subroutine performs a trivial task: it outputs the ASCII character stored in the accumulator to the port whose number is specified by the symbol PORTN.

The main program stores a character in the accumulator, calls the subroutine to output the character, and then halts.

You can think of the subroutine as a carefully prepared component of a major programming project. The main program can be viewed as a quickly written test for the subroutine.



Subroutine OUTSUB:

```
        TITLE   "SAMPLE SUBROUTINE"
        NAME    SUBSMOD
        GLOBAL  PORTN,OUTSUB
        SECTION SUBS1
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
OUTSUB  OUT     PORTN   ; OUTSUB STARTS HERE.
        RET             ; RETURN TO PROGRAM.
        END
```

Main Program:

```
        GLOBAL  PORTN,OUTSUB
PORTN   EQU     15      ; PORT = 15
START   MVI     A,"?"   ; CHARACTER = "?"
        CALL    OUTSUB  ; SEND "?" TO PORT 15...
        HLT             ; ... AND STOP.
        END     START
```

3454-2

Fig. 1-2. Source code for the sample subroutine and program.

Subroutine OUTSUB outputs a single ASCII character to port number PORTN. The main program specifies a port number and a character and calls OUTSUB to output the character. The subroutine and main program are discussed in more detail later in this section.

Assembly Language Statements

An assembler source module is made up of assembly language statements. There are three types of assembly language statements:

- An **assembly language instruction** is translated by the assembler into an 8080A machine instruction.
- An **assembler directive** indicates a special action to be taken by the assembler. Assembler directives define data items, constants, and variables; provide information to the linker; control macros and conditional assembly; and specify options for the assembler and linker listings.
- A **macro invocation** is replaced by the statements of the macro it invokes. (Macro invocations are not discussed in this demonstration.)

Each assembly language statement has four fields. Each field may vary in width, and certain fields may be blank. However, the fields always occur in the following order:

1. The **label** field. The label field always begins in column 1 of the statement. The label allows the statement to be referenced by other statements. The label usually represents the address of the instruction or data item represented by the statement.
2. The **operation** field. The word in the operation field indicates the type of action to be taken by the assembler. The word may be an assembler directive, an 8080A mnemonic, or the name of a macro. If the word is an 8080A mnemonic, the assembler translates the statement into a machine instruction.
3. The **operand** field. The operand field completes the assembly language statement. Most assembler directives and 8080A instructions contain one or more operand expressions. The type and number of operands depend on the operation.

4. The **comment** field. Comments are used for program documentation only; they are ignored by the assembler. A semicolon (;) indicates that the remainder of the line is a comment. A comment may follow the operand field, or may begin with a semicolon in column 1 and take up an entire source line.

Explanation of the Subroutine Source Code

The following text explains each statement in the sample subroutine (shown in Fig. 1-2). The two statements preceding the END statement are 8080A instructions. The rest of the statements are assembler directives.

```
        TITLE   "SAMPLE SUBROUTINE"
```

The phrase "SAMPLE SUBROUTINE" will appear in the heading on each page of the assembler source listing.

```
        NAME    SUBSMOD
```

When the subroutine is assembled, the resulting object module will be named "SUBSMOD".

```
        GLOBAL  PORTN,OUTSUB
```

PORTN and OUTSUB are declared as global symbols, since each symbol is given a value in one module and referred to in another module. For example, OUTSUB is defined in the subroutine and referred to in the main program. PORTN is called an **unbound global** because it is not defined anywhere in this module. OUTSUB is a **bound global**.

```
        SECTION SUBS1
```

Each object module is composed of one or more sections. The linker treats each section as a separate unit: sections from the same module may be placed in different ends of memory. The one section in object module SUBSMOD will be called SUBS1. (If you were to add other sections to this source module, they might be called SUBS2, SUBS3, and so on.)

The assembler directives SECTION, COMMON, and RESERVE each declare a different type of section, and may also specify restrictions on the relocatability of the section. When no restriction is specified, the section is **byte-relocatable**; that is, the section may begin at any byte in memory. The Linker section of this manual contains an explanation of the five attributes of a section: name, section type, relocation type, size, and memory location.

```
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
```

This is a comment.

```
OUTSUB OUT      PORTN   ; OUTSUB STARTS HERE.
```

This 8080A instruction outputs the contents of the accumulator to port number PORTN. The symbol OUTSUB becomes defined as the address of this instruction, which is the first instruction in the subroutine. A program that contains the instruction CALL OUTSUB can execute this subroutine.

```
        RET             ; RETURN TO PROGRAM.
```

This 8080A instruction returns control to the calling program.

```
        END
```

This assembler directive marks the end of the source module.

Explanation of the Main Program Source Code

The following text explains each statement in the sample main program (shown in Fig. 1-2). The program contains three assembler directives (GLOBAL, EQU, and END) and three 8080A instructions (MVI, CALL, and HLT).

```
        GLOBAL  PORTN,OUTSUB
```

As in the subroutine, PORTN and OUTSUB are global symbols. However, in this module PORTN is a bound (defined) global while OUTSUB is an unbound (undefined) global. The GLOBAL statements allow the two modules to share the number of the port and the address of the subroutine.

```
PORTN   EQU     15      ; PORT = 15
```

This assembler directive assigns the value 15 to the symbol PORTN. "PORTN" becomes synonymous with the constant "15".

```
START   MVI     A,"?"   ; CHARACTER = "?"
```

This 8080A instruction stores the hexadecimal value 3F (the ASCII code for question mark) in the accumulator. This statement is given a label, "START", so the END statement may refer to it.

```
        CALL    OUTSUB  ; SEND "?" TO PORT 15...
```

This 8080A instruction transfers control to the instruction labeled OUTSUB in the subroutine module. The subroutine sends the question mark to port 15.

```
        HLT             ; ... AND STOP.
```

Control returns from the subroutine to this 8080A instruction. The HLT instruction halts program execution.

```
        END     START
```

This assembler directive terminates the source module and indicates that program execution should begin with the instruction labeled "START". START is called the **transfer address.** The transfer address is passed through the assembler and linker to the operating system LO (Load) and G (Go) commands.

Notice that this program source module does not contain a TITLE, NAME, or SECTION directive. The following default conditions result:

- No special title will appear in the page heading of the source listing.
- The object module will be called *NONAME*.

- The one section in *NONAME* will be given a default name, section type, and relocation type.

Naming Files

This Demonstration Run produces several files. To give each file a name that reflects its contents and importance, we will use the file naming standards described in the Files section of the 8550 (DOS/50 Version 2) System Users Manual:

- The first part of the file name is an optional descriptive name followed by a period.
- The last part of the file name is a 3- or 4-character identifier that reflects the file type.

The following files will be produced:

File Name	Description	How Created
SUB.ASM	SUBroutine ASseMbler source file	by you
SUB.OBJ	SUBroutine OBJect file	by assembler
SUB.ASML	SUBroutine ASseMbler List file	by assembler
PROG.ASM	PROGram ASseMbler source file	by you
PROG.OBJ	PROGram OBJect file	by assembler
PROG.ASML	PROGram ASseMbler Listing file	by assembler
LOAD	Program LOAD file	by linker
LNKL	Program LiNKer Listing file	by linker

Create the Subroutine Source File

The DOS/50 Editor helps create and modify source files. The Editor Users Manual contains a complete explanation of the editor.

How to Correct Typing Mistakes in the Editor

If you notice a mistake in your command line, you have two ways of correcting it before you enter a carriage return: delete the entire line and start again, or correct the characters one-by-one.

- To delete the entire line, press your terminal's ESC (escape) key once. You may then reenter the line.
- To delete characters one-by-one, press the BACKSPACE or RUBOUT key. Either key will backspace the cursor and erase the deleted character.

Start Editing

The EDIT command invokes the DOS/50 Editor. The file name in the EDIT command indicates the file to be edited. Enter the following line to begin the editing session that creates SUB.ASM, the subroutine source file:

```
> EDIT SUB.ASM <CR>

 ** EDIT VERSION x.x
 ** NEW FILE
*
```

The asterisk (*) is the editor prompt character. When you see the asterisk, you may enter the next editor command.

An assembly language program is easier to read if the statement fields are aligned as in Fig. 1-2. The editor has tab stops at columns 8, 16, 24, 32. 40, 48, 56, and 64, which are convenient for aligning assembly language text. For example, in Fig. 1-2, the operation field begins in column 8, the operand field begins in column 16, and the comment field begins in column 24.

Enter the following command line to declare the dollar sign as the editor tab character, and to enable the spaces created by the tab to be output:

```
*TAB $:XTABS ON <CR>
```

The editor will interpret every dollar sign you enter as a skip to the next tab stop.

Enter input mode and type in the subroutine. Be sure to misspell "GLOBAL" in the third line of text. This deliberate typing error will be used to illustrate features of the assembler and editor.

```
*INPUT <CR>
INPUT:
$TITLE$"SAMPLE SUBROUTINE" <CR>
$NAME$SUBSMOD <CR>
$GLOABL$PORTN,OUTSUB <CR>
$SECTION$SUBS1 <CR>
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.  <CR>
OUTSUB$OUT$PORTN$; OUTSUB STARTS HERE.  <CR>
$RET$$; RETURN TO PROGRAM.  <CR>
$END <CR>
<CR>
*
```

When you enter the carriage return on the empty line, input mode is terminated and the editor prompt (*) appears.

The text you entered is stored in the editor workspace. To display the workspace contents from beginning to end, enter the following command:

```
*TYPE B-E <CR>
        TITLE   "SAMPLE SUBROUTINE"
        NAME    SUBSMOD
        GLOABL  PORTN,OUTSUB
        SECTION SUBS1
; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
OUTSUB  OUT     PORTN   ; OUTSUB STARTS HERE
        RET             ; RETURN TO PROGRAM.
        END
*
```

Now enter the FILE command to copy the text in the workspace out to the new source file and end the editing session:

```
*FILE <CR>
 ** END OF TEXT

>
```

The system prompt (>) indicates that you are out of the editor and may enter another DOS/50 command.

Assemble the Subroutine and Examine Any Errors

The operating system command ASM invokes the assembler and specifies the source file(s) to be assembled and the object and listing files to be produced. The ASM command has the following format:

ASM, objfile, lisfile, soufile [, soufile] . . .

objfile—filespec of object file to be produced
lisfile—filespec of listing file to be produced
soufile—filespec(s) of source file(s) to be assembled

To scan source file SUB.ASM for errors, enter the following command:

```
> ASM,,,SUB.ASM <CR>
```

Omitting the filespecs of the object and listing files has two advantages:

1. The assembler runs faster because it produces no object code or listing.
2. The ASM command line is shorter.

You may want to omit these filespecs from your ASM command line whenever you suspect that your source code contains errors.

The assembler responds as follows on your system terminal:

```
Tektronix  8080/8085 ASM Vx.x
**** Pass 2
00003 0000 000000             GLOABL   PORTN,OUTSUB
***** ERROR 039: Invalid operation code
00006 0000 D300        OUTSUB OUT      PORTN    ; OUTSUB STARTS HERE.
***** ERROR 074: Undefined symbol
    8 Source Lines       8 Assembled Lines   47415 Bytes available
    2 ERRORS             2 UNDEFINED SYMBOLS

>
```

The assembler's response can be interpreted as follows:

```
Tektronix  8080/8085 ASM Vx.x
```

The assembler announces itself as it begins executing. The assembler reads through your source file twice. The first time through (Pass 1), the assembler makes a list of symbols that appear in the source code and tries to assign an address or other value to each symbol.

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

The assembler begins its second pass through your source file. During Pass 2, the assembler produces the object and listing files and displays error messages and statistics.

```
00003 0000 000000            GLOABL  PORTN,OUTSUB
***** ERROR 039: Invalid operation code
```

The assembler cannot translate the above statement because "GLOABL" is not an 8080A mnemonic, an assembler directive word, or the name of a macro. The erroneous source line and the error message would appear in the listing (if any) just as they appear on the system terminal. The three numbers to the left of the statement will be explained when you examine an assembler listing later in this demonstration run.

```
00006 0000 D300        OUTSUB OUT     PORTN    ; OUTSUB STARTS HERE.
***** ERROR 074: Undefined symbol
```

Because the assembler did not understand the GLOBAL statement, it does not know that PORTN is a global symbol. The assembler expects PORTN to be defined in this module.

```
     8 Source Lines       8 Assembled Lines    47718 Bytes available
     2 ERRORS             2 UNDEFINED SYMBOLS
```

These lines summarize the assembler's activities. There are eight lines of code in your source file. The number of assembled lines differs from the number of source lines only in programs that contain macros or conditional assembly.

The "Bytes available" message indicates the amount of Program Memory not used by the assembler. If the "Bytes available" figure is ever less than 1000 or so, you may need to divide your source module into smaller modules before you add any more statements.

The two errors, already discussed, produced the two undefined symbols GLOABL and PORTN.

Correct the Error in the Subroutine Source Code

Both errors detected by the assembler arose from the misspelling of "GLOBAL" in line 3 of the source file, SUB.ASM. Invoke the editor so that you may correct the misspelling:

```
> EDIT SUB.ASM <CR>

 ** EDIT VERSION x.x
*
```

The editor command GET brings text into the workspace from the file being edited. Specify a large number of lines (99) to assure that the entire file is brought into the workspace:

```
*GET 99 <CR>
 ** EOF
*
```

The message **EOF** indicates that the end of the input file has been reached.

Enter the following command line to find the line that contains the misspelling.

```
*BEGIN.:FIND /GLO/ <CR>
       GLOABL  PORTN,OUTSUB
```

The BEGIN command moves the workspace pointer to line 1. Starting at that line, the FIND command searches for the character string "GLO" and moves the pointer to the first line that contains the string.

Now the workspace pointer is at the line you want to modify. Use the SUBSTITUTE command to reverse the letters "A" and "B" in "GLOABL":

```
*SUB /AB/BA/ <CR>
      GLOBAL   PORTN,OUTSUB
```

The modified line is displayed.

As before, the FILE command copies the edited source code to the source file and closes the editing session:

```
*FILE <CR>
 ** END OF TEXT
 ** EOF

>
```

Re-Assemble the Subroutine

Enter the following command to create an object file (SUB.OBJ) and an assembler listing file (SUB.ASML) from the subroutine source file:

```
> ASM SUB.OBJ SUB.ASML SUB.ASM <CR>

Tektronix  8080/8085 ASM Vx.x
**** Pass 2
    8 Source Lines       8 Assembled Lines   47401 Bytes available
            >>> No assembly errors detected <<<

>
```

This time the assembler finds no errors.

Examine the Subroutine Listing

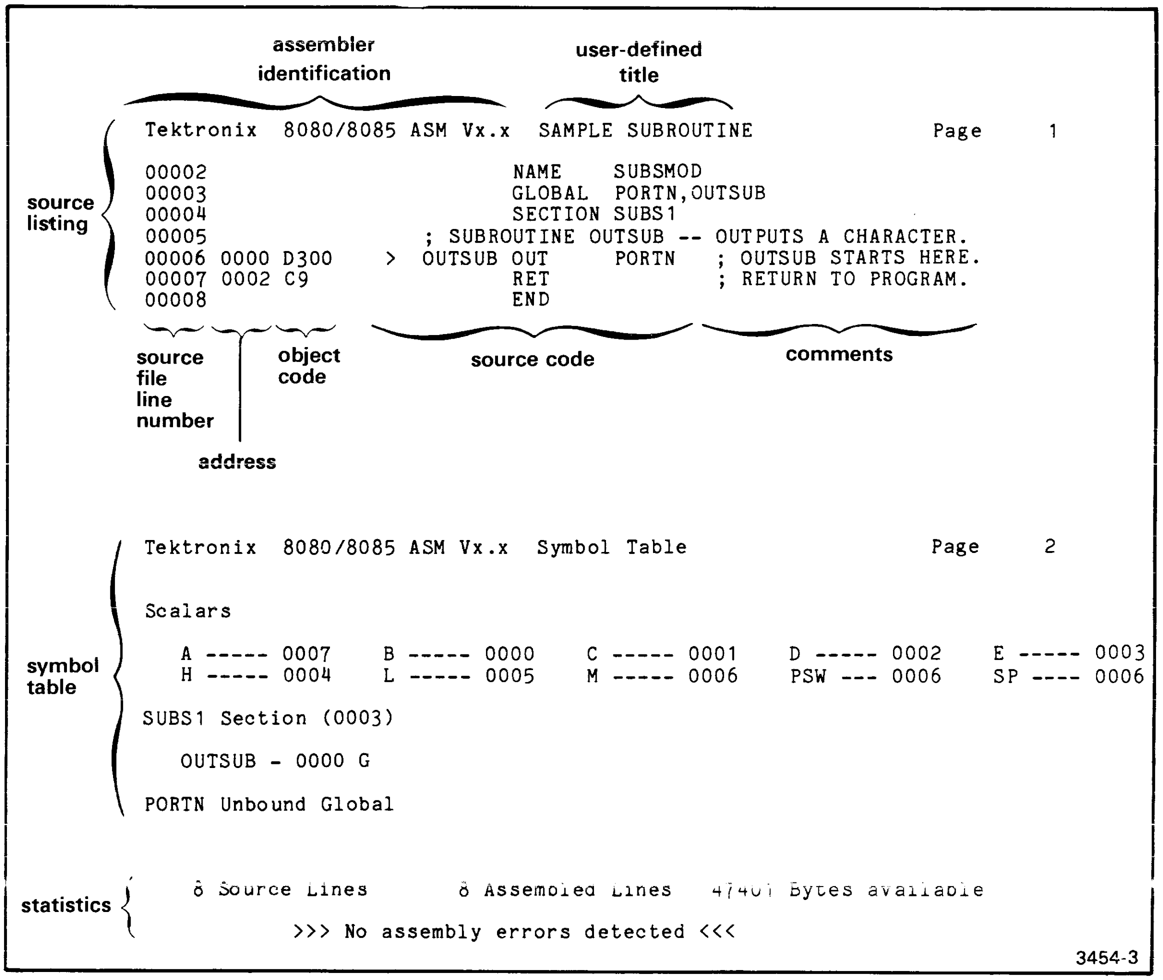
In order to examine the assembler listing stored on file SUB.ASML, copy the file to your line printer:

```
> COP SUB.ASML LPT <CR>
```

If you have no line printer, enter the following command to list the file on your system terminal. (Remember that you may use CTRL-S to suspend display and CTRL-Q to resume display on a CRT terminal.)

```
> CON SUB.ASML <CR>
```

Figure 1-3 shows the listing of the sample subroutine.



```
Tektronix  8080/8085 ASM Vx.x  SAMPLE SUBROUTINE              Page      1

00002                        NAME    SUBSMOD
00003                        GLOBAL  PORTN,OUTSUB
00004                        SECTION SUBS1
00005                 ; SUBROUTINE OUTSUB -- OUTPUTS A CHARACTER.
00006 0000 D300     > OUTSUB OUT     PORTN    ; OUTSUB STARTS HERE.
00007 0002 C9                RET              ; RETURN TO PROGRAM.
00008                        END


Tektronix  8080/8085 ASM Vx.x  Symbol Table                   Page      2

Scalars

   A ----- 0007   B ----- 0000   C ----- 0001   D ----- 0002   E ----- 0003
   H ----- 0004   L ----- 0005   M ----- 0006   PSW --- 0006   SP ---- 0006

SUBS1 Section (0003)

   OUTSUB - 0000 G

PORTN Unbound Global


    8 Source Lines        8 Assembled Lines    47401 Bytes available

            >>> No assembly errors detected <<<
```

Fig. 1-3. Assembler listing for the sample subroutine.

The command ASM SUB.OBJ SUB.ASML SUB.ASM produces this listing file from the subroutine source file. The command COP SUB.ASML LPT copies the listing file to the line printer.

Every assembler listing has two parts: the source listing and the symbol table. Each page of the listing begins with a standard page heading.

The Source Listing

Page 1 of your assembler listing contains the source listing. The heading includes the words "SAMPLE SUBROUTINE", which you supplied with the TITLE directive.

Each line of the source listing contains the following information:

1. the line number (decimal);
2. the memory location (hexadecimal) of the object code generated (if any);
3. the assembled object code (hexadecimal);
4. a relocation indicator (>) if the object code may be adjusted by the linker;
5. a text substitution indicator (+) if the assembler has modified the source statement (this demonstration gives no examples of text substitution);
6. the source statement.

If any statement contains an error, the appropriate error message appears directly after the statement.

Examine each line of your source listing:

- Line 1 (the TITLE directive) is not printed because it is a listing control directive.
- Lines 2, 3, 4, and 8 are assembler directives that produce no object code. The information they provide is stored in special areas of the object module.
- Line 5 is a comment.
- Lines 6 and 7 are 8080A assembly language instructions:
  - The 8080A instruction OUT PORTN produces the two-byte machine instruction D300. D3 is the hexadecimal operation code for the OUT instruction. The dummy value 00 will be used for the port number until the linker supplies a value for PORTN. The machine instruction D300 is stored in bytes 0000 and 0001 of section SUBS1.
  - The 8080A instruction RET produces the one-byte machine instruction C9, which is stored in byte 0002 of section SUBS1.

The Symbol Table

Page 2 of your listing contains the symbol table, which indicates the value and type of each symbol in your source code.

The assembler symbol table is divided into the following categories:

1. Strings and macros
2. Scalars (numeric values other than addresses; includes undefined symbols)
3. Sections (and addresses within each section)
4. Unbound globals

Examine the symbol table in your listing:

1. The strings and macros table is omitted, since the sample subroutine uses neither string variables nor macros.
2. The scalars table lists every scalar in the symbol list and the value associated with each scalar. The sample subroutine defines no scalars, but the names of the 8080A registers are pre-defined symbols and thus always appear in the symbol list.
3. SUBS1 is the only section in the sample subroutine. The line

   ```
   SUBS1 Section (0003)
   ```

   tells you the following attributes of section SUBS1:

   - its name: SUBS1;
   - its section type: SECTION (as opposed to COMMON or RESERVE);
   - its relocation type: byte-relocatable (the default relocation type is implied when no other relocation type is specified);
   - its length: 3 bytes

   OUTSUB has the value 0000 because OUTSUB is the address of the first byte in section SUBS1. The letter "G" indicates that OUTSUB is a global symbol.
4. PORTN is the only unbound (undefined) global symbol in the subroutine.

The statistics at the bottom of the symbol table are the same statistics that appeared on the system terminal when the assembler finished execution.

When there are errors in your source code, the two most useful parts of your listing are the source listing and the scalars table. The source listing contains the error messages and shows the erroneous lines in context with the rest of the program. The scalars table flags undefined symbols with the value "****".

Create the Main Program Source File

Now that you have created, corrected, and assembled the sample subroutine, it is time to create the main program that uses the subroutine. Enter the following command to begin the editing session that creates the main program source file, PROG.ASM:

```
> EDIT PROG.ASM <CR>

 ** EDIT VERSION x.x
 ** NEW FILE
*
```

Declare the editor tab character and type in the source code, just as you did for the subroutine. (This time, however, don't include any typing errors.)

```
*TAB $:XTABS ON <CR>
*INPUT <CR>
INPUT:
$GLOBAL$PORTN,OUTSUB <CR>
PORTN$EQU$15$; PORT = 15 <CR>
START$MVI$A,"?"$; CHARACTER = "?"  <CR>
$CALL$OUTSUB$; SEND "?" TO PORT 15... <CR>
$HLT$$; ... AND STOP.  <CR>
$END$START <CR>
<CR>
*
```

The editor interprets every dollar sign as a skip to the next tab stop. Inspect the text you have entered to be sure there are no errors:

```
*TYPE B-E <CR>
       GLOBAL  PORTN,OUTSUB
PORTN  EQU     15      ; PORT = 15
START  MVI     A,"?"   ; CHARACTER = "?"
       CALL    OUTSUB  ; SEND "?"  TO PORT 15 ...
       HLT             ; ... AND STOP.
       END     START
*
```

Enter the FILE command to save the text onto the source file and return to DOS/50:

```
*FILE <CR>
 ** END OF TEXT

>
```

Assemble the Main Program

Enter the following command line to create an object file (PROG.OBJ) and a listing file (PROG.ASML) from the main program source file:

```
> ASM PROG.OBJ PROG.ASML PROG.ASM <CR>

Tektronix  8080/8085 ASM Vx.x
**** Pass 2
    6 Source Lines       6 Assembled Lines   47424 Bytes available
            >>> No assembly errors detected <<<

>
```

The main program contains no errors.

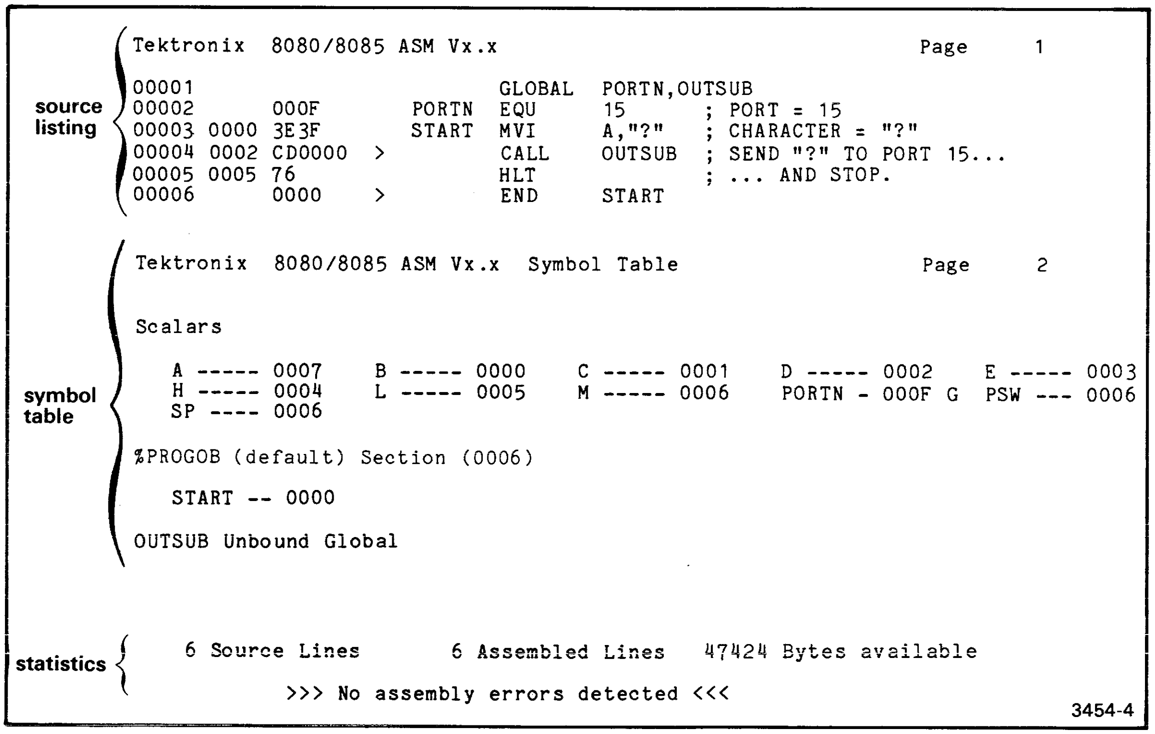
Examine the Main Program Listing

Copy the assembler listing to the line printer or to the system terminal:

```
> COP PROG.ASML LPT <CR>
```

or

```
> CON PROG.ASML <CR>
```



source listing
symbol table
statistics
3454-4

```
Tektronix  8080/8085 ASM Vx.x                                   Page     1

00001                        GLOBAL  PORTN,OUTSUB
00002         000F     PORTN  EQU     15       ; PORT = 15
00003 0000 3E3F       START  MVI     A,"?"    ; CHARACTER = "?"
00004 0002 CD0000  >          CALL    OUTSUB   ; SEND "?" TO PORT 15...
00005 0005 76                HLT               ; ... AND STOP.
00006         0000   >          END     START

Tektronix  8080/8085 ASM Vx.x  Symbol Table                     Page     2

Scalars

   A ----- 0007    B ----- 0000    C ----- 0001    D ----- 0002    E ----- 0003
   H ----- 0004    L ----- 0005    M ----- 0006    PORTN - 000F G  PSW --- 0006
   SP ---- 0006

%PROGOB (default) Section (0006)

   START -- 0000

OUTSUB Unbound Global


    6 Source Lines      6 Assembled Lines    47424 Bytes available

              >>> No assembly errors detected <<<
```

Fig. 1-4. Assembler listing for the sample main program.

The command ASM PROG.OBJ PROG.ASML PROG.ASM produces this listing file from the main program source file. The command COP PROG.ASML LPT copies the listing file to the line printer.

Compare the listing of the sample main program (Fig. 1-4) with the listing of the sample subroutine (Fig. 1-3).

The Source Listing

Page 1 of your assembler listing contains the source listing. Notice that there is no user-defined title for the program listing: the source code contains no TITLE directive.

Examine each line of the program source listing.

1. As in the subroutine, the GLOBAL statement produces no object code.

2. The EQU statement assigns the value 15 (000F hexadecimal) to the symbol PORTN. The symbol PORTN and its value are stored in the global symbol block of the program object module. At link time, the value of PORTN will be substituted into the OUT instruction in the subroutine.
3. The 8080A assembly language instruction MVI A,"?" generates the machine instruction 3E3F. 3E is the operation code for MVI A and 3F is the ASCII code for the question mark. The machine instruction 3E3F is stored in bytes 0000 and 0001 of the main program.
4. The 8080A assembly language instruction CALL OUTSUB generates the machine instruction CD0000 in bytes 0002 through 0004. CD is the operation code for the CALL instruction. 0000 is a dummy value: the address of OUTSUB will be provided at link time.
5. The 8080A assembly language instruction HLT produces the one-byte machine instruction 76 in byte 0005 of the main program.
6. The END statement specifies that the transfer address is 0000, the address of the MVI instruction. The transfer address will be adjusted if this section of object code is not loaded at the beginning of memory.

The Symbol Table

1. The strings and macros table is again omitted because it is empty.
2. The scalars table lists the usual pre-defined scalars, plus the symbol PORTN. The value of PORTN is 000F hexadecimal. The "G" indicates that PORTN is a global symbol.
3. Because the main program source code contains no SECTION directive, the section produced by this assembler run is given the following default attributes:
   - name: %PROGOB (derived from the name of the object file);
   - section type: SECTION;
   - relocation type: byte-relocatable.

   Section %PROGOB contains six bytes of code. START is the address of the first byte of the section.
4. OUTSUB is the only unbound (undefined) global symbol in the main program.

The statistics at the bottom of the symbol table are the same statistics that appeared on the system terminal when the assembler finished execution.

Link the Object Modules

Now both the subroutine and the main program have been translated into machine language. In order for the subroutine and main program object modules to communicate with each other, they must be linked. The linker performs the following tasks in creating a load file of executable object code:

- It finds a block of memory for each section in the specified object files.
- It adjusts addresses to reflect relocation of sections.
- It provides values for unbound globals.

Enter the following command to create a load file (LOAD) and a linker listing file (LNKL) from your two object files:

```
> LINK LOAD LNKL PROG.OBJ SUB.OBJ <CR>
```

The linker responds as follows:

```
8550 LINKER Vx.x

    NO ERRORS      NO UNDEFINED SYMBOLS
    2 MODULES      2  SECTIONS
    TRANSFER ADDRESS IS 0000

>
```

Examine the Linker Listing

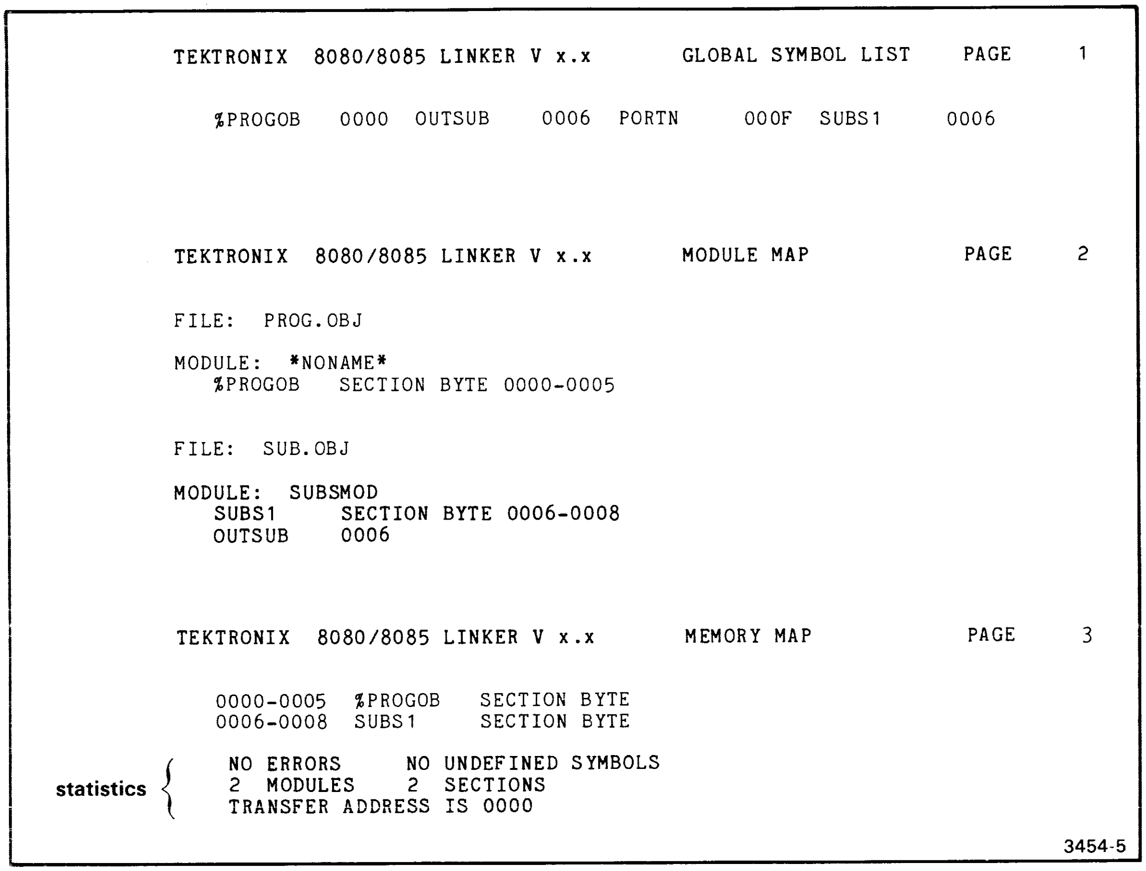
Copy the linker listing file to the line printer or system terminal:

```
> COP LNKL LPT <CR>
```

or

```
> CON LNKL <CR>
```

Figure 1-5 shows the linker listing.



```
TEKTRONIX  8080/8085 LINKER V x.x       GLOBAL SYMBOL LIST     PAGE      1

   %PROGOB    0000  OUTSUB    0006  PORTN      000F  SUBS1      0006



TEKTRONIX  8080/8085 LINKER V x.x       MODULE MAP             PAGE      2

FILE:  PROG.OBJ

MODULE:  *NONAME*
   %PROGOB   SECTION BYTE 0000-0005

FILE:  SUB.OBJ

MODULE:  SUBSMOD
   SUBS1     SECTION BYTE 0006-0008
   OUTSUB    0006


TEKTRONIX  8080/8085 LINKER V x.x       MEMORY MAP             PAGE      3

   0000-0005  %PROGOB    SECTION BYTE
   0006-0008  SUBS1      SECTION BYTE

    NO ERRORS      NO UNDEFINED SYMBOLS
    2  MODULES     2  SECTIONS
    TRANSFER ADDRESS IS 0000
```

Fig. 1-5. Linker listing.

The command LINK LOAD LNKL PROG.OBJ SUB.OBJ produces this linker listing file. The command COP LNKL LPT copies the listing file to the line printer.

The linker listing contains three parts:

1. The **global symbol list** (page 1 of your listing) lists the value assigned to each global symbol. The name and starting address of each section are included. Undefined globals are flagged with the value "****".
2. The **module map** (page 2) provides the following information for each object module being linked:
   - the name of the object file or library file supplying the object module;
   - the name and attributes of each section in the module. Any entry points (addresses declared as global symbols) for each section are also listed.

   The module map allows you to verify that each section of your program has been assigned a place in memory.
3. The **memory map** (page 3) lists the sections in the order they occur in memory. Conflicting (overlapping) memory allocations are indicated with an asterisk (*).

   Linker statistics appear at the bottom of the memory map.

An optional feature of the linker listing, the internal symbol list, is useful for program debugging. The internal symbol list is not demonstrated here but is discussed in the Linker section.

The Memory Map

The memory map (page 3 of your listing) provides the most concise summary of the load file produced by the linker.

The memory map shows that bytes 0000 through 0005 of memory will contain section %PROGOB (the main program) and that bytes 0006 through 0008 will contain section SUBS1 (the subroutine).

The memory map also gives the section type (SECTION) and relocation type (byte-relocatable) for each section.

Notice that the transfer address remains unchanged because the section containing the transfer address is located at the beginning of memory.

The Module Map

The module map (page 2) shows much the same information as the memory map. The module map, however, reports the sections by module rather than by memory location.

The first object file, PROG.OBJ, contains the object module called *NONAME*. (Recall that the main program source code contains no NAME directive.) The main program consists of the single section %PROGOB, whose attributes you already know from the memory map.

The second object file, SUB.OBJ, contains the subroutine object module, SUBSMOD. SUBSMOD consists of the single section SUBS1. The single entry point to SUBS1 is OUTSUB, whose adjusted address (after relocation) is 0006.

The Global Symbols List

The global symbols list (page 1) shows the two symbols declared in GLOBAL statements (OUTSUB and PORTN) and the two section names (%PROGOB and SUBS1).

Load the Executable Object Code into Memory

Before you load the object code, use the DOS/50 command F to fill the beginning of program memory with zeros. Later, when you examine memory, the zeros make it easy to identify the end of your code. Enter the following command to fill memory locations 0000 through 000F with zeros:

```
> F 0 0F 00 <CR>
```

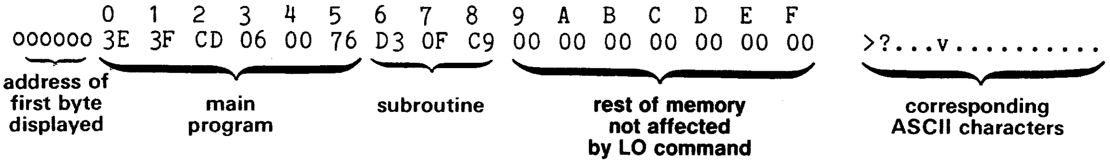
Now copy the executable object code from the load file into program memory:

```
> LO <LOAD <CR>
```

Bytes 0000 through 0008 of program memory now contain the nine bytes of machine language that form the executable program.

The DOS/50 command D displays the contents of a specified section of memory. Each byte is displayed as a two-digit hexadecimal number and as the ASCII character it represents (if any). Enter the following command to display the contents of memory locations 0000 through 000F:

```
> D 0 0F <CR>
```



```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
000000 3E 3F CD 06 00 76 D3 0F C9 00 00 00 00 00 00 00    >?...v..........
```

address of first byte displayed
main program
subroutine
rest of memory not affected by LO command
corresponding ASCII characters

Compare the relocatable object code produced by the assembler with the executable object code produced by the linker. (The addresses and object bytes adjusted by the linker are underlined.)

RELOCATABLE OBJECT CODE (from assembler listings)			EXECUTABLE OBJECT CODE (from DISPLAY output)		
Address	Object Code	Source Code	Address	Object Code	Source Code
0000	3E3F	MVI A,"?"	0000	3E3F	MVI A,"?"
0002	CD0000	CALL OUTSUB	0002	CD0600	CALL OUTSUB
0005	76	HLT	0005	76	HLT
0000	D300	OUT PORTN	0006	D30F	OUT PORTN
0002	C9	RET	0008	C9	RET

Note the adjustments made by the linker:

- The subroutine is relocated from byte 0000 to byte 0006.
- The address of the subroutine is substituted into the CALL instruction.
- The port number is substituted into the OUT instruction.

Enter the following command to reestablish the system volume as the current directory. (Enter your system volume name in place of **sysvol**.)

```
> USER /VOL/sysvol <CR>
```

"sysvol" represents the name of your system volume

Summary of Demonstration Run

Enter the L command to list the files you have created:

```
    > L ASM.DEMO <CR>

FILENAME

SUB.ASM#
SUB.ASM
SUB.OBJ
SUB.ASML
PROG.ASM
PROG.OBJ
PROG.ASML
LOAD
LNKL

FILES USED                   aaa
FREE FILES                   bbb
FREE BLOCKS                  ccc
BAD BLOCKS                     0
```

Recall the eight files you have created in this Demonstration Run:

- the two source files (SUB.ASM and PROG.ASM) you created using the editor;
- the two object files (SUB.OBJ and PROG.OBJ) and the two listing files (SUB.ASML and PROG.ASML) generated by the assembler;
- the load file (LOAD) and the listing file (LNKL) generated by the linker.

When you corrected the misspelling of GLOBAL in your source file, the editor retained the original file SUB.ASM as a backup file named SUB.ASM#.

You have now finished the Demonstration Run. It emphasized how to:

- create a source file, using the editor;
- create an object file from a source file, using the assembler;
- create a load file from object files, using the linker;
- copy the load file into memory, using the LOAD command;
- interpret listings generated by the assembler and linker.

The Demonstration Run in the Learning Guide of the 8550 System Users Manual (DOS/50 Version 2) shows you how to execute and monitor the program you have loaded into memory.

```
> DEL -N ASM. DEMO/* ASM.DEMO <CR>
```

Using This Manual With DOS/50 Version 1

To convert this manual for use with DOS/50 Version 1, you must perform the following steps:

1. Throughout this manual, change all instances of the "CON" command to "COPY", so that the manual will reflect DOS/50 Version 1 syntax.

2. In the "Summary of Demonstration Run", earlier in this section, change the first operating system command line to:

```
> LDIR /VOL/sysvol/ASM.DEMO <CR>
```

name of your system volume

3. At the end of the "Summary of Demonstration Run", change the operating system command line to:

```
> DELETE:N /VOL/sysvol/ASM.DEMO/*,/VOL/sysvol/ASM.DEMO <CR>
```

name of your system volume

4. Throughout this manual, change all references of "Version 2" to "Version 1".

8550 System Users Manual (DOS/50 Version 2). Describes how to use the 8550 Microcomputer Development Lab and its operating system, DOS/50 Version 2.

FOR CONTINUED LEARNING

This Learning Guide explained the basic concepts needed to use the assembler and linker. These same concepts will help you learn to use the library generator. For a more detailed explanation of how to use these system programs, refer to the following sections:

Section 2, Procedures. Gives you step-by-step instructions for accomplishing common tasks in assembling, linking, and library maintenance.

Section 3, Assembler Introduction. Shows the use of the operating system command ASM. Reviews the notational conventions used throughout this manual. Explains the assembler listing in more detail.

Section 4, Assembly Language Elements. Describes the fundamental elements of an assembly language statement. Gives rules for creating symbols, constants, and expressions. Describes special characters and assembler functions.

Section 5, Assembler Directives. Describes the function and use of each assembler directive. Each description is accompanied by one or more examples. Directives are arranged in alphabetical order.

Section 6, Macros. Shows how to create and use assembler macros. Demonstrates the macro features of the TEKTRONIX Assembler.

Section 7, The Linker. Describes the function and use of the operating system command LINK, and of each command in the linker subsystem. Explains the linker listing in more detail.

Section 8, The Library Generator. Describes the function and use of the operating system command LIBGEN, and of each command in the library generator subsystem. Explains the library generator listing.

Section 9, Programming Examples. Demonstrates and explains useful applications of the assembler, linker, and library generator.

Section 10, Tables. Summarizes reference information in tabular form.

Section 11, Technical Notes. Provides information on special applications of the assembler, linker, and library generator.

Section 12, Assembler Specifics. Provides information that varies with each microprocessor: registers, instruction sets, special error messages, etc. Section 12 also contains Demonstration Runs for microprocessors other than the 8080A. An Irregularities paragraph for each microprocessor lists exceptions to the standard reference material in this manual.

Section 13, Error Messages. Lists the error messages for the assembler, linker, and library generator. Each error message is accompanied by a description of the problem and possible solutions.

Section 14, Glossary. Defines special terms used in this manual.

8550 System Users Manual (DOS/50 Version 2). Describes how to use the 8550 Microcomputer Development Lab and its operating system, DOS/50 Version 2.

8550 Editor Users Manual. Describes how to use the 8550 Microcomputer Development Lab Editor.

Section 2
PROCEDURES

Page

Section 2

PROCEDURES

INTRODUCTION

In the previous section, the Learning Guide, you were presented with the basic concepts of the Tektronix Assembler and Linker. In this section, Procedures, you are shown procedures for using the assembler, the linker, and the library generator.

Each procedure in this section is simply a series of one or more command entries or actions for you to perform. Most of the procedures contain parameters (places for values) that you will supply when you perform the procedure.

Each procedure is presented in the following format:

Description: A summary of the operation(s) performed by the procedure.

Procedure: The information entered or displayed at the system terminal. The following conventions are used in the procedure description:

Underlined: The character sequence that you will enter. The sequence may consist of the exact characters to be entered, or parameters for which you must substitute your values.

No underline: A character sequence that is displayed by the operating system.

UPPERCASE: An exact character sequence; if these characters are underlined, enter them exactly as shown. (If they are not underlined, you will see these characters displayed by the operating system.)

lowercase: A parameter for which you will supply a value when you perform the procedure.

(Parentheses): A comment, or an action for you to perform at the indicated time.

Parameters: The filespecs or options that you provide.

Comments: The operating limits and options fo, this procedure.

Examples: One or more demonstrations of the correct entry format.

See Also: Cross-references to related procedures.

For a full description of any given command, refer to Assembler Directives, Linker, or Library Generator section in this manual, as appropriate.

ASSEMBLING YOUR PROGRAM

Invoking the Assembler

Description: This procedure generates an object file (in machine language) from an assembly language source file.

Procedure:

```
> ASM object listing source
```

Parameters: **object**—The filespec of the resulting object code.

listing—The filespec of the listing file or device.

source—The filespec of the assembly source code for the program.

Examples:

```
> ASM OBJ ASML ASM
```

This command line assembles the source code present in the file ASM. The resulting object code is placed in the file OBJ. The assembler also produces a program listing and a list of defined symbols. This listing is placed in the file ASML. All the files in this example reside in the current directory.

See also:

- Invoking the Linker
- Displaying Internal Symbols in the Linker Listing

Combining Source Files During Assembly

Description: This procedure assembles code from several source files into a single object module.

Procedure:

```
> ASM object listing asrc bsrc csrc
```

Parameters: **object**—The filespec of the resulting object code.

listing—The filespec of the listing file or device.

asrc—The filespec of the first source file to be assembled.

bsrc—The filespec of the second source file to be assembled.

csrc—The filespec of the third source file to be assembled.

Comments: The command line assembles the source files in left-to-right order. An END statement should appear only in the last source file to be assembled (in this case, **csrc**).

Example:

```
> ASM OBJ LPT A.ASM B.ASM C.ASM
```

The three source files are assembled left-to-right: A.ASM, then B.ASM, and then C.ASM. The object code is placed in the file OBJ. All of the files reside in the current directory. The program listing is output on the line printer (LPT).

See also:

- Invoking the Assembler

Displaying Internal Symbols in the Linker Listing

Description: This procedure adds the internal symbol list to the linker listing. The internal symbol list contains all of the symbols in the source file and their final values. These internal symbols include: scalars, section labels, and labels for unbound globals. This listing is useful for debugging high-level and assembly language programs.

Procedure: (Invoke the editor and include the LIST DBG option at the beginning of the assembler source file.)

```
> EDIT yourfile

 ** EDIT VER X.X
*INPUT
INPUT:
tLISTtDBG <cr>
<cr>
```

(t represents a tab or control-I key)

```
*FILE

*EDIT* EOJ
```

(Assemble the **yourfile** source file)

```
> ASM object,,yourfile
```

(Link the assembled **object** file)

```
> LINK load listing object
```

(The **load** may now be LOADed into program memory. The linker listing will show the memory addresses and the values of all symbols used in the program.)

Parameters:

object—The filespec of the assembled object code.

yourfile—The filespec of the assembly source file that will have the LIST DBG option added.

load—The filespec of the resulting load file.

listing—The filespec of the listing file or device. This listing includes a global symbol list and optional internal symbol or map listings. Refer to the Linker section of this manual for more information on the listing options.

Comments:

When you insert this listing control directive at the beginning of your assembly code the linker will list the values assigned to the internal symbol table. You may then examine or modify the program by using the linker listing to find the memory locations of symbols.

See also:

- Invoking the Assembler
- Invoking the Linker (Simple Invocation)

LINKING YOUR PROGRAM

Invoking the Linker (Simple Invocation)

Description:

This procedure converts the contents of object files (produced by the assembler) into a single file that is suitable for loading.

Procedure:

```
> LINK load listing object LIB(library)
```

Parameters:

load—The filespec of the resulting load file.

listing—The filespec of the listing file or device. This listing includes a global symbol list and optional internal symbol or map listings. Refer to the Linker section of this manual for more information on the listing options.

object—The filespec of the object file from which the load file is generated.

library—The filespec of a library file to be linked. You may omit this parameter.

Comments:

Additional object files may be linked by entering additional **object** parameters.

Examples:

```
> LINK LOAD LNKL OBJ LIB(YOUR.LIB)
```

This command uses the object code stored in the file OBJ and any object modules selected from the YOUR.LIB library file to generate a load file LOAD. The linker also produces a listing file LNKL. All files are located in the current directory.

```
> LINK LOAD LNKL HER.OBJ HIS.OBJ ITS.OBJ
```

The object code stored in the file HER.OBJ is linked first, then the file HIS.OBJ, then last of all the file ITS.OBJ. The resulting load file is named LOAD. The linker also produces a listing file named LNKL. All files are located in the current directory.

See also:

- Invoking the Assembler
- Invoking the Linker (Interactive Invocation)
- Displaying Internal Symbols in the Linker Listing

Invoking the Linker (Interactive Invocation)

Description: This procedure converts the contents of object files (produced by the assembler) into a single file that is suitable for loading. This form of invocation also permits a series of additional commands to be given to the linker.

Procedure:

```
> LINK
*LOG
*MAP
*LOAD load
*LIST listing
*LINK object
*LINK LIB(library)
*  (Enter one linker command per line)
            .
            .
            .
*END
```

Parameters:

load—The filespec of the resulting load file.

listing—The filespec of the listing file or device. This listing includes a global symbol list and optional internal symbol or map listings. Refer to the Linker section of this manual for more information on the listing options.

object—The filespec of the object file from which the load file is generated.

library—The filespec of a library file to be linked.

Comments:

This form of linker invocation allows you to enter commands line-by-line to specify: the listing format, the output file, the listing device or file, the input file, and several other options. These optional commands LOCATE sections, provide TRANSFER addresses for modules, and DEFINE values for symbols. If you wish to link more than one file, repeat the LINK command on the following line with the additional file as its parameter. The END command begins linker execution. For more information about linker invocation, see the Linker section of this manual.

Example:

```
> LINK
*LOG
*MAP
*LOAD LOAD
*LIST LPT
*LINK MY.OBJ
*LINK YR.OBJ
*LINK LIB(COMMON.LIB)
*DEFINE INITSP=8000
*END
```

The object code stored in the two files MY.OBJ and YR.OBJ is linked with selected object modules from the library file COMMON.LIB. The global symbol INITSP is assigned a hexadecimal value of 8000 and the linked body of code is placed in a load file named LOAD. The line printer (LPT) records the linker commands, the values of symbols, and the locations of sections.

```
> LINK
*LOG
*MAP
*LOAD OUT.LOAD
*LIST LST.LNKL
*LINK IN.OBJ
*LOCATE SECTNAME,BASE(0)
*END
```

The input file IN.OBJ contains a section named SECTNAM. The BASE attribute of the LOCATE command positions the starting address of this section at address 0. The resulting load file is named OUT.LOAD. The list file LST.LNKL records the linker commands, the values of symbols, and the locations of sections.

See also:

- Assigning Object Code to an Address Range
- Reserving an Area of Memory

Assigning Object Code to an Address Range

Description: This procedure assigns an object code section to a specified address range.

Procedure:

```
> LINK
*LOG
*MAP
*LOAD load
*LIST listing
*LINK object
*LOCATE sectionname,RANGE(startaddr,endaddr)
*END
```

Parameters: **load**—The filespec of the resulting load file.

listing—The filespec of the listing file or device. This listing includes a global symbol list and optional internal symbol or map listings. Refer to the Linker section of this manual for more information on the listing options.

object—The filespec of the object file from which the load file is generated. Repeat this line for each additional object file to be linked.

sectionname—The name of the object section that is relocated into the specified address range.

startaddr—The lower bound of the allowed relocation range. This address is specified in hexadecimal.

endaddr—The upper bound of the allowed relocation range. This address is specified in hexadecimal.

Comments: The object files to be LINKed are scanned for the section name specified by the LOCATE command. This object section is relocated within a RANGE bounded by the specified starting and ending addresses. All other object sections may be relocated anywhere within the processor's address range (this includes the specified RANGE). The LOAD command specifies the name of the load file. The listing may be sent to either a file or device.

Example:

```
> LINK
*LOG
*MAP
*LOAD LOAD
*LIST LNKL
*LINK IN.OBJ
*LOCATE SECTNAM,RANGE(0,FF)
*END
```

The input file IN.OBJ contains an object section named SECTNAM. The LOCATE command assigns the object section SECTNAM within a range of addresses 0 to FF. This range is specified by the RANGE parameter. Any object sections remaining in IN.OBJ are relocated within the address range of the processor. The resulting linked object code is sent to the load file LOAD. The linker commands are recorded in the file LNKL, along with the values of symbols and the locations of sections.

See also:

- Reserving an Area of Memory
- Invoking the Linker (Interactive Invocation)

Reserving an Area of Memory

Description: This procedure prevents relocation within a predefined address range.

Procedure:

```
Label   Operation      Operand

        SECTION        dummyname,ABSOLUTE
        ORG            startaddr
        BLOCK          areasize
        END
```

(Assemble an object file from the dummy source file:)

```
> ASM skipobject,,skipsource
```

(Link **skipobject** and your object file:)

```
> LINK load listing skipobject yourobject
```

Parameters:

dummyname—The name of the dummy section. The SECTION statement includes an ABSOLUTE option. This option directs the linker to use the address specified by the ORG directive as the starting address in memory. The operand of the BLOCK directive specifies the size of the reserved block of memory.

startaddr—The starting address of the reserved area.

areasize—The size of the reserved area.

skipsource—The filespec of the dummy source file.

skipobject—The filespec of the assembled dummy file.

yourobject—The filespec of the object file containing your program.

listing—The filespec of the listing file or device. This listing includes a global symbol list and optional internal symbol or map listings. Refer to the Linker section of this manual for more information on the listing options.

load—The filespec of the resulting load file.

Comments: A dummy file is linked with the object file containing your program. The ABSOLUTE option within the dummy section directs the linker to start the section at the address specified by the ORG directive. The BLOCK directive specifies the size of the reserved area. The object sections of your program are then relocated around the reserved area occupied by the dummy section. This reserved area may be used for ROM storage or memory-mapped I/O.

Example:

```
Label   Operation         Operand           Comment

        SECTION           SKIPSEC,ABSOLUTE;
        ORG               0                 ;Base Address
        BLOCK             100H              ;Reserved Memory
        END                                 ;
```

(Assemble the source file SKIP.ASM:)

```
> ASM SKIP.OBJ,,SKIP.ASM
```

(Link SKIP.OBJ and your object file)

```
> LINK LOAD LPT SKIP.OBJ YR.OBJ
```

The dummy file SKIP.ASM is assembled and stored in SKIP.OBJ. This object file is linked with the object file YR.OBJ. The ABSOLUTE option within the dummy section SKIPSEC directs the linker to start the section at address 0 (as specified by the ORG directive). The BLOCK directive then reserves a specified area of 100 (hexadecimal) bytes.

The linker relocates the remaining object sections around the SKIPSEC dummy section, leaving the first 256 bytes of memory unused by this program. The resulting load file is named LOAD. The listing device is the line printer (LPT).

See also:

- Invoking the Linker (Interactive Invocation)

BUILDING AND MAINTAINING A LIBRARY

Invoking LibGen

Description: This procedure shows the general format for invoking the library generator (LibGen).

Procedure:

```
> LIBGEN newlib listing oldlib
*  (Enter one LibGen command per line)
           .
           .
           .
*END
```

Parameters: **newlib**—The filespec of the new version of the library.

listing—The filespec of the listing file or device.

oldlib—The filespec of the old library file.

Comments: When you invoke LibGen you may use one or more commands to INSERT, EXTRACT, REPLACE, or DELETE library modules. If these commands are not entered, **oldlib** will be copied into **newlib** without modification. The END command must always terminate the invocation. For more information, see the Library Generator section of this manual.

Example:

```
> LIBGEN BIG.LIB LBGL SML.LIB
*INSERT BIG.OBJ
*END
```

The library file SML.LIB is copied into a new library file named BIG.LIB. The object module in file BIG.OBJ is inserted at the beginning of the BIG.LIB library file. The LBGL file lists the modules within the output library, the global symbols within each each module, and the actions performed by the library generator.

```
> LIBGEN,,LBGL AD.LIB
*END
```

The module names in the library file AD.LIB are sent to the listing file LBGL. This file may be copied to the system terminal (CONO) or a line printer (LPT).

See also:

- Creating a User-Defined Library
- Adding a New Library Module
- Examining a Library Module
- Replacing a Library Module
- Combining Libraries

Creating a User-Defined Library

Description: This procedure creates a new library from object modules.

Procedure:

```
> LIBGEN newlib listing
*INSERT object  (repeat as necessary)
          .
          .
          .
*END
```

Parameters: **newlib**—The filespec of the new library file.

listing—The filespec of the listing file or device.

object—The filespec of one of the files containing the object modules to be included in the library. Enter one INSERT command for each object file to be inserted.

Comments: Each object module in the library should be uniquely named (with the NAME directive) at assembly time. If you do not name the object modules, you will not be able to modify or maintain the library.

For more information about naming modules, refer to the NAME directive in the Assembler Directives section of this manual.

Example:

```
> LIBGEN MY.LIB LBGL
*INSERT A.OBJ
*INSERT B.OBJ
*INSERT C.OBJ
*INSERT Z.OBJ
*END
```

The library file MY.LIB is created in the current directory. The library file contains object modules from files A.OBJ, B.OBJ, C.OBJ, and Z.OBJ. The LBGL file lists the modules within the new library, the global symbols within each each module, and the actions performed by the library generator.

See also:

- Adding a New Library Module
- Combining Libraries

Adding a New Library Module

Description: This procedure copies a library and inserts an object module from a file into the new library.

Procedure:

```
> LIBGEN newlib listing oldlib
*INSERT addobj
*END
```

Parameters:

newlib—The filespec of the new library version.

listing—The filespec of the listing file or device.

oldlib—The filespec of the old library file.

addobj—The filespec of the file containing the object module that is added to the library.

Comments: Each object module in the library should have a unique name given at assembly time. If you do not name the object modules, you will not be able to modify or maintain the library.

Example:

```
> LIBGEN AD.LIB LPT MY.LIB
*INSERT D.OBJ
*END
```

The MY.LIB library is copied into AD.LIB. The object module within D.OBJ is inserted at the beginning of the library. The list of the modules within the output library, the global symbols within each each module, and the actions performed by the library generator are listed on the line printer.

See also:

- Examining a New Library Module
- Replacing a Library Module

Extracting a Library Module

Description: This procedure copies a library module into an object file.

Procedure:

```
> LIBGEN,,listing lib
*EXTRACT modlib TO newfile
*END
```

Parameters:

listing—The filespec of the listing file or device.

lib—The filespec of the library file.

modlib—The name of the library module that will be copied into a new file.

newfile—A filespec of the new file used to store the extracted library module.

Comments: The library **lib** and the **newfile** both contain the object module **modlib** after this procedure is complete.

Example:

```
> LIBGEN,,LPT AD.LIB
*EXTRACT AMOD TO X.OBJ
*EXTRACT BMOD TO Y.OBJ
*END
```

Two modules are copied from the library file AD.LIB into two object files. AMOD is copied into the object file X.OBJ; BMOD is copied into the object file Y.OBJ. The line printer (LPT) lists the modules within the library, the global symbols within each each module, and the actions performed by the library generator.

See also:

- Adding a New Library Module
- Replacing a Library Module

Replacing a Library Module

Description: This procedure replaces an existing library module with a new one.

Procedure:

```
> LIBGEN newlib listing oldlib
*REPLACE modlib BY newfile
*END
```

Parameters:

newlib—The filespec of the new version of the library.

listing—The filespec of the listing file or device.

oldlib—The filespec of the old library file.

modlib—The name of the library module that will be replaced by the new module.

newfile—The filespec of the new module.

Comments:

The old library is scanned for **modlib**, which is then deleted and replaced by the object module within **newfile**.

Each object module in the library should have a unique name given at assembly time. If you do not name the object modules, you will not be able to modify or maintain the library. For more information about naming modules, refer to the NAME directive in the Assembler Directives section of this manual.

Example:

```
> LIBGEN NU.LIB LBGL AD.LIB
*REPLACE AMOD BY X.OBJ
*REPLACE BMOD BY Y.OBJ
*END
```

The AD.LIB library is copied into NU.LIB. The object module within X.OBJ replaces AMOD, and the object module within Y.OBJ replaces BMOD. The listing file LBGL lists the contents of the new library and the actions performed by the library generator.

See also:

- Examining a Library Module
- Adding a New Library Module

Combining Libraries

Description:

This procedure adds the contents of a small library to a larger library.

Procedure:

```
> LIBGEN,,oldlisting smallib
*EXTRACT mod1 TO file1
*EXTRACT mod2 TO file2
          .
          .
          .
*EXTRACT modx TO filex
*END
```

(All of the modules in the smaller library have been copied into individual files.)

```
> LIBGEN newlib newlisting biglib
*INSERT file1
*INSERT file2
          .
          .
          .
*INSERT filex
*END
```

Parameters:

oldlisting—The filespec of the listing file or device that shows the contents of the old library.

smallib—The filespec of the smaller library file.

mod1, mod2, ..., modx—The library modules extracted out of the smaller library.

file1, file2, ...,filex—The filespecs designating the individual files used to store the modules extracted from **smallib**.

newlib—The filespec of the new library created from the combination of **smallib** and **biglib**.

newlisting—The filespec of the listing file or device that shows the contents of the new library.

biglib—The filespec of the larger library file.

Comments:

This procedure copies all of the modules **mod1, mod2, ..., modx** from the **smallib** library into individual object files **file1, file2, ..., filex**. The **biglib** library is copied into the new library file **newlib**. The files **file1, file2, ..., filex** are then inserted into the beginning of the **newlib** library. The first listing shows the contents of the **smallib** library and the second listing shows the contents of the **newlib** combined library.

Example:

```
> LIBGEN,,OLD.LBGL MY.LIB
*EXTRACT AMOD TO A.OBJ
*EXTRACT BMOD TO B.OBJ
*EXTRACT CMOD TO C.OBJ
*END
```

```
> LIBGEN NU.LIB NEW.LBGL YR.LIB
*INSERT A.OBJ
*INSERT B.OBJ
*INSERT C.OBJ
*END
```

Modules AMOD, BMOD, and CMOD are copied from the MY.LIB library into intermediate object files A.OBJ, B.OBJ, and C.OBJ. The YR.LIB library is copied into a new library named NU.LIB. The intermediate files are then inserted at the beginning of the NU.LIB library. The listing file OLD.LBGL lists the contents of the old library and the actions of the first LIBGEN command; the listing file NEW.LBGL lists the contents of the new library and the actions of the second LIBGEN command.

See also:

- Examining a Library Module
- Adding a New Library Module
- Invoking LIBGEN

Section 3
ASSEMBLER INTRODUCTION

Page

ILLUSTRATIONS

Fig.
No.

Section 3

ASSEMBLER INTRODUCTION

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:

- **Syntax Notation.** Describes the syntax conventions used throughout this manual.
- **Assembler Invocation.** Describes how to invoke the assembler with the operating system ASM command.
- **Assembler Input.** Describes how the source module is used as input to the assembler.
- **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.

SYNTAX NOTATION

Introduction

This manual uses syntax blocks to present:

- operating system commands,
- linker commands,
- LibGen commands,
- assembler directives, and
- assembler functions.

The conventions used in the syntax blocks are described in this subsection. Figure 3-1 illustrates a sample syntax block.

```
                                SYNTAX

                                    [,PA] {param2}
  COMMAND param1[/par-one]          [,PB] {param3}...
```

3454-6

Fig. 3-1. Sample syntax block.

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

In this fictitious example, 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.

Command Name

A command name is a word that represents a command or assembler directive. Uppercase characters in the command name must be entered exactly as shown. When part of the command name is underlined, you may enter that shortened form. In Fig. 3-1, the short form of the command is COM.

Parameters

Parameters specify or modify how the command is executed. Parameters may be names, addresses, devices, numbers, characters, or symbols. Capitalized 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.

Lowercase 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.

Parameters may be required or optional in the command line. 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. 3-1 "/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. In the example, either "param2" or "param3" must be selected. If the parameters are stacked within brackets [], the selection is optional. In the example, you may select either "PA" or "PB" or neither. 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. The choice of "param2" or "param3" may be repeated as many times as the line permits.

ASSEMBLER INVOCATION

The assembler is invoked by the operating system command ASM.

SYNTAX

ASM [object] [listing] source...

PARAMETERS

object — The filespec where the object module is written. If this parameter is omitted, no object module is created.

listing — The filespec where the assembler listing is to be written. If this parameter is omitted, no listing is created. The listing can be printed directly to the line printer, by specifying LPT as the listing device.

source — The filespec of the source code.

EXPLANATION

The ASM command invokes the Tektronix Assembler. The source code residing on one or more files is translated into object code (machine language), which is stored on the specified object file or device. An assembler listing is generated and written on the specified file or device. If either the object or listing is omitted, you must enter two commas. If both are omitted, you must enter three commas. (See the Examples.)

The assembler makes two passes through the source code. (See the Assembler Execution subsection in this section.) If you are entering the source code from a device, you must enter the source code twice, once for each assembler pass.

EXAMPLES

```
ASM OBJ ASML ASM
```

This example assembles the source file ASM, creating the object file OBJ. The assembler listing is stored in the file ASML. All files reside in the current directory.

```
ASM,,LPT MY.ASM
```

This example assembles the source file MY.ASM but does not generate an object file. The assembler listing is output to the line printer.

```
ASM,,,MY.ASM
```

This example assembles the source file MY.ASM that resides in the current directory, but does not generate an object file or an assembler listing. This form of invocation might be used when errors are suspected in the source file. The errors are listed on the system terminal.

ASSEMBLER INPUT

Assembler input consists of assembly language statements, as defined in the Language Elements section of this manual. There are three types of assembler 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 ASM command line. If the input is read from a 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.

ASSEMBLER EXECUTION

Two Passes

The assembler makes two passes over 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. (See the LIST directive in the Assembler Directives section of this manual.)

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 performs the following steps:

1. 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. (See the Language Elements section of this manual.)
2. Performs the indicated action according to the type of statement:
   a. **assembly language instruction**—The assembler translates each assembly language instruction into the corresponding machine instruction.
   b. **assembler directives**—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.)

      For example, some directives may simply define assembler symbols, while some may alter the processing order of the statements. An IF directive causes a block of code to be assembled or skipped depending on the true/false value of the IF condition. When a MACRO directive is encountered, the assembler simply stores the macro definition.

c. **macro invocation**—The assembler processes each statement within the previously defined macro. (See the Macros section of this manual.)

The REPEAT directive within a macro causes a block of statements to be assembled more than once. (See the REPEAT directive in the Assembler Directives section of this manual.)

ASSEMBLER OUTPUT

The assembler generates an object module and an assembler listing. Any assembler errors are displayed to 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 two parts: the source listing, and the symbol table. Figures 3-2, 3-3, and 3-4 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 three pages: pages 1 (Fig. 3-2) and 2 (Fig. 3-3) show the source listing, which includes the source program and the object code generated for each statement; page 3 (Fig. 3-4) shows the symbol table. Refer to Figs. 3-2, 3-3, and 3-4 as you read the following descriptions.

Source Listing

Each line of the source listing contains the following information:

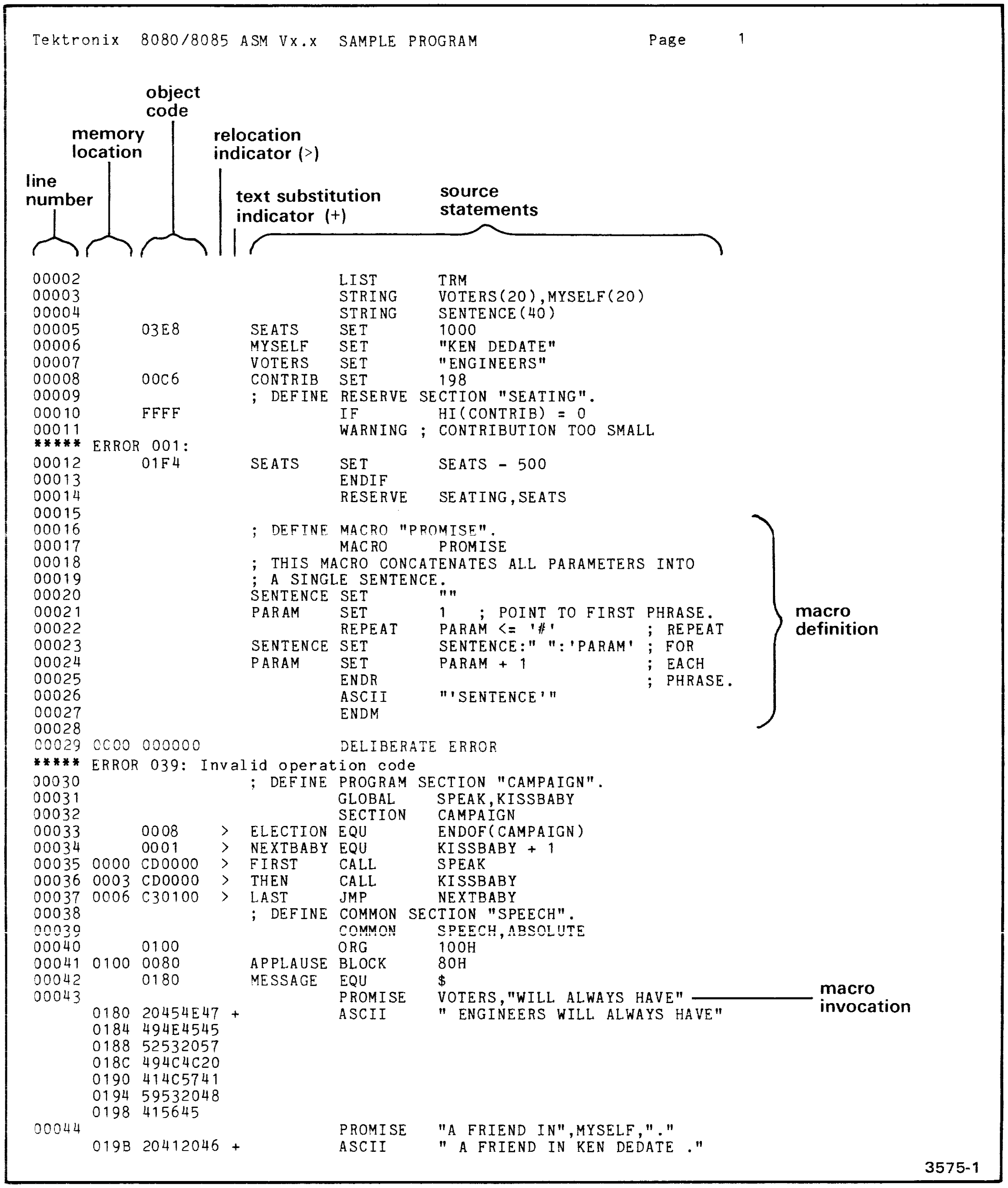
1. the **line number** (decimal).
2. the **memory location** (hexadecimal) of the object code generated (if any).
3. the assembled **object code** (hexadecimal).
4. a **relocation indicator** (>) if the object code is to be adjusted by the linker.
5. a **text substitution indicator** (+) if the assembler has modified the source statement.
6. the **source statement**.

If any statement contains an error, the appropriate error message appears directly after the statement.

Symbol Table

The assembler symbol table displays the value and type of each symbol. The symbol table is divided into the following groups:

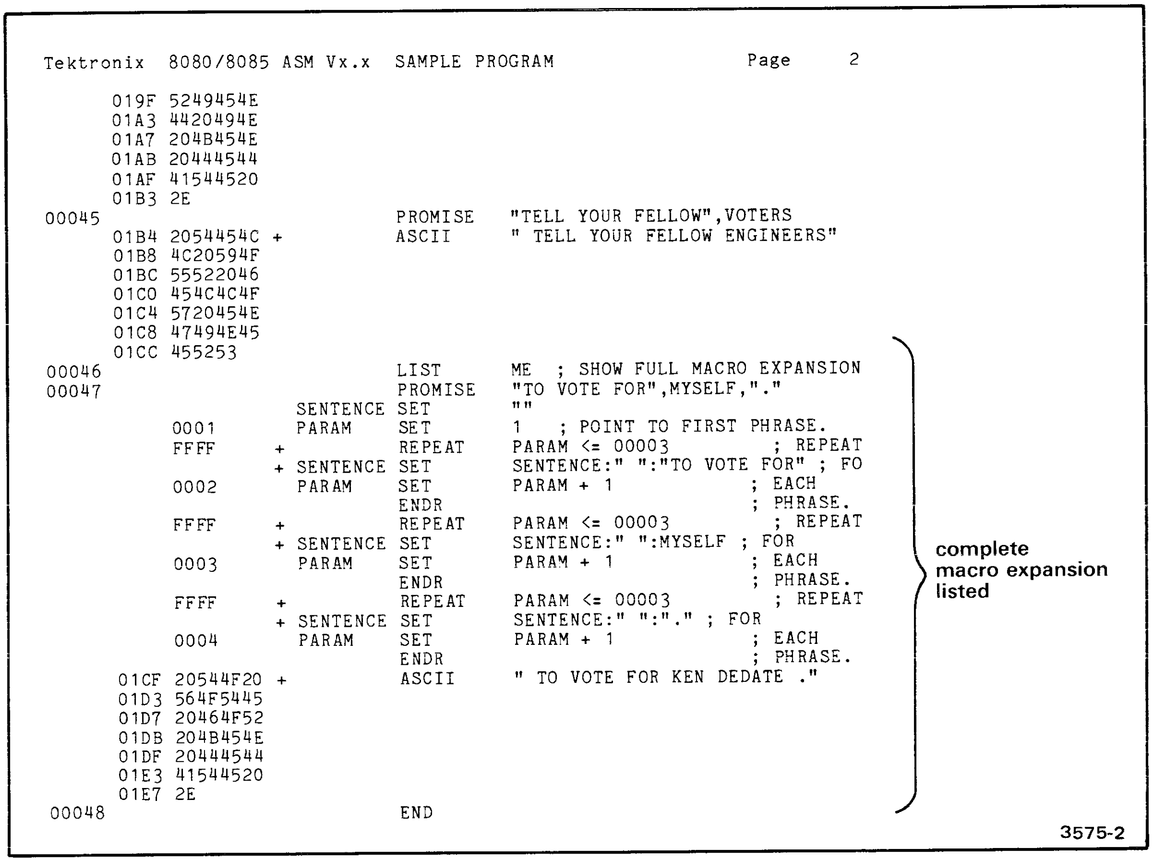
1. **Strings and Macros**—Symbols that are declared as string variables or defined as macro names are listed in this group. The letter "S" after the symbol indicates a string variable and "M" indicates a macro. A number (in hexadecimal) follows each symbol. That number represents the number of bytes required by the assembler to store the character string or macro definition.
2. **Scalars**—Scalar symbols are listed in this group. The letter "G" following the symbol indicates a global symbol. The letter "V" indicates a variable defined with the SET directive. The number 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. "*****" indicates an undefined symbol.
3. **Sections**—Each section of the program is listed alphabetically in 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—PAGE, INPAGE, ABSOLUTE, or, if not specified, byte-relocatable.
   - Length of section—the number of bytes of object code generated (in hexadecimal).
   - All address symbols within the section—each with its address relative to the beginning of the section. "E" indicates that the ENDOF function is used to determine the address. "H" indicates that the HI function is used and "L" indicates that the LO function is used.
4. **Unbound Globals**—Symbols used in this module but defined elsewhere are listed in this group. Any symbols based on an unbound global are listed below that global.
5. **Statistics**—Two summary lines of statistics appear at the end of the symbol table. The first line shows the number of source lines, the number of assembled lines, and the number of available bytes. The number of available bytes indicates the amount of space remaining in the assembler for storage of string variables, macros, and labels. The second line indicates the number of errors and undefined symbols, if any. These lines of statistics also appear on the system terminal at the end of the assembly process.



```
Tektronix  8080/8085 ASM Vx.x  SAMPLE PROGRAM              Page     1

00002                          LIST      TRM
00003                          STRING    VOTERS(20),MYSELF(20)
00004                          STRING    SENTENCE(40)
00005      03E8       SEATS    SET       1000
00006                 MYSELF   SET       "KEN DEDATE"
00007                 VOTERS   SET       "ENGINEERS"
00008      00C6       CONTRIB  SET       198
00009                 ; DEFINE RESERVE SECTION "SEATING".
00010      FFFF                IF        HI(CONTRIB) = 0
00011                          WARNING ; CONTRIBUTION TOO SMALL
***** ERROR 001:
00012      01F4       SEATS    SET       SEATS - 500
00013                          ENDIF
00014                          RESERVE   SEATING,SEATS
00015
00016                 ; DEFINE MACRO "PROMISE".
00017                          MACRO     PROMISE
00018                 ; THIS MACRO CONCATENATES ALL PARAMETERS INTO
00019                 ; A SINGLE SENTENCE.
00020                 SENTENCE SET       ""
00021                 PARAM    SET       1   ; POINT TO FIRST PHRASE.
00022                          REPEAT    PARAM <= '#'         ; REPEAT
00023                 SENTENCE SET       SENTENCE:" ":'PARAM' ; FOR
00024                 PARAM    SET       PARAM + 1            ; EACH
00025                          ENDR                           ; PHRASE.
00026                          ASCII     "'SENTENCE'"
00027                          ENDM
00028
00029 0000 000000              DELIBERATE ERROR
***** ERROR 039: Invalid operation code
00030                 ; DEFINE PROGRAM SECTION "CAMPAIGN".
00031                          GLOBAL    SPEAK,KISSBABY
00032                          SECTION   CAMPAIGN
00033      0008    >  ELECTION EQU       ENDOF(CAMPAIGN)
00034      0001    >  NEXTBABY EQU       KISSBABY + 1
00035 0000 CD0000  >  FIRST    CALL      SPEAK
00036 0003 CD0000  >  THEN     CALL      KISSBABY
00037 0006 C30100  >  LAST     JMP       NEXTBABY
00038                 ; DEFINE COMMON SECTION "SPEECH".
00039                          COMMON    SPEECH,ABSOLUTE
00040      0100                ORG       100H
00041 0100 0080       APPLAUSE BLOCK     80H
00042      0180       MESSAGE  EQU       $
00043                          PROMISE   VOTERS,"WILL ALWAYS HAVE"
      0180 20454E47 +          ASCII     " ENGINEERS WILL ALWAYS HAVE"
      0184 494E4545
      0188 52532057
      018C 494C4C20
      0190 414C5741
      0194 59532048
      0198 415645
00044                          PROMISE   "A FRIEND IN",MYSELF,"."
      019B 20412046 +          ASCII     " A FRIEND IN KEN DEDATE ."
```

Fig. 3-2. Sample assembler listing (Part 1 of 3).

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



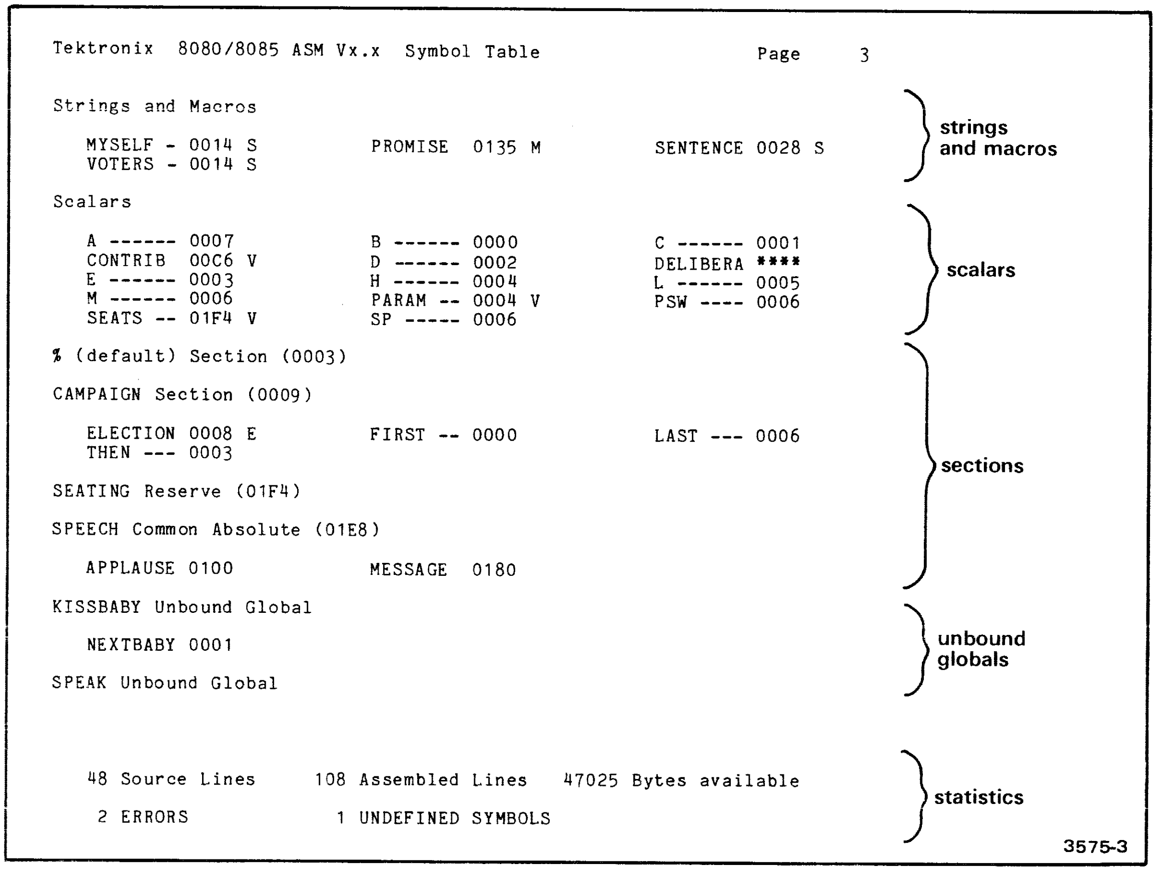
```
Tektronix  8080/8085 ASM Vx.x  SAMPLE PROGRAM                Page     2

      019F 5249454E
      01A3 4420494E
      01A7 204B454E
      01AB 20444544
      01AF 41544520
      01B3 2E
00045                          PROMISE   "TELL YOUR FELLOW",VOTERS
      01B4 2054454C +          ASCII     " TELL YOUR FELLOW ENGINEERS"
      01B8 4C20594F
      01BC 55522046
      01C0 454C4C4F
      01C4 5720454E
      01C8 47494E45
      01CC 455253
00046                          LIST      ME  ; SHOW FULL MACRO EXPANSION
00047                          PROMISE   "TO VOTE FOR",MYSELF,"."
                      SENTENCE SET       ""
           0001       PARAM    SET       1    ; POINT TO FIRST PHRASE.
           FFFF     +          REPEAT    PARAM <= 00003         ; REPEAT
                    + SENTENCE SET       SENTENCE:" ":"TO VOTE FOR" ; FO
           0002       PARAM    SET       PARAM + 1            ; EACH
                               ENDR                           ; PHRASE.
           FFFF     +          REPEAT    PARAM <= 00003         ; REPEAT
                    + SENTENCE SET       SENTENCE:" ":MYSELF ; FOR
           0003       PARAM    SET       PARAM + 1            ; EACH
                               ENDR                           ; PHRASE.
           FFFF     +          REPEAT    PARAM <= 00003         ; REPEAT
                    + SENTENCE SET       SENTENCE:" ":"." ; FOR
           0004       PARAM    SET       PARAM + 1            ; EACH
                               ENDR                           ; PHRASE.
      01CF 20544F20 +          ASCII     " TO VOTE FOR KEN DEDATE ."
      01D3 564F5445
      01D7 20464F52
      01DB 204B454E
      01DF 20444544
      01E3 41544520
      01E7 2E
00048                          END
```

complete macro expansion listed

3575-2

Fig. 3-3. Sample assembler listing (Part 2 of 3).

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



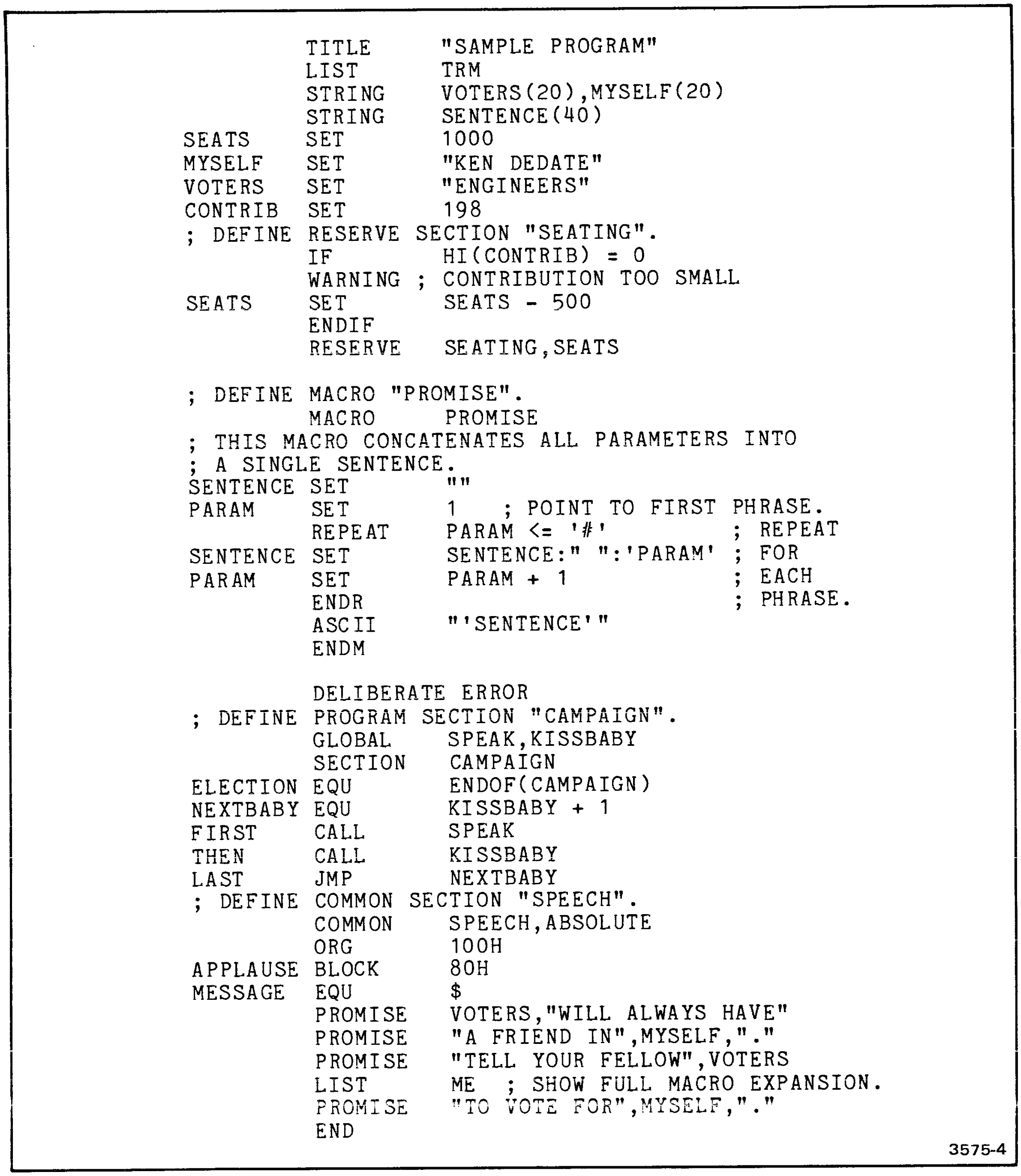
```
Tektronix  8080/8085 ASM Vx.x  Symbol Table                       Page     3

Strings and Macros

   MYSELF - 0014 S          PROMISE  0135 M          SENTENCE 0028 S
   VOTERS - 0014 S

Scalars

   A ------ 0007            B ------ 0000            C ------ 0001
   CONTRIB  00C6 V          D ------ 0002            DELIBERA ****
   E ------ 0003            H ------ 0004            L ------ 0005
   M ------ 0006            PARAM -- 0004 V          PSW ---- 0006
   SEATS -- 01F4 V          SP ----- 0006

% (default) Section (0003)

CAMPAIGN Section (0009)

   ELECTION 0008 E          FIRST -- 0000            LAST --- 0006
   THEN --- 0003

SEATING Reserve (01F4)

SPEECH Common Absolute (01E8)

   APPLAUSE 0100            MESSAGE  0180

KISSBABY Unbound Global

   NEXTBABY 0001

SPEAK Unbound Global


   48 Source Lines     108 Assembled Lines    47025 Bytes available

    2 ERRORS             1 UNDEFINED SYMBOLS
```

Fig. 3-4. Sample assembler listing (Part 3 of 3).

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

Sample Source Program

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



```
         TITLE     "SAMPLE PROGRAM"
         LIST      TRM
         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
         ENDIF
         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    CALL      SPEAK
THEN     CALL      KISSBABY
LAST     JMP       NEXTBABY
; 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
```

3575-4

Fig. 3-5. Sample 8080A source program.

This source program generated the sample assembler listing that was shown in Figs. 3-2, 3-3, and 3-4. 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. 3-5) with the assembler listing (Figs. 3-2, 3-3, and 3-4). 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 8080/8085 ASM Vx.x  SAMPLE PROGRAM        Page 1
                              \_____ _____/
                                    v
                                  title
```

The next statement in the source program is:

```
        LIST       TRM
```

The LIST directive controls various features of the assembler listing. This particular use, with the TRM option, prints the assembler listing in a 72-character width instead of the default 132-character width. Although this line appears in the assembler listing, it does not generate object code.

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

The next 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 code. The variables appear in the symbol table of the assembler listing (Fig. 3-4). The "S" following each symbol indicates 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 (line 00005 in the assembler listing) as 03E8 hexadecimal. No memory location appears on the line because the value is not stored in the object program. MYSELF and VOTERS require string values enclosed in double quotes (" ") since they are string variables. The numeric value 198 (00C6H) 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
          ENDIF
```

These four statements are a conditional assembly block. The IF directive causes the block of statements between the IF and ENDIF 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:

$$\underbrace{\texttt{00}}_{\text{high byte}}\texttt{C6H (198 decimal)}$$

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

The SET directive changes the value of the symbol SEATS from 03E8H (1000 decimal) to 01F4H (1000–500 decimal). See line 00012 of the assembler listing.

```
          RESERVE   SEATING,SEATS
```

This statement is an assembler directive that reserves a section in memory. The section is named "SEATING" and has 01F4H bytes (the current value of SEATS). The section SEATING appears in the symbol table (Fig. 3-4), 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 source listing.

```
; DEFINE MACRO "PROMISE".
```

Although this comment line appears in the assembler listing, 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 statements in a macro definition (with the exception of full comment lines) are 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 because DELIBERATE, which appears in the operation field, is not an assembly language instruction, an assembler directive, or a macro invocation. An error message is printed on the terminal and listed in the assembler listing.

```
; 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, although they are 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 a section. The assembler listing for this source line is:

```
00033       0008     >  ELECTION EQU          ENDOF(CAMPAIGN)
                     |
               relocation indicator
```

The relocation indicator (>) 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 undetermined until link time. The 0008, which is the value assigned to ELECTION, tells us that there are nine bytes (0000 through 0008) in the section. (The beginning address of every relocatable section is 0000 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 address value of the unbound global KISSBABY. In the assembler listing (Fig. 3-2, line 00034), the relocation indicator again shows that the object code will be adjusted by the linker. The 0001 indicates that the adjusted address will be +1 relative to the address of KISSBABY.

```
FIRST     CALL      SPEAK
```

This statement is an 8080A assembly language instruction which calls the subroutine SPEAK. The assembler listing shows the object code that is generated:

```
00035 0000 CD0000  >  FIRST    CALL      SPEAK
       |    |  |
       |    |  └── address of subroutine SPEAK
       |    └───── OP code of the instruction CALL
       └────────── memory location
```

Since this is the first statement in section CAMPAIGN that produces object code, the memory location assigned is 0000. CD is the OP code for the instruction CALL. 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.) The ">" indicates that the object code (the address of the subroutine SPEAK) will be adjusted by the linker.

```
THEN      CALL      KISSBABY
```

This statement calls the subroutine KISSBABY, another unbound global. In the listing of this statement (line 00036), the memory location is 0003, since the previous instruction (CALL SPEAK) occupies bytes 0000-0002.

```
LAST      JMP       NEXTBABY
```

This statement is an 8080A assembly language instruction. The object code generated is "C30100". (See line 00037 in the assembler listing.) C3 is the OP code for the instruction JMP. NEXTBABY has the value 0001 (the 8080A stores two-byte numbers in low-byte/high-byte order). 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 location of the first byte of the section is defined to be 100H by the ORG statement.

```
APPLAUSE  BLOCK     80H
```

This statement generates the first byte of the common section SPEECH. The memory location of the first byte is 0100H.

```
00041 0100 0080        APPLAUSE BLOCK      80H
      └──┘
  memory location
```

This BLOCK directive reserves a block of 80H bytes. The symbol APPLAUSE represents the address of the first byte of the block (0100).

```
MESSAGE   EQU       $
```

This statement is an assembler directive that 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.

```
00042       0180          MESSAGE  EQU        $
```

The location counter was advanced to 0180H 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:

```
00043                              PROMISE   VOTERS,"WILL ALWAYS HAVE"
        0180 20454E47 +            ASCII     " ENGINEERS WILL ALWAYS HAVE"
        0184 494E4545
        0188 52532057
        018C 494C4C20
        0190 414C5741
        0194 59532048
        0198 415645
```

text substitution indicator

ASCII representation of " ENGINEERS WILL ALWAYS HAVE"

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"
```

Let's look at the other 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 single 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 (+).

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: When a macro is invoked, the assembler listing shows all of the assembled statements of the macro expansion. (Comment lines within a macro are not listed because they are not saved with the macro definition.)

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

This macro invocation returns the ASCII representation of " TO VOTE FOR KEN DEDATE ." Notice in the assembler listing (following line 00047) that the text substitution indicator appears on seven lines. The '#' is replaced by "00003" 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:

```
        + SENTENCE SET        SENTENCE:" ":"TO VOTE FOR" ; FO
```

In the source program, the comment was "; FOR". The "R" does not appear on the source listing because the LIST TRM directive had previously trimmed the listing to 72 characters.

The last statement of the source code is:

```
            END
```

This statement marks the end of the source program.

Sample Symbol Table

Now let's examine the symbol table for the sample program (Fig. 3-4). 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. The number of bytes required to store the macro definition is 0135H.

Listed under **Scalars** are not only the numeric symbols used in the program (CONTRIB, PARAM, and SEATS), but also 8080A register names, since they are also symbols with scalar values. Each variable is listed with the last value assigned to it.

"DELIBERA" is listed in this section of the table. The four stars (****) flag it as an undefined symbol. When the assembler examined the statement "DELIBERATE ERROR", the word was treated as an undefined symbol since it was not an assembly language instruction, an assembler directive, or a defined macro. (Only the first eight characters of a symbol are recognized.) This error also explains the next line of the symbol table:

```
 % (default) Section (0003)
```

When there are statements in an undefined section, the assembler assigns them to the default section. (See the SECTION directive, in the Assembler Directives section of this manual, for a description of the default section.) In our sample program, the assembler generated three bytes of zeros in response to the "DELIBERATE ERROR" line and created a default section.

There are four **Sections** in our program: the default section, CAMPAIGN, SEATING, and SPEECH.

```
CAMPAIGN Section (0009)

    ELECTION 0008 E         FIRST -- 0000          LAST --- 0006
    THEN --- 0003
```

In this section summary, the name of the section is CAMPAIGN, which is of type "Section". The section is 0009 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 byte-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.**

Let's look at the lines of statistics:

```
48 Source Lines     108 Assembled Lines    47025 Bytes available

 2 ERRORS             1 UNDEFINED SYMBOLS
```

There are more **Assembled Lines** (108) then **Source Lines** (48) because the macro invocations and REPEAT block cause some of the source lines to be assembled more than once.

The statistics also include the number of **Bytes Available** in the assembler for further storage of labels, string variables, and macros.

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

Section 4
LANGUAGE ELEMENTS

TABLES

ILLUSTRATIONS

Section 4

LANGUAGE ELEMENTS

INTRODUCTION

This section provides reference information about the Tektronix Assembler language elements. The section discusses 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. Defines and gives the results of assembler **functions.** The functions are listed alphabetically for reference.

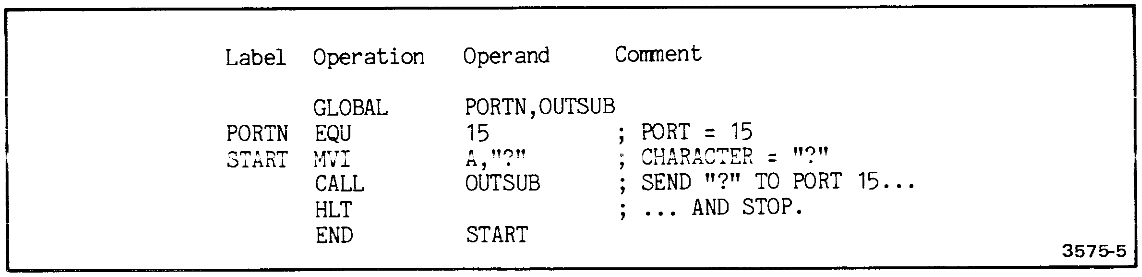
STATEMENT FIELDS

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 (ASCII code 13). 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
```

Readability is improved when each field has a constant width on each line. This columnar format can be implemented with tab settings. Fig. 4-1 is an example of a formatted 8080A source file.



```
Label  Operation   Operand        Comment

       GLOBAL      PORTN,OUTSUB
PORTN  EQU         15             ; PORT = 15
START  MVI         A,"?"          ; CHARACTER = "?"
       CALL        OUTSUB         ; SEND "?" TO PORT 15...
       HLT                        ; ... AND STOP.
       END         START
```

Fig. 4-1. Formatted Source File.

Each field has a constant width in this 8080A 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 or tab 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 permissible in all statements, including assembler directives, assembly language instructions, and macro invocations.

The meaning of the label in an **assembler directive** statement depends upon the particular directive. For most 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 constant symbol 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  MVI         A,"?"      ; CHARACTER = "?"
```

In this line, the label START represents the address of the first byte of the MVI 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 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 character, or a semicolon (indicating the beginning of a comment field).

The word in the operation field indicates the type of action to be taken by the assembler. The word 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  MVI         A,"?"      ; CHARACTER = "?"
```

MVI (an 8080A 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 an assembler directive. In this case, the name of that assembler directive is changed. A list of any changed assembler directive names are found in the appropriate Assembler Specifics section for your microprocessor.

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  MVI         A,"?"      ; CHARACTER = "?"
```

The 8080A MVI instruction requires two operands: a register, followed by a value. In this example, register A (a predefined symbol) and a string value are used.

The value in the operand field may be represented by an expression. (See the Expression discussion in this section.) An expression may consist 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 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 assembly language instruction, or of an ASCII, BLOCK, BYTE, EQU, SET, or WORD directive; or
- the symbol must appear in the operand field of a GLOBAL, STRING, SECTION, COMMON, or RESERVE 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      ; OUTSUB STARTS HERE
```

In this example, the first statement has no effect on the object code produced, because 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 single quote (') signals substitution. Therefore, to include a single quote (') character within a comment, you must precede the ' character with an up-arrow (∧) character.

NOTE

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

SYMBOLS

A symbol is a user-defined or predefined word that represents a value or an 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 modify 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 eight characters are considered significant; any additional characters are discarded. Some examples of valid user-defined symbols are:

```
PORTN
OUTSUB
LOOP
LOOP.5
A123456$
TO_DO
AVERYLONGSYMBOL  (same as AVERYLON)
```

Defining Symbols

User-defined symbols are defined when they appear in: (1) the label field of assembly lanugage instructions, macro invocations, and assembler directives, or (2) the operand field of GLOBAL, SECTION, COMMON, RESERVE, MACRO, or STRING directives. 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 8080A) can be represented with a symbol.

```
LOOP  INC        A
       .
       .
       .
      JMP        LOOP
```

LOOP is a user-defined symbol representing the address of the instruction INC (Increment).

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 in 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,
- assembly language instruction mnemonics, and
- processor register names and symbols.

The assembler directives and options are listed in the Assembler Directives section of this manual. See the Assembler Specifics section of this manual for the list of instruction mnemonics and reserved words for your processor.

VALUES

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

Numeric Values

The assembler uses two types of numeric values: scalars and addresses. All numeric values are treated as 16-bit (2-byte) numbers. Scalars are signed values. Addresses are unsigned values.

Scalar Values

Scalar values are signed integers ranging from ⁻32768 to 32767. (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 unsigned 16-bit address takes a value in the range 0 to 65535.

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. (See the Linker section of this manual for a discussion on section relocation.) The base address of each section is then redefined by the linker. The actual address of a byte is unknown 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 machine instruction or data item currently being generated). For example:

```
Label  Operation   Operand

        IF          $ > 0FFH
```

In this statement the current value of the location counter is compared with the value 0FFH.

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 **binary** numbers.

  `1010B and 11111111B are binary numbers`

- **O** (capital letter O, not zero) or **Q** denotes **octal** numbers.

  `377Q and 177777O are octal numbers`

- **H** denotes **hexadecimal** numbers.

  `1A2CH and 0FFFFH 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 made synonymous with the constant 15.

Numeric Variables

During assembly, a numeric value may be assigned to an assembler variable with the SET directive. An assembler variable allows temporary assignments to be made to a symbol. When the variable is encountered, the current value is used. Rules for creating an assembler variable follow the rules for creating a user-defined symbol. (See User-defined Symbols in this section.) A symbol used as an assembler variable must not have been previously defined.

```
COUNT  SET        1
```

In this example, the symbol COUNT is an assembler variable and may be assigned various numeric values with the SET directive. When the symbol COUNT is encountered by the assembler, the current value is used. If another SET directive reassigns another value to COUNT, the reassigned value is used.

String Values

Character strings are accepted by the assembler. Individual characters are translated into their ASCII representation (8 bits).

String Constants

String values entered as constants in an assembler program are enclosed in double quotes ("):

```
"STRINGS"
```

The null string ("") contains zero characters. 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 (") or a single quote ('), precede the special character with an up-arrow (∧).

NOTE

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

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 be declared with the STRING directive before being used. The STRING directive specifies the maximum length of the string variable. The maximum length (which defaults to 8) is enclosed in parentheses. For example:

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

In this example, the symbol STVAR is a string variable. The maximum length for any string assigned to the variable STVAR is ten.

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

Conversions

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

```
Label Operation Operand

SYM1  EQU       "AB"
```

The string is converted to a two-byte numeric value. The numeric value is the ASCII representation of the string. If the string is longer than two characters, the first two characters are converted and an error message is generated. If the string length is one character, the high-order byte of the resulting value is zero. The value of the null string ("") is zero. For example:

Character String	Numeric Value
""	0000H
"A"	0041H
"?"	003FH
"AB"	4142H
"ABC"	4142H (error message 085 is generated)
"12"	3132H

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 string representation is six characters long. The first character is a zero or minus (-) depending on the sign of the number. The remaining five 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	"000000"
-1	"-00001"
400	"000400"
200H	"000512"

For example:

```
        STRING      STRVAR(10)
STRVAR  SET         -1
```

STRVAR now has the value "-00001".

TEXT SUBSTITUTION

String values can be substituted within a statement line during assembly by the use of string variables. The single quote (') is the substitution delimiter. When the assembler encounters a string variable enclosed within single quotes ('variable'), the variable is replaced by the current string value assigned to that string variable.

When processing a statement, the assembler first performs all indicated text substitutions. For example:

```
Label  Operation   Operand    Comment

       STRING      OP
OP     SET         "WORD"
        .
        .
        .
       'OP'        1,2,3      ; DO 'OP' TO 1,2,3
```

When the assembler scans the line "'OP' 1,2,3", it first performs the following substitution:

```
       WORD        1,2,3      ; DO WORD TO 1,2,3
```

The statement line then contains the assembler directive WORD.

During assembly, the percent sign (%) represents the current section name. With the use of text substitution, the current section name can be inserted into the assembly language program. For example:

```
Label    Operation   Operand    Comment

         STRING      SECNAME(8)
SECNAME  SET         "'%'"      ; SAVE CURRENT SECTION NAME
         SECTION     QQ         ; SWITCH TO NEW SECTION
          .
          .
          .
         RESUME      'SECNAME' ; SWITCH BACK TO PREVIOUS SECTION
```

Parameter substitution performed during macro expansion is a form of text substitution. See the Macros section for information on parameters.

EXPRESSIONS

Introduction

An expression is a combination of constants, variables, or functions connected by operators that yields a numeric or string value. The assembler accepts expressions in the operand field. An operand expression is evaluated at assembly time, and the numeric or string value is used. Table 4-1 lists the operators and functions that can be used in expressions.

Table 4-1
Expression Operators and Functions

Type	Operator/ Function	Meaning
Unary Arithmetic	+	Identity
	–	Sign inversion
Binary Arithmetic	*	Multiplication
	/	Division
	+	Addition
	–	Subtraction
	MOD	Remainder
	SHL	Left shift
	SHR	Right shift
Unary Logical	\	NOT (bit inversion)
Binary Logical	&	AND
	!	Inclusive 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	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

Hierarchy

In an expression involving more than one operator, the operators are performed according to a predefined order of precedence. Table 4-2 shows the operator hierarchy.

Table 4-2
Hierarchy of Operators

Precedence	Operators
1.	BASE DEF ENDOF HI LO NCHR SCALAR SEG STRING (functions)
2.	: (concatenation)
3.	+ - (unary plus and minus) \ (logical NOT)
4.	* / MOD SHL SHR (multiplication, division, shifts)
5.	+ - (addition, subtraction)
6.	= <> > >= < <= (relational)
7.	& (logical AND)
8.	! !! (logical OR, exclusive OR)

Expression operators at the top of the table have the highest precedence and are performed first. Operators at the bottom have the lowest precedence and are performed last. Operators on the same line have equal precedence, and are performed from left to right within the expression.

Parentheses may be used to override the order of precedence. The most deeply nested subexpressions are evaluated first. It is possible to create an expression that is too complex for the assembler to evaluate. 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, as described in String Conversions (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 Remainder — The remainder that results from dividing the first term by the second. 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 Left shift

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	0	0	0	0	0	0	0	0	1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

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

OFOFOH SHL 1 results in OE1EOH

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0

1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0

If the second term is greater than 16, the result is zero, and an error message is generated.

SHR Right shift

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	0	0	0	0	0	0	0	1	0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

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 16, 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

\ 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.)

\0F0FH results in 0F0F0H

0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0

& AND

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

Example:

```
DVAL   EQU     0F0F0H & 0CCC0H
```

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0

DVAL is assigned 0C0C0H

1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0

! OR

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

Example:

```
RVAL   EQU     0F0F0H ! 0CCC0H
```

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0

RVAL is assigned 0FCF0H

1	1	1	1	1	0	1	0	1	1	1	1	0	0	0	0

!! Exclusive-OR

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

Example:

```
ERVAL  EQU    0F0F0H !! 0CCC0H
```

1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0

ERVAL is assigned 5A50H

0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	0

Relational Operators

Relational operators compare two terms and return the value −1 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 cannot be compared.

```
START  MVI        A,"?"
        .
        .
        .
NEXT   MVI        H,0000
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 two characters of the string are converted to a scalar value and a numeric comparison is performed.

The types of comparisons are summarized in Table 4-3.

Table 4-3
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

  ENDOF—Returns the address of the last byte of a section.
  HI—Returns the high byte of an address.
  LO—Returns the low byte of an address.
  SCALAR—Converts an address value to a scalar value.

- String Functions

  NCHR—Returns the current length of a string variable.
  SEG—Returns a substring of a string.
  STRING—Converts a scalar value to a string.

Each of these functions is described in detail in the following pages. The same conventions as described in the Assembler Introduction section of this manual are used in these descriptions.

Function: BASE

Determines whether two values have common base

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         }
       IF         BASE(Q,R)
         .                    } Statements assembled
         .                    } because Q and R share
         .                    } common base
       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 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
        .                               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.

Function: DEF

Determines if symbol is defined

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 Assembler Introduction section of this manual for a description of the two passes of the assembler.) A value of -1 (true) is returned if the symbol is defined. A value of 0 (false) is returned if the symbol is not defined.

EXAMPLES

```
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. Therefore, 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.

Function: ENDOF

Returns end address of section

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.

EXAMPLES

```
Label  Operation   Operand

       RESERVE     STACK,100
       LXI         SP,ENDOF(STACK)
```

This 8080A 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 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 of a numeric expression. The result is a one-byte numeric value. 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.

EXAMPLES

```
Label  Operation   Operand

       SECTION     TABLE,INPAGE
Q      BLOCK       50
R      BLOCK       50
       SECTION     MAIN
       MVI         H,HI(TABLE)
         .
         .
         .
       MVI         L,LO(Q)
       MOV         A,M
       MVI         L,LO(R)
       MOV         M,A
```

In this 8080A example, the high byte of the beginning address of the section TABLE is loaded into the H register. Both Q and R will have the same high byte because INPAGE defines the section to be within one page. (See the Assembler Directives section for more information on the INPAGE relocation-option for sections.) The low byte of the addresses for Q and R is loaded into the L register. Data can be transferred without changing the H register.

Function: LO

Returns low byte of numeric value

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. 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.

EXAMPLES

See the HI function example.

Function: NCHR

Returns number of characters in string

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 numeric 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.

EXAMPLES

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 respresentation 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, and subtraction. To perform any other operations on address values, you must first convert the addresses to scalar values with the SCALAR function.

EXAMPLES

```
Label  Operation   Operand

TABLE  BLOCK       100
XXX    EQU         SCALAR(TABLE) / 2 + TABLE
```

This example shows an arithemetic operation (division by 2) performed on an address value. The address value is converted to a scalar value by the SCALAR function.

Function: SEG

Returns substring

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.
char-count	Any numeric expression that evaluates to the number of characters to be returned.

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 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"

EXAMPLES

```
Label  Operation  Operand

       STRING     STR(12), LST(1)
STR    SET        "CHARACTERS"
LST    SET        SEG(STR,NCHR(STR),1)
```

number of characters to be returned

first character of substring (NCHR function returns the number of characters in STR)

character string

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.

Function: STRING

Converts scalar to string

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 six characters long. The first character is a zero or minus (-) depending on the sign of the number. The remaining five 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	"000000"
-1	"-00001"
400	"000400"
200H	"000512"

EXAMPLES

```
Label   Operation   Operand

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

This example converts the value of XVAL times YVAL (4 * 50 = 200) to the string "000200". DIGIT4 is defined to be the fourth character in the string MATSIZE ("2")

Section 5
ASSEMBLER DIRECTIVES

ASSEMBLER DIRECTIVE INDEX

Page

Listing Control Directives

Symbol Definition Directives

Data Storage Directives

Macro Definition Directives

Page

File Inclusion Directive

Conditional Assembly Directives

Module Definition Directives

ILLUSTRATIONS

Fig.
No. Page

Section 5

ASSEMBLER DIRECTIVES

INTRODUCTION

This section describes the directives you may use when programming for the Tektronix 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 consists 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 Assembler Introduction section of this manual.

Label	Operation	Operand
[symbol]	DIRECT	string-expression[,string-expression]...

The above example shows the syntax for DIRECT, a fictional 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 string expression. If two or more string expressions are entered, they must be separated by commas. The number of string expressions is limited only by the maximum line length (127 characters).

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.
- Only the ENDM directive **must not have a label**.
- The following directives generate object code and therefore **often have labels**. The label is assigned the address of the first byte of code generated.

  ASCII BLOCK BYTE WORD

- 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

- All other directives (listed below) do not even 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.

  ELSE
END
ENDIF
ENDR
EXITM
GLOBAL
IF
INCLUDE
LIST
MACRO
NAME
NOLIST
PAGE
REPEAT
SPACE
STITLE
STRING
TITLE
WARNING

THE ASSEMBLER DIRECTIVE DICTIONARY

SYNTAX

Label	Operation	Operand
[symbol]	ASCII	string-expression[,string-expression]...

PARAMETERS

symbol — A user-defined label representing the address of the first character in the string.

string-expression — Any expression that yields a string value.

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.

EXAMPLES

```
Label      Operation  Operand

CHESSMEN   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. CHESSMEN 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. 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, it is truncated to 8 bits and the following message is displayed:

```
***** ERROR 035: Value truncated to byte
```

EXAMPLES

```
Label      Operation  Operand

MONTHS     BYTE       31,28,31,30,31,30
           BYTE       31,31,30,31,30,31
```

Twelve 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.

SYNTAX

Label	Operation	Operand
[symbol]	COMMON	section-name[,relocation-type]

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

PAGE—The common section is relocated to the beginning of a page of memory. See the Assembler Specifics section of this manual for the page size for your microprocessor.

INPAGE—The common 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.

If you do not specify PAGE, INPAGE, or ABSOLUTE, the relocation type defaults to byte-relocatable: the relocated common section may begin at any byte in memory.

EXPLANATION

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

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 ASCII, BYTE, 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 linked last takes precedence; neither the linker nor the LOAD command flags the error.

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.

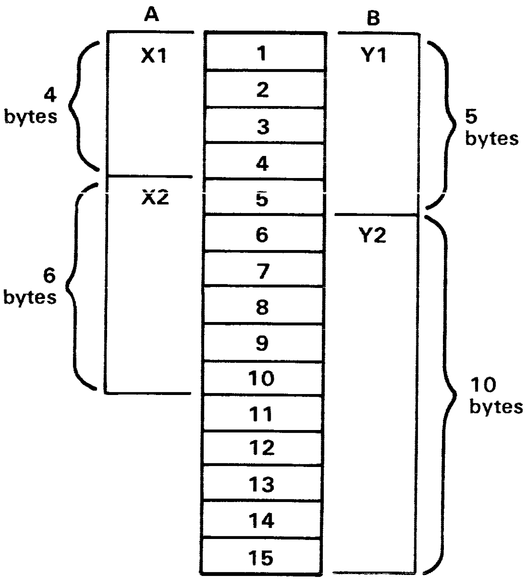
In source module A, the following statements define common section SCRATCH:

```
Label       Operation   Operand

            COMMON      SCRATCH
X1          BLOCK       4
X2          BLOCK       6
```

In source module B, SCRATCH is defined as follows:

```
            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, which is the length of the larger section named SCRATCH. Both subroutines may use this area of memory.



3575-6

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
```

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  source:  J  A  N  F  E  B  M  A  R  A  P  R  M  A  Y  J  U  N  \
1-18   object:  4A 41 4E 46 45 42 4D 41 52 41 50 52 4D 41 59 4A 55 4E  |
                                                                        } MONTHS
bytes  source:  J  U  L  A  U  G  S  E  P  O  C  T  N  O  V  D  E  C   |
19-36  object:  4A 55 4C 41 55 47 53 45 50 4F 43 54 4E 4F 56 44 45 43  /

bytes  source:  S  U  N  M  O  N  T  U  E  W  E  D  T  H  U  F  R  I   \
37-54  object:  53 55 4E 4D 4F 4E 54 55 45 57 45 44 54 48 55 46 52 49  |
                                                                        } DAYS
bytes  source:  S  A  T                                                 |
55-57  object:  53 41 54                                                /
```

ELSE

Begins alternate conditional block

SYNTAX

Label	Operation	Operand
[symbol]	ELSE	

EXPLANATION

The ELSE directive separates the branches of an IF...ELSE...ENDIF block.

If the conditional expression in the IF directive is nonzero (true), the statements between the IF directive and the ELSE directive are assembled. Otherwise, the statements between the ELSE directive and the ENDIF directive are assembled.

EXAMPLES

```
Label    Operation  Operand    Comment

         IF         YEAR MOD 4 = 0
NDAYS    EQU        366        ; LEAP YEAR     ◄——— Assembled if YEAR is
         ELSE                                       divisible by 4.
NDAYS    EQU        365        ; NOT LEAP YEAR ◄——— Assembled if YEAR is
         ENDIF                                      not divisible by 4.
```

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.

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 ASM command line.

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 TRANSFER command in 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     XRA        A
           .
           .
          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 8080A instruction XRA A.

SYNTAX

Label	Operation	Operand
[symbol]	ENDIF	

EXPLANATION

The ENDIF directive marks the end of an IF...ENDIF or IF...ELSE...ENDIF block of statements. See the IF directive.

SYNTAX

Label	Operation	Operand
	ENDM	

EXPLANATION

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

The ENDM directive must **not** have a label.

SYNTAX

Label	Operation	Operand
[symbol]	ENDR	

EXPLANATION

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

EQU
Assigns value to symbol

SYNTAX

Label	Operation	Operand
symbol	EQU	numeric-value

PARAMETERS

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

numeric-value — Any numeric expression. Each symbol in the expression must have been defined previously.

EXPLANATION

The EQU directive assigns a value to a symbol. 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: a BLOCK, EQU, IF, ORG, REPEAT, SET, or STRING directive that refers to the symbol must not precede the EQU directive that defines the symbol.

EXAMPLES

```
Label      Operation  Operand     Comment

           MVI        B,ROWS      ; NUMBER OF ROWS TO B REGISTER.
           MVI        C,COLS      ; NUMBER OF COLUMNS TO C 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 8080A MVI 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
[symbol]	EXITM	

EXPLANATION

The EXITM directive terminates the current macro **expansion**; 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
```

GLOBAL

Declares global symbol(s)

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
           CALL       HIM
           CALL       HER
           CALL       MYSELF
             .
             .
MYSELF     XRA        A
             .
             .
```

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 8080A XRA instruction) and is not used in any other module.

```
           NAME       HISMOD
           GLOBAL     HIM,VALUE
HIM        MVI        A,VALUE
             .
             .
```

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

```
           NAME       HERMOD
           GLOBAL     HER,HIM
HER        CALL       HIM
             .
             .
```

In module HERMOD, HIM is an unbound global. HER is defined as the address of the CALL 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 and true if the value is nonzero.

EXPLANATION

The IF directive marks the beginning of an IF...ENDIF or IF...ELSE...ENDIF block of statements. The value of the expression in the IF directive determines which statements (if any) in the block are assembled.

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), those statements are skipped. (See Example 1.)

IF...ELSE...ENDIF

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

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

If the condition-value is true (nonzero), the statements between the IF 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 (16 1-bits) when true and the value 0 (16 0-bits) when false. Thus the bit manipulation operators &, !, and !! may be used as the conjunctions AND, OR, and exclusive-OR, respectively, in complex relational expressions. (See Example 3.) The Language Elements section of this manual explains expressions and operators in detail.

Each IF directive must have a corresponding ENDIF directive and may have one corresponding ELSE directive.

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. See Fig. 5-1, which illustrates the allowed forms of nesting for IF blocks.

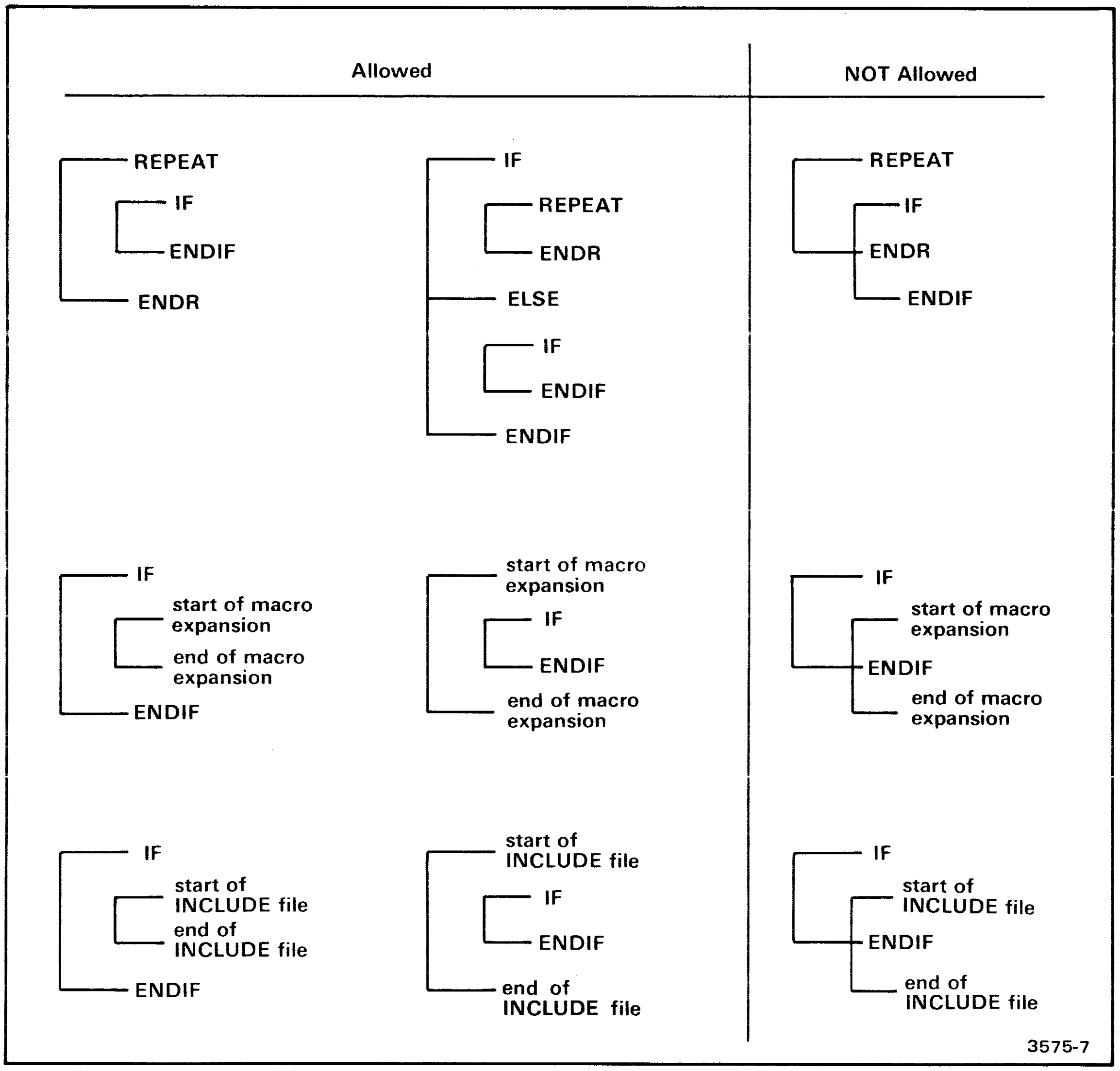


Fig. 5-1. Allowed forms of IF block nesting.

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

In this example, a warning is displayed at assembly time if the object code for section ASEC occupies more than one page of memory. The dollar sign represents the current value of the location counter.

```
Label       Operation  Operand

PAGESIZE    EQU        100H
            SECTION    ASEC,INPAGE
             .
             .
            IF         $ >= PAGESIZE
            WARNING ; SECTION ASEC TOO LONG
            ENDIF
```

IF Example 2

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

```
Label       Operation  Operand     Comment

            MACRO      WORDS
             .
             .
            IF         "'1'" = ""  ; IF FIRST PARAMETER IS ABSENT...
              WORD     0           ; ... STORE A WORD OF ZEROS.
            ELSE
              WORD     '1'         ; OTHERWISE STORE FIRST PARAMETER.
            ENDIF
             .
             .
            ENDM
```

The construct '1' is replaced by the first parameter. If there is no first parameter, '1' is replaced with the null string (nothing); since the expression "" = "" is true, the statement WORD 0 is assembled and the statement WORD '1' is skipped. On the other hand, if the parameter exists, the second WORD directive is assembled, taking the parameter as its operand.

IF Example 3

```
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. (& is the logical AND operator.) 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.

INCLUDE

Assembles source code from another file

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.

The INCLUDE file may not contain an INCLUDE directive.

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 for information about these constructs.

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 file MACR.ASM in the current directory (/USR) and file COM.ASM in the system directory are assembled at the beginning of module MAINMOD. MACR.ASM contains macro definition blocks; COM.ASM defines a common section.

SYNTAX

Label	Operation	Operand
[symbol]	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 linker listing to include an internal symbols list for this module at link time. Defaults to OFF.

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 its default setting, MEG: lists only those macro expansion statements that generate object code.

SYM—Lists the symbol table. Defaults to ON.

TRM—Trims the assembler listing to 72 characters. The listing width defaults to 72 characters wide if the listing device is CONO; otherwise the width defaults to 132 characters.

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. If no option is named, the master option (which controls the source listing) is turned on. The NOLIST directive may be used to turn any of these options off.

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 expansion, its previous setting is restored when the expansion ends.

An assembler listing contains two parts: the **source listing**, which shows the source code and object code for each statement assembled; and the **symbol table**, which lists the symbols used in the source module. The master option, CND option, and ME/MEG option determine which lines of code appear in the source listing, and are discussed in the following paragraphs. The SYM option controls display of the symbol table, and is discussed with the CON, DBG, and TRM options under the heading **Other Options**.

Source Listing

Master Option. The master option 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, 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, and 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 ASCII, BLOCK, BYTE, 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 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.

Summary. The following table shows how the master option and CND option control the display of statements outside macro expansions.

Option Settings		
Master	**CND**	**Type of Statements Listed**
OFF	[a]	errors
ON	OFF	assembled statements (default)
ON	ON	all statements

[a] don't care

The following table shows how the master option, ME/MEG option, and CND option control the display of statements in a macro expansion.

Option Settings			
Master	ME/MEG	CND	Type of Statements Listed
OFF	[a]	[a]	errors
ON	OFF	[a]	errors
ON	MEG	[a]	statements that generate object code (default)
ON	ME	OFF	assembled statements
ON	ME	ON	all statements

[a] don't care

Other Options

CON. Normally the CON option is ON, and every erroneous statement and its accompanying error message are displayed on the system terminal (CONO) 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 left at its default setting (OFF), the linker listing will contain no internal symbols list for the current module when the module is linked. If the DBG option is ON when assembly ends, an internal symbols list will be created, and it will list all symbols in the module. The internal symbols list is described in the Linker section of this manual.

SYM. If the SYM option is left at its default setting (ON), 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 Introduction section.

TRM. Normally, the TRM option is OFF, and the assembler listing contains lines of up to 132 characters. When the TRM option is ON, all lines are truncated to 72 characters. (Source lines that contain more than about 50 characters are truncated, since the source listing displays 20 to 25 characters of information—depending on your microprocessor—to the left of each source line.) If TRM is ON when assembly ends, the symbol table is rearranged to fit a 72-character format.

When the listing device is the system terminal (CONO), the TRM option is automatically turned ON before assembly starts. The directive NOLIST TRM restores the 132-character format.

EXAMPLES

```
Label  Operation  Operand

       LIST       DBG
```

This statement causes the linker to display an internal symbols list for this module when it is linked.

```
       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 and the LIST directive turns it 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.

SYNTAX

Label	Operation	Operand
[symbol]	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 number in the specified 8080A register to its two's complement:

```
 Label      Operation  Operand  Comment

            MACRO      NEGATE
            SUB        A        ; SET A TO ZERO.
            SUB        '1'      ; SUBTRACT '1' FROM ZERO.
            MOV        '1',A    ; STORE RESULT BACK INTO '1'.
            ENDM
```

The macro invocation

```
            NEGATE     B
```

yields the following macro expansion:

```
            SUB        A         ; SET A TO ZERO.
            SUB        B       ; SUBTRACT B FROM ZERO.
            MOV        B,A     ; STORE RESULT BACK INTO B.
```

Every occurrence of the first formal parameter ('1') is replaced by the first actual parameter (B). The 8080A instruction SUB A clears the A register; SUB B subtracts the contents of the B register from the A register; MOV B,A moves the result back into the B register.

NAME
Declares object module name

SYNTAX

Label	Operation	Operand
[symbol]	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
[symbol]	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 displav of assembly errors on the system terminal.

DBG—Suppresses the internal symbols list for this module at link time.

ME—Suppresses display of all macro expansion statements.

MEG—Suppresses display of all macro expansion statements.

SYM—Suppresses listing of the symbol table.

TRM—Changes the listing width from 72 characters to 132 characters.

If no option is specified, the source listing is turned off.

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.

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. 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. Avoid using ORG in a byte-relocatable or inpage-relocatable section, since the conditions 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.

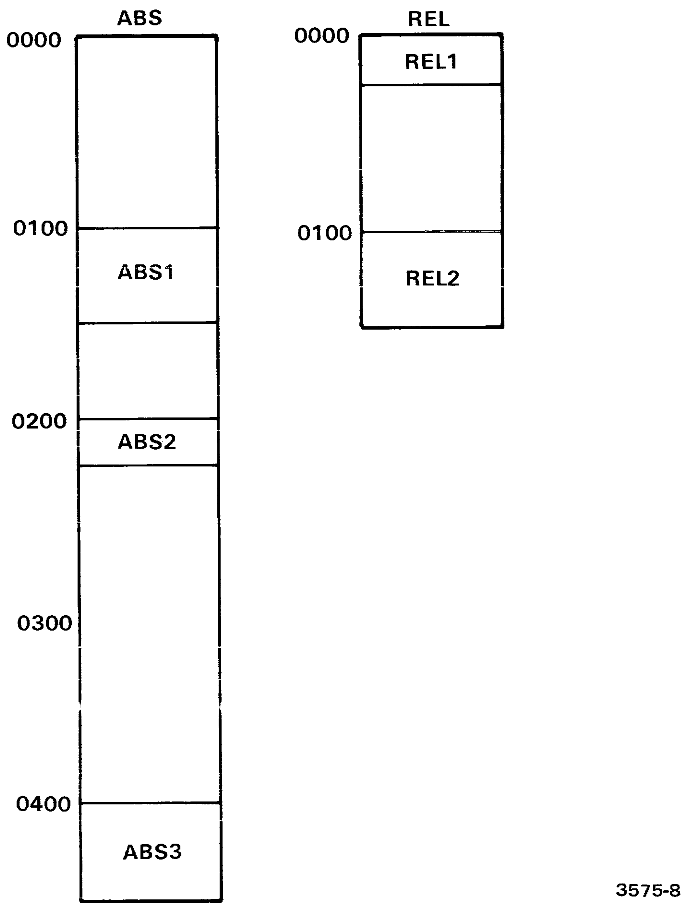
If you use ORG incorrectly, you may end up specifying more than one value for the same byte of object code. (See Example 2.) Such a situation is not detected by the assembler, linker, or LOAD command.

EXAMPLES

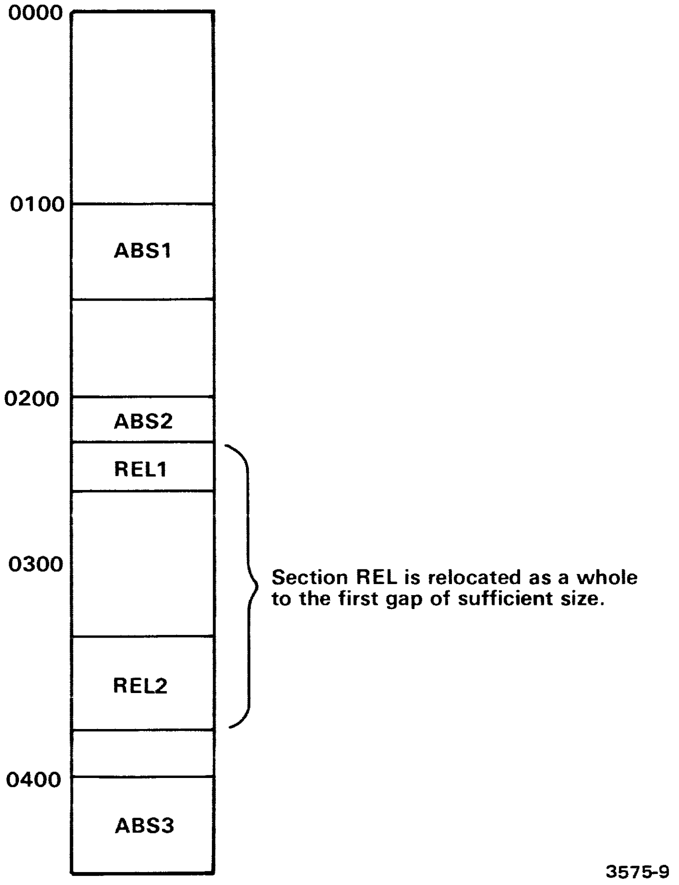
ORG Example 1

```
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 the example above, two sections of object code are generated. Section ABS is divided into three blocks and section REL is divided into two blocks. The layout of the two sections is shown below.



The linker will arrange the two sections as shown below.


0000
0100
ABS1
0200
ABS2
REL1
0300
REL2
0400
ABS3
Section REL is relocated as a whole to the first gap of sufficient size.
3575-9

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 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"
```

PAGE

Skips to new page in listing

SYNTAX

Label	Operation	Operand
[symbol]	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  xxxxxxxxx ASM Vx.x  THIS IS THE TITLE            Page    x

xxxxx                         SECTION    MAIN     ; BEGIN CODE FOR MAIN.
```

SYNTAX

Label	Operation	Operand
[symbol]	REPEAT	condition-value[,limit]

PARAMETERS

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

limit — The maximum number of times the block may repeat: any non-negative scalar expression. Defaults to 255.

EXPLANATION

The statements between a REPEAT directive and its matching ENDR directive are assembled repeatedly until the condition-value becomes false (zero). A REPEAT...ENDR block is valid only within a macro.

If the condition-value is still true (nonzero) after the repetition limit has been reached, the assembler responds

```
 ***** ERROR 017: Iteration limit exceeded
```

and skips to the statement following the ENDR directive.

If the condition-value is false before the first repetition, the REPEAT...ENDR block is not assembled at all

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. See Fig. 5-2, which illustrates the allowed forms of nesting for REPEAT blocks.

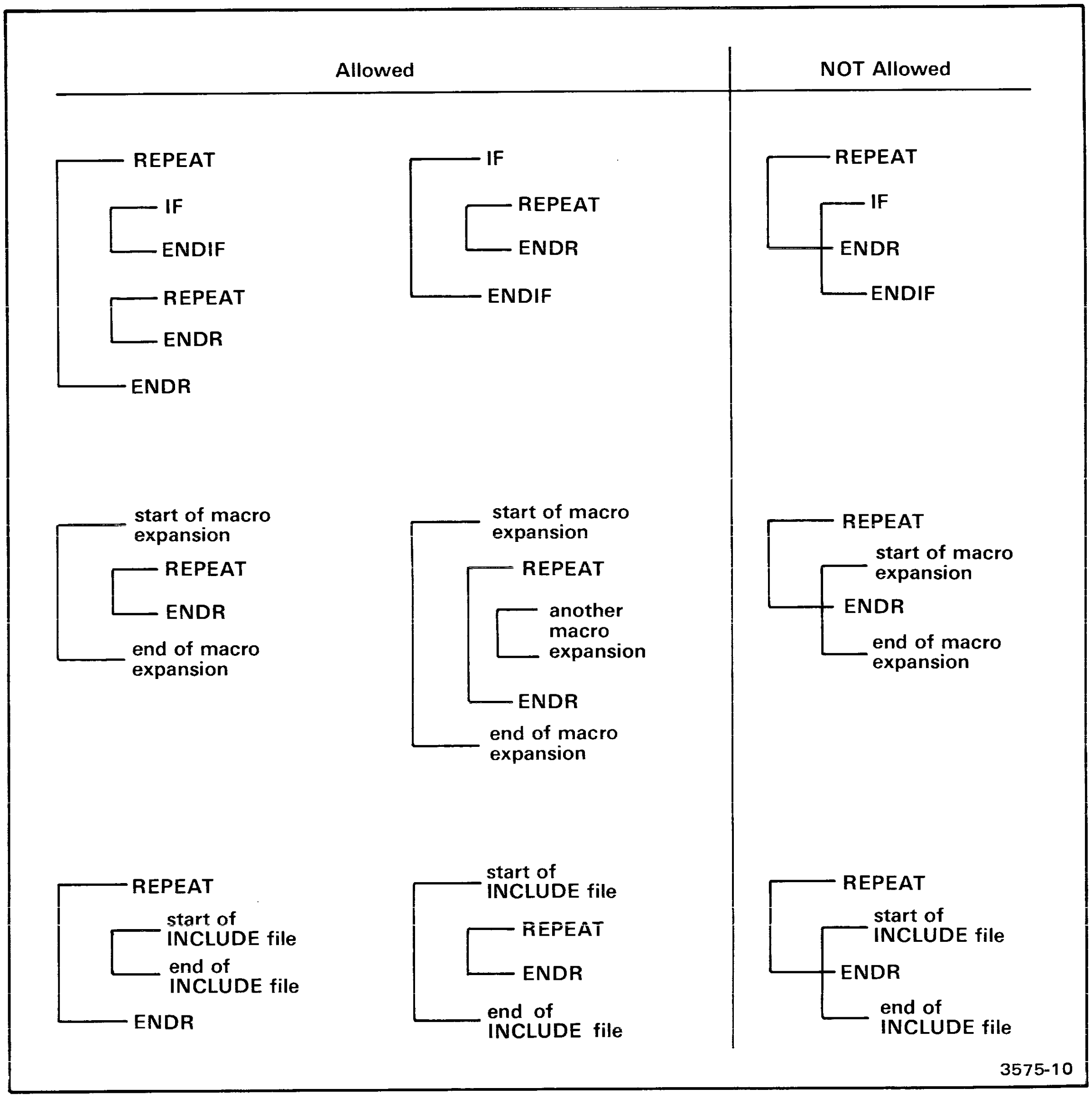


Fig. 5-2. Allowed forms of REPEAT block nesting.

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.

EXAMPLES

```
Label       Operation  Operand

            MACRO      LOOP
COUNT       SET        1
            REPEAT     COUNT <= '1'
            BYTE       '2'
COUNT       SET        COUNT + 1
            ENDR
            ENDM
```

The statement

```
            LOOP       3,0
```

invokes the above macro and produces the following expansion:

```
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.

RESERVE

Reserves section of memory

SYNTAX

Label	Operation	Operand
[symbol]	RESERVE	section-name,section-length[,relocation-type]

PARAMETERS

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

section-name — The name assigned to the section.

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. 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 relocation type defaults to byte-relocatable: the relocated section may begin at any byte in memory.

EXPLANATION

The RESERVE directive creates a section with the specified name, length, and relocation type. 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 chiefly to set aside memory for a workspace or stack.

A reserve section may not have the relocation type ABSOLUTE; 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 everywhere 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.

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.

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.

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.
           .
           .
          STA        TEMP       ; 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]

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. 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.

ABSOLUTE—The section is not relocated.

If you do not specify PAGE, INPAGE, or ABSOLUTE, the relocation type defaults to byte-relocatable: the relocated section may begin at any byte in memory.

EXPLANATION

The SECTION directive declares a section of type SECTION and defines the name and relocation type 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.

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 default section is a byte-relocatable SECTION; its name is derived as follows:

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

For example, the default section for object file MY.OBJ is %MYOBJ. 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
or		
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 16-bit value formed by the first two bytes of the string. If the string exceeds two characters, the assembler responds

  ```
  ***** ERROR 085: String value too large
  ```

  If the string contains only one character, its ASCII code is copied to the low-order byte of the variable and the high-order byte is set to zero.

- If you assign a numeric value to a string variable, the STRING function is automatically invoked to convert the number to a six-digit string.

Text substitution (signaled by single quotes ' ') often involves variables. A string variable in single 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.

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 example below, 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 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
```

SPACE

Inserts blank lines into listing

SYNTAX

Label	Operation	Operand
[symbol]	SPACE	[line-count]

PARAMETERS

line-count — The number of blank lines to be generated: any expression that yields a scalar in the range 0 to 255. Defaults to 1.

EXPLANATION

The SPACE directive generates the specified number of blank lines in the source listing. If no line count is given, one line is generated. The SPACE directive itself is not listed.

If the line count exceeds the number of lines left on the current page, the SPACE directive merely skips to the top of the next page.

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

EXAMPLES

```
Label      Operation  Operand

                .
                .
; END OF SECTION AAAA.
           SPACE      5
           SECTION    BBBB
; BEGIN SECTION BBBB.
```

These lines of code will be listed as follows:

```
; END OF SECTION AAAA.
                           }
                           }  5 blank lines
                           }
                           }

           SECTION    BBBB
; BEGIN SECTION BBBB.
```

SYNTAX

Label	Operation	Operand
[symbol]	STITLE	subtitle-string

PARAMETERS

subtitle-string	The subtitle for the source listing: any expression that yields a string of up to 72 characters.

EXPLANATION

The STITLE directive creates a subtitle of up to 72 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 72 characters, only the first 72 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"
```

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

```
Tektronix  xxxxxxxxx ASM Vx.x  THIS IS THE TITLE              Page     1
SUBTITLE FOR PAGES 1 AND 2

00003                 ; THIS IS THE FIRST LISTABLE LINE.



Tektronix  xxxxxxxxx ASM Vx.x  THIS IS THE TITLE              Page     2
SUBTITLE FOR PAGES 1 AND 2



Tektronix  xxxxxxxxx ASM Vx.x  THIS IS THE TITLE              Page     3
SUBTITLE FOR PAGE 3
```

STRING
Declares string variable(s)

SYNTAX

Label	Operation	Operand
[symbol]	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 8.

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 a maximum length is not specified, it defaults to eight 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 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(20)
             .
             .
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, 8, and 20, 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").

SYNTAX

Label	Operation	Operand
[symbol]	TITLE	title-string

PARAMETERS

title-string The title for the source listing: any expression that yields a string of up to 30 characters.

EXPLANATION

The TITLE directive creates a title of up to 30 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 30 characters, only the first 30 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 above statements produce the following page headings in the source listing:

```
Tektronix  xxxxxxxxx ASM Vx.x  THE SAME OLD TITLE           Page    1
THE SAME OLD SUBTITLE


Tektronix  xxxxxxxxx ASM Vx.x  THE SAME OLD TITLE           Page    2
THE SAME OLD SUBTITLE


Tektronix  xxxxxxxxx ASM Vx.x  A NEW TITLE                  Page    3
THE SAME OLD SUBTITLE
```

WARNING

Displays warning

SYNTAX

Label	Operation	Operand
[symbol]	WARNING	[;message]

PARAMETERS

message Any user-defined error message.

EXPLANATION

When a WARNING directive is assembled, it is treated as an erroneous statement: the WARNING line and the message

```
***** ERROR 001:
```

are displayed on the system terminal and in the source listing.

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

EXAMPLES

```
Label      Operation  Operand

PAGESIZE   SET        100H
           SECTION    SEC1,INPAGE
             .
             .
           IF         $ >= PAGESIZE
           WARNING ; SECTION '%' TOO LONG
           ENDIF
```

In this example, section SEC1 must not exceed one page in length. If the location counter ($) has exceeded its maximum when the IF directive is assembled, the WARNING is assembled and the following message is displayed:

```
xxxxx                               WARNING ; SECTION SEC1 TOO LONG
***** ERROR 001:
```

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

WORD

Generates word(s) of data

SYNTAX

Label	Operation	Operand
[symbol]	WORD	word-value[,word-value]...

PARAMETERS

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

word-value — Any expression that yields a number in the range -32768 to 65535.

EXPLANATION

The WORD directive stores the specified values in consecutive words of the object program. Each word consists of two bytes. The low-order byte of the word may precede or follow the high-order byte, depending on the convention for your microprocessor.

Each value may be a scalar in the range -32768 to +32767 or an address in the range 0 to 65535. Any value outside the range -32768 to 65535 is truncated to 16 bits without notice.

EXAMPLES

```
Label      Operation  Operand

YEARS      WORD       1775,1812,1861
POINTER    WORD       TABLE
             .
             .
TABLE      BLOCK      12
```

In this example, the first statement stores three two-byte numbers: 1775, 1812, and 1861. YEARS is the address of the first byte of 1775.

The second statement stores the address of a 12-byte table. POINTER is the address of the address stored. A microprocessor with indirect addressing can refer to the table by its address (TABLE) or by its indirect address (POINTER).

Section 6
MACROS

Illustration

Section 6

MACROS

INTRODUCTION

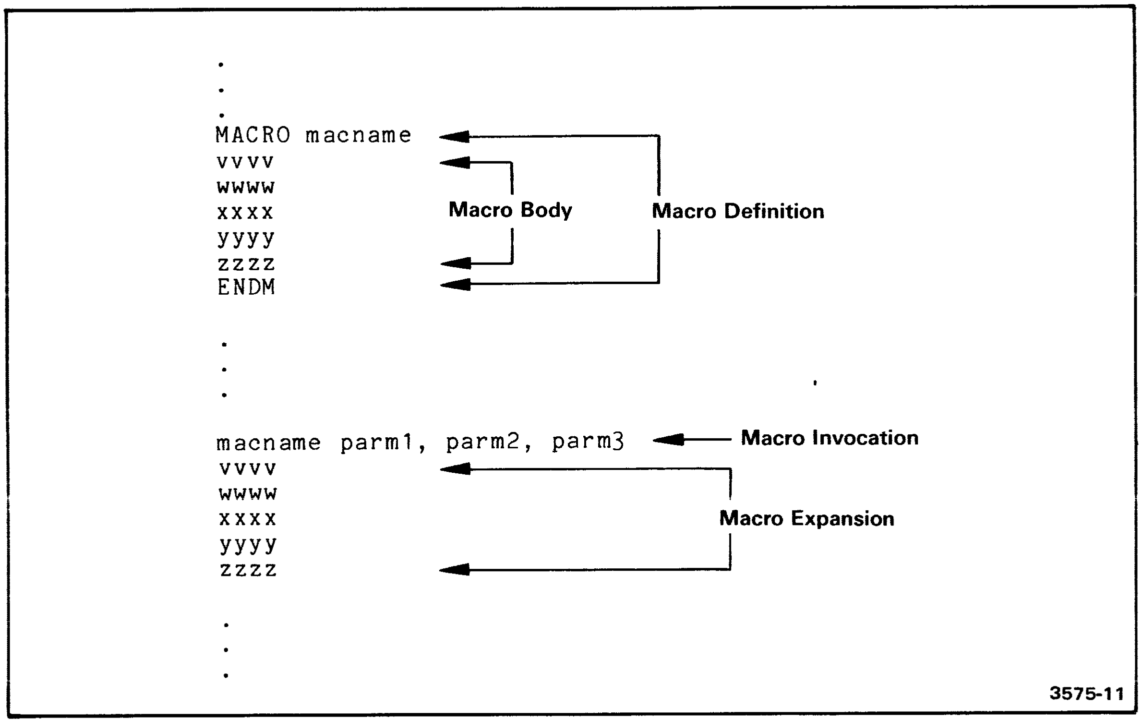
A macro is a frequently used sequence of assembler statements. Once a group of statements is defined as a macro in the beginning of your assembly language program, the macro can be invoked many times.

A macro is invoked with a single line, which generates zero or more lines of assembler statements. This invocation is called the macro expansion process. The macro can make use of parameters given in the macro invocation line; with conditional assembly, the macro may expand differently with each invocation.

This section describes macro definition, invocation, and expansion. An overview of the entire process is given, followed by a detailed description of each phase of the process. The last part of this section gives examples of macro usage.

MACRO EXPANSION PROCESS

The macro expansion process is illustrated in Fig. 6-1.



```
.
.
.
MACRO macname
vvvv
wwww
xxxx
yyyy
zzzz
ENDM
.
.
.
macname parm1, parm2, parm3
vvvv
wwww
xxxx
yyyy
zzzz
.
.
.
```

Fig. 6-1. Sample macro usage.

The three phases of macro usage are definition, invocation, and expansion.

Definition. A macro is defined with the MACRO directive. The macro is given a name ("macname" in the figure) that is used later to invoke the macro. The sequence of assembler statements that make up the macro follows the MACRO directive ("vvvv", "wwwww", "xxxx", "yyyy" and "zzzz" in the figure). This sequence of statements is sometimes called the body of the macro. An ENDM directive terminates the definition.

The assembler saves the macro name and its associated body for later invocation. The contents of the body are ignored until expansion time.

Invocation. The macro is invoked when the macro name appears in the operation field of an assembly language statement. One or more parameters may follow the macro name ("parm1", "parm2", and "parm3" in the example). These parameters may be used by the body of the macro to control the expansion process.

Expansion. Each line from the macro body is inserted into your assembler source program, as if the program were cut apart at the macro invocation and the invocation line replaced with the entire macro body. The assembler then interprets the statements within the body as if they were part of the original source program. Any line of the body may reference the parameters passed to the macro at invocation time; these references can be used to alter the contents of each assembler line.

MACRO DEFINITION

You define a macro once in your program—before its first use. The macro definition consists of three parts:

- the MACRO assembler directive, which gives the name of the macro,
- the sequence of statements constituting the body of the macro,
- the ENDM assembler directive, which terminates the macro definition.

You must define any macro prior to its first invocation. You cannot define a macro within another macro definition.

The MACRO Directive

The MACRO assembler directive begins a macro definition. The format of the MACRO directive is:

```
            MACRO   name    ; comments here (optional)
```

The name is a standard assembler symbol: a letter, optionally followed by one to seven alphabetic, numeric, dollar sign, underscore, or period characters. Since you use this name to invoke the macro, it is wise to choose a name that indicates the macro's function.

The symbol chosen as the macro's name must be unique—it cannot be identical with any other symbol used within the assembler source file.

The Macro Body

The macro body is a sequence of assembler statements. Any statements, except the MACRO, ENDM, and END directives, may be included in the body. The statements can include processor instructions, assembler directives, invocations of other macros, or even invocations of the given macro.

Comments and blank lines within the macro body are discarded, since they do not affect macro expansion.

Macro Body Operators

The macro body can contain special operators not available outside of macro definitions. These special operators give the macro access to assembler values, such as:

- each parameter passed to the macro,
- a unique character sequence for each macro invocation,
- the number of parameters passed to the macro, and
- the current section name.

The following paragraphs describe these operators in detail.

Parameter Access ('1', '2', ...)

The macro can access any parameter given when the macro is invoked. Parameters are identified with consecutive positive integers, starting at 1 for the leftmost parameter. Within the body of the macro, any number enclosed within single quotes is replaced with the corresponding parameter from the macro invocation line. For example, during macro expansion, any occurrence of '1' in the macro body is replaced with the first parameter.

Text substitution can occur anywhere on the line, including text within the comment field. If text substitution causes the line to exceed 127 characters, an error is generated and the line truncated. Examples of text substitution can be found in the Macro Invocation subsection later in this section.

The value within single quotes may be either a constant or a numeric-valued SET variable. Refer to the description of the SET directive in the Assembler Directives section of this manual for further information on numeric assembler variables.

If the value within quotes is greater than the number of parameters actually provided, a null string is substituted at the time of expansion.

Unique Label Generation (the @ Character)

The "at" character, when enclosed within single quotes ('@'), is used to provide unique labels for each macro expansion. Each time a macro is invoked, the '@' construct is replaced with a unique four-character value. When this value is appended to a one-to-four character symbol within the macro body, a unique five-to-eight character label is created for each invocation. In the following example, a unique seven-character label is generated each time the macro is invoked. That label is used as the destination of a processor jump instruction:

```
          MACRO   Q
          .
          .
LAB'@'    EQU     $
          .
          .
          JNZ     LAB'@'
          .
          .
          ENDM
```

If "LAB" had not been followed by the '@' construct in this example, the first invocation of macro Q would have defined the location of LAB. Any subsequent invocations would attempt to redefine the location of LAB, resulting in an error.

Determining Parameter Count (the # Character)

The "pound" character, when enclosed within single quotes ('#'), is replaced at time of expansion with a five-digit number. This number represents the total number of parameters passed to the current macro expansion. For example, if three parameters are passed to the macro, '#' is replaced with 00003 during macro expansion.

You can use the '#' in an assembler IF or REPEAT directive to cause conditional expansion of the macro to depend on the number of parameters passed. Examples of '#' usage are given in the Macro Examples subsection, at the end of this section.

Determining Current Section Name (the % Character)

The "percent" character, when enclosed within single quotes ('%'), is replaced by the name of the current section (as defined with the SECTION or COMMON assembler directives). The section name is given as a sequence of characters. If the current section is the default section, '%' is replaced with a null string.

The '%' construct is usually used when the macro must define instructions or data in a new, distinct section, and then return back to the original section definition. To accomplish this task, the macro must save the name into an assembler string variable, change section names, give the declarations for the new section, and then use a RESUME directive to return to the original section, as illustrated in the following example:

```
        STRING  SECNAME(8)      ; Defines SECNAME as a string of up to
                                ; 8 characters
        .
        .
        .
        MACRO   Q               ; Beginning of macro definition
        .
        .
SECNAME SET     "'%'"           ; Save current section name in SECNAME
        SECTION QQ              ; Switch to new section (QQ)
        .
        .
        RESUME  'SECNAME'       ; Switch back to previous section
        .
        .
        ENDM                    ; End macro definition
```

In the above example, the '%' construct is enclosed within double quotes. The SET directive expects a string expression, but the '%' construct is replaced with a sequence of characters. When this sequence of characters is enclosed within double quotes, it becomes a string literal, which is an acceptable string expression.

Disabling Special Character Significance (the ∧ Character)

The up-arrow character (∧), when immediately preceeding any special character, disables the special meaning of that character, and causes the character to be interpreted as part of the text. In the following example, the up-arrow character removes the special significance of the single-quote character:

```
ASCII   "That^'s all, folks!"
```

When the macro is expanded, the following text string is generated in the program:

```
That's all, folks!
```

The ENDM Directive

The macro definition is terminated with the ENDM directive. This directive should not have a label field.

MACRO INVOCATION

A macro is invoked when its name appears in the operation field of an assembler line. For example, the macro QQQ is invoked by the following assembler statement:

```
          QQQ            ; Comments (if used) go out here
```

Parameters

The macro body can make use of information given to the macro at the time of invocation. This information is given as a series of one or more **parameters** in the operand field of the macro invocation. Each parameter is a sequence of characters separated from other parameters by commas. For example, the following assembler statement invokes macro QQQ with parameters of 123 and ABC:

```
          QQQ     123,ABC          ; Invokes macro QQQ
                                   ;  with parameters 123 and ABC
```

As QQQ is expanded, any occurrence of '1' within the macro body is replaced with 123, and any occurrence of '2' is replaced with ABC.

Any number of parameters can be passed to a macro, so long as the invocation line (including the comment) does not exceed 128 characters. Any parameters given in the invocation that are not examined within the body of the macro are simply ignored. Any parameter requested within the body but not given in the invocation is replaced with the null string.

Macro Parameter Conventions

The Square Brackets

Any leading or trailing spaces surrounding a macro parameter are removed upon macro expansion. You may, however, force the spaces to be retained by placing the parameter within square brackets ([ ]). The square brackets group together all text within them as one parameter. The brackets themselves are removed during macro invocation. For example, invoking QQQ with the following assembler line defines the parameters listed below:

```
          QQQ     ABC,  DEF  ,[GHI],[  JKL   ], MNO PQR
```

The parameters are listed below, surrounded by asterisks. The asterisks are not part of the text, however, but are used here to show the leading and trailing spaces.

```
Parameter '1' = *ABC*
Parameter '2' = *DEF*
Parameter '3' = *GHI*
Parameter '4' = *  JKL  *
Parameter '5' = *MNO PQR*
```

A parameter containing a comma must also be surrounded by brackets, or the parameter will be separated into two distinct parameters. For example, the invocation:

```
        QQQ     ABC,DEF,[GHI,JKL]
```

generates the following parameters (again, the asterisks are not part of the parameters):

```
Parameter '1' = *ABC*
Parameter '2' = *DEF*
Parameter '3' = *GHI,JKL*
```

Square brackets may not be nested.

Double Quote Characters

All text enclosed within double quote marks ("") is considered to be a single parameter. The quote marks are **not** removed from the text during macro expansion, but are considered as part of the parameter. For example, QQQ invoked with the assembler line:

```
        QQQ     "ABC","DEF,GHI","  JKL  ",  "MNO"
```

generates the following parameters (again, the asterisks are not part of the parameters):

```
Parameter '1' = *"ABC"*
Parameter '2' = *"DEF,GHI"*
Parameter '3' = *"  JKL  "*
Parameter '4' = *"MNO"*
```

Square brackets can appear within parameters enclosed in quote marks; the brackets in this case are treated as normal text characters. Double quote marks can appear within a parameter surrounded by square brackets; the quote marks are then treated as normal text characters. For example, the macro invocation line:

```
        QQQ     "A[B", [C"D], [ " ], "]"
```

generates the following parameters upon expansion (again, the asterisks are not part of the parameters):

```
Parameter '1' = *"A[B"*
Parameter '2' = *C"D*
Parameter '3' = * " *
Parameter '4' = *"]"*
```

Null Parameters

Two consecutive commas, or two commas separated only by blanks, define a null parameter. The parameter is counted in the total parameter count (for '#'), and returns a null string if requested in the body of the macro. For example, the macro invocation line:

```
        QQQ     ABC,,DEF,     ,GHI,[],JKL
```

Null parameters

generates the following parameters:

```
    Parameter '1' = *ABC*
    Parameter '2' = **          (Null parameter)
    Parameter '3' = *DEF*
    Parameter '4' = **          (Null parameter)
    Parameter '5' = *GHI*
    Parameter '6' = **          (Null parameter)
    Parameter '7' = *JKL*
```

Leading and trailing commas in the parameter list also generate null parameters, as in the following example:

```
        QQQ     ,,ABC,,
```

Null parameters

```
    Parameter '1' = **          (Null parameter)
    Parameter '2' = **          (Null parameter)
    Parameter '3' = *ABC*
    Parameter '4' = **          (Null parameter)
    Parameter '5' = **          (Null parameter)
```

The Up-Arrow Character in Macro Invocations

To include a special character in a macro parameter, the character must be immediately preceded by an up-arrow (/\) character. The up-arrow character disables the special significance of any character, and is removed before the macro is expanded. For example, invoking QQQ with the assembler line:

```
        QQQ     A^,B , ^[CD^], ^"EF, G^^H, I^'J, K^^^'L
```

generates the following parameters:

```
    Parameter '1' = *A,B*
    Parameter '2' = *[CD]*
    Parameter '3' = *"EF*
    Parameter '4' = *G^H*
    Parameter '5' = *I'J*
    Parameter '6' = *K^'L*
```

MACRO EXAMPLES

The examples in this subsection illustrate macro usage through typical macro definitions and expansions.

Example 1: Simple Macro Invocation

In this example, macro QQQ is defined. QQQ contains two assembler statements: a BYTE directive and a WORD directive. The operands for these assembler statements are obtained from the parameters given with each invocation of QQQ.

```
MACRO   QQQ      ; Beginning of definition

BYTE    5, '1'   ; BYTE directive, with a fixed operand
                 ; of 5, and an operand provided by the first
                 ; parameter of QQQ at invocation

WORD    '2'      ; WORD directive, with operand provided by
                 ; second parameter of QQQ at invocation

ENDM             ; End of macro definition
```

Invoking this macro with the following assembler statement:

```
QQQ     35, 40
```

produces the following sequence of assembler statments upon macro expansion:

```
BYTE    5, 35
WORD    40
```

During expansion, each occurrence of '1' is replaced with 35 before the assembler statement is processed. Each occurrence of '2' is similarly replaced with 40. The resulting BYTE and WORD directives are then processed as if the assembler statements had been part of the original source text.

Example 2: Nested Macro Invocations

In this example, an assembler statement in the body of one macro invokes another macro.

```
MACRO   Q1              ; Beginning of Q1 definition

WORD    '1', 0          ; Generate a word containing
                        ;  the first parameter, and a
                        ;  second word containing zero

ENDM                    ; End of Q1 definition

MACRO   Q2              ; Beginning of Q2 definition

Q1      '1'             ; Invoke Q1 with first parameter
Q1      '2'             ; and again with second parameter

ENDM                    ; End of Q2 definition
```

Invoking Q2 with the following assembler statement:

```
Q2      3, 5
```

generates the following equivalent assembler source statements during the expansion process:

```
WORD    3, 0
WORD    5, 0
```

The assembler performs the following steps during evaluation of the Q2 invocation line given above:

- Q2 is invoked, with parameters of 3 and 5.
- The first statement in the body of Q2 is examined. This statement contains a reference to the first parameter, so the appropriate parameter (the number 3) is substituted before proceeding.
- The statement invokes macro Q1, with a parameter of 3.
- Q1 is invoked, and the first (and only) statement of Q1 is examined. This statement contains a parameter reference, so the appropriate parameter (3) is substituted.
- The resulting assembler statement (WORD 3, 0) is processed, generating two words of memory.
- Expansion of Q1 terminates, and expansion of Q2 resumes with the second line in its body.
- This line of Q2 has a reference to the second parameter, so the appropriate parameter (5) is substituted before further processing.
- The assembler statement invokes Q1, with a parameter of 5.
- Q1 is invoked as described above, resulting in the assembler statement " WORD 5,0".
- When the expansion of Q1 is completed, expansion of Q2 resumes.
- Q2 contains no further statements in its body, so expansion of Q2 also terminates.

Example 3: Conditional Macro Expansion

In this example, a macro expands one of two different ways, depending on whether one of its parameters is present of absent. Macro QQ generates three WORDs of its first parameter, followed by one WORD of its second parameter. If the second parameter does not exist (or is null), one word of 13 (decimal) follows the first three words.

```
MACRO   QQ                ; Beginning of definition

WORD    '1', '1', '1'     ; Generate three words of the first parameter

IF      "'2'"=""          ; If the second parameter is null:

WORD    13                ;   generate a word of 13

ELSE                      ; Else, (if the second parameter is not null):

WORD    '2'               ;   generate a word of the second parameter.

ENDIF                     ; Terminate the conditional assembly block

ENDM                      ; Terminate the macro definition
```

Invoking this macro with the following assembler statement:

```
          QQ      5, 24
```

generates the following assembler statements:

```
          WORD    5, 5, 5
          WORD    24
```

Invoking this macro with the following assembler statement:

```
          QQ      7
```

produces the following sequence of assembler statements:

```
          WORD    7, 7, 7
          WORD    13
```

In the first invocation, both parameters are specified. During the expansion of QQ, the IF directive substitutes the appropriate parameter and evaluates the expression "24"="". This expression is false, and the statements between the IF statement and the ELSE statement are skipped.

In the second invocation, the expression at the IF statement reduces to ""="". This expression is true, so the assembler statements between the IF and ELSE are processed, and the statement between the ELSE and ENDIF is skipped.

Example 4: Repetitive Macro Expansion

In this example, a macro performs a single operation on each of its parameters. The macro contains a REPEAT..ENDR loop that is controlled by the '#' value.

```
        MACRO   BACK              ; Beginning of macro definition

PARMCNT SET     1                 ; Initialize the parameter counter

        REPEAT  PARMCNT<='#'      ; Repeat the following group of
                                  ; assembler statements while the
                                  ; current parameter count is less
                                  ; than or equal to the total number
                                  ; of parameters

        BYTE    HI('PARMCNT')     ; Store the high byte
        BYTE    LO('PARMCNT')     ; followed by the low byte of the
                                  ; selected parameter

PARMCNT SET     PARMCNT+1         ; Advance to the next parameter

        ENDR                      ; and repeat as necessary

        ENDM                      ; End of BACK definition
```

Macro BACK takes each of its parameters (one at a time), and generates two bytes for that parameter: the most significant byte of the parameter, followed by the the least significant byte. For example, the assembler statement:

```
BACK            25, 26, 27, LAB
```

generates the following assembler statements during the expansion of BACK:

```
BYTE    HI(25)
BYTE    LO(25)
BYTE    HI(26)
BYTE    LO(26)
BYTE    HI(27)
BYTE    LO(27)
BYTE    HI(LAB)
BYTE    LO(LAB)
```

The assembler performs the following operations during this expansion of BACK:

- BACK is invoked with the indicated parameters.
- The assembler variable PARMCNT is initialized to 1. PARMCNT always contains the number of the parameter currently being processed.
- The REPEAT loop is entered. The expression PARMCNT<='#' is expanded to PARMCNT<=4, since the total parameter number is 4. This expression is true (1 is less than 4), so the body of the REPEAT loop is evaluated.
- The assembler directive BYTE HI('PARMCNT') is expanded to BYTE HI(25). PARMCNT contains 1, so the first parameter is substituted for 'PARMCNT'. This assembler directive is then processed.
- In a similar manner, BYTE LO('PARMCNT') is expanded and processed.
- To select the second parameter, PARMCNT is incremented by one.
- The REPEAT loop is processed again, with PARMCNT equal to 2 (selecting the second parameter of 26). At the end of the loop, PARMCNT is incremented once again to 3.
- The REPEAT loop is processed once again, generating bytes for the third parameter (27). PARMCNT is again incremented.
- The REPEAT loop is processed once more, generating bytes for the fourth (and last) parameter, LAB. PARMCNT is incremented, and now contains the value 5.
- The expression of the REPEAT loop, PARMCNT<='#', is no longer true, since PARMCNT (5) is now greater than 4 (the total parameter count). The statements within the REPEAT loop are skipped, and processing continues after the ENDR statement.
- Expansion of BACK terminates, because no more statements remain to be processed.

Section 7
THE LINKER

Section 7

THE LINKER

INTRODUCTION

NOTE

The information in this section supports DOS/50 Version 1 ***and*** *DOS/50 Version 2.*

The linker merges one or more independently-assembled object files into a load file, suitable for loading into memory. Linker input may come from the assembler, or from library files. (See the Library Generator section of this manual for further information on library files.)

This section describes the operations and use of the linker, and is divided into the following subsections:

- **Linker Invocation.** Describes how you invoke the linker, using the operating system LINK command.
- **Linker Execution.** Describes operations performed by the linker.
- **Linker Output.** Describes the listing file generated by the linker.
- **Linker Commands.** Presents a detailed description of each command used to control the operation of the linker.

Some typical uses of the linker are presented in the Operating Procedures and Programming Examples sections of this manual.

LINKER INVOCATION

NOTE

The linker ***must*** *know what the target processor is in order to operate properly. Thus, you* ***must*** *use the DOS/50 command SEL prior to invoking the linker.*

You may invoke the linker by one of the following three methods:

- **Simple Invocation:** Requires you to specify only the input and output filespecs. Other linker parameters are set to default values. This method is adequate for most linking situations.
- **Interactive Invocation:** Allows you to control the linker more precisely using a series of commands. These commands define global symbols, listing content, and linker parameters, and specify section attributes and location. The commands used in interactive invocation are given later in this section.
- **Command File Invocation:** Allows you to place commands normally given in interactive invocation into a file. You can then direct the linker to process those commands when you specify only the filespec of that file.

Command file invocation is helpful whenever a particular sequence of linker commands must be used more than once. The sequence of commands can be entered once to a file, then processed many times by the linker. If you invoke the linker from an operating system command file, and simple invocation of the linker is not sufficient, then linker command file invocation can be used. In this case, interactive invocation requires you to be present during the linker's execution; this is generally not true in normal use of the operating system command files.

The method of invocation that you choose will depend on the linking situation. Each type of invocation is described in detail in the following pages.

Simple Invocation

SYNTAX

```
                            ⎧LIB(library)⎫
LINK     [load]     [list]  ⎩object      ⎭  ...
```

load	The filespec of the linker-created load file.
list	The filespec of the file or device that receives the linker listing file.
object	The filespec of an object file to be linked.
library	The filespec of a library file to be linked.

EXPLANATION

In simple linker invocation, you specify all input and output files in a single command line. The one or more object and library files are linked together to produce the load file. The linker's listing can be directed to a device or file, also specified in the command line.

Filespecs may not exceed 64 characters in length. If the complete filespec is longer than 64 characters, you may use a brief name for a portion of the filespec.

The **load** filespec may not begin with the "@" character, which would cause the filespec to be interpreted as a command file invocation. To prevent misinterpretation, precede the filespec with /USR/.

Similarly, an **object** filespec may not begin with the characters "LIB(". This would cause the filespec to be interpreted as a library. To prevent misinterpretation, precede the filespec with /USR/.

EXAMPLES

```
LINK LOAD LNKL OBJ
```

This invocation line links the object file OBJ to produce the load file LOAD and the listing file LNKL. All the files reside in the current directory.

```
LINK,,LPT MY1.OBJ MY2.OBJ
```

This invocation line links MY1.OBJ and MY2.OBJ (both in the current directory) generating a listing on the line printer (LPT). No load file is generated.

```
LINK LOAD LNKL MY1.OBJ MY2.OBJ LIB(/SYS/MY.LIB)
```

This invocation line links object files MY1.OBJ and MY2.OBJ (both in the current directory) to produce listing file LNKL and load file LOAD. If any unbound (undefined) globals remain after the two object files are linked, the linker will search through library file MY.LIB (in the system directory) for definitions of these unbound globals.

Interactive Invocation

SYNTAX

LINK

EXPLANATION

When you enter the LINK command without any parameters, the linker is invoked in interactive mode. The linker displays a prompt character (an asterisk), and waits for you to enter a series of linker commands. When you enter the linker END command, the linker processes the files you have specified in a linker LINK, LIST, or LOAD command line.

Linker commands are fully described in the Linker Commands subsection (later in this section).

NOTE

The operating system LINK command (described here) invokes the linker. The linker also has a command called LINK, which specifies a series of input files to the linker; that command is described in the Linker Commands subsection of this section. These two commands have distinctly different functions, and should not be confused.

Command File Invocation

SYNTAX

LINK @command-filespec

PARAMETERS

command-filespec — The filespec of a file or device (CONI, PPRT, etc.) from which the linker will read a series of commands.

EXPLANATION

This type of linker invocation is similar to interactive invocation, but commands are read from the designated file or device, rather than from the system terminal. Commands are taken from the file (or device) until the END command is read, or the end of the file is reached (whichever comes first).

Filespecs may not exceed 64 characters in length. If the complete filespec is longer than 64 characters, you may use a brief name for a portion of the filespec.

EXAMPLES

```
LINK @LNKC
```

This invocation line executes the linker commands contained within file LNKC in the current directory.

LINKER EXECUTION

A program consists of one or more object modules. Each object module contains one or more sections. Each section is an independent entity: a contiguous block of instructions and data that will eventually be located somewhere in memory. The linker derives the final position of each section in accordance with the attributes of the section. These section attributes, provided in the object module by the assembler, are described in the following paragraphs.

NOTE

Throughout this discussion, "section" (in lowercase) refers to an assembler-generated SECTION, COMMON, or RESERVE program/data block. "SECTION" (all uppercase) refers only to a program/data block generated with the SECTION assembler directive.

Section Attributes

Every section has five attributes that provide the linker with the necessary memory allocation information. These section attributes are name, section type, size, relocation type, and memory location.

Name: Each section has a name of up to eight characters. The name is assigned to the section with a SECTION, COMMON, or RESERVE assembler directive. The section name can be used as a global symbol to reference the first memory address of a section.

Section type: Each section is of type SECTION, COMMON, or RESERVE, as defined by the corresponding assembler directive.

Each SECTION must have a unique name. Multiple SECTIONs having the same name are flagged as errors.

All COMMON sections having the same name are allocated the same space and beginning address in memory. The length of this memory space is the size of the largest COMMON section of this name.

All RESERVE sections having the same name are concatenated by the linker. The length of a given RESERVE section in the program is the sum of all RESERVE sections having that same name.

Size: The size of each section, determined at assembly time, is the total number of memory bytes that the instructions or data of the section must occupy.

Relocation type: Each section has one of four relocation types: byte-relocatable, inpage-relocatable, page-relocatable, or absolute (non-relocatable).

Byte-relocatable sections may be placed anywhere within the microprocessor address space.

Inpage-relocatable sections are placed entirely within a microprocessor page. The length of the page is microprocessor-specific. Page length for each microprocessor is given in the corresponding Assembler Specifics section elsewhere in this manual. If an inpage-relocatable section exceeds one page in length, the linker displays an error, redefines the relocation type of the section to be page-relocatable, and continues the linking process.

Page-relocatable sections begin on a page boundary (an integral multiple of the page length).

Absolute sections are not relocated by the linker. Their position in memory is determined at assembly time through the use of the ORG directive. If two absolute sections are both designated by the assembler for the same memory area, the linker notes this conflict on the memory map, and the contents of this memory area are undefined.

Memory location: The memory location of all absolute sections is defined at assembly time. For relocatable sections, a beginning address or range of addresses may be specified with the LOCATE command at link time. The default address range for a relocatable section is the entire microprocessor addressing space.

Allocation of Sections

The linker computes an address range for each section to exclusively occupy in the linked program. Sections with more restrictive relocation types are given the first opportunity to obtain their required addresses. For example, an absolute section is allocated its (very) restrictive address range before any relocatable section is linked. The precise order of linking is as follows:

1. Absolute sections.
2. Based sections: any sections defined with the BASE attribute in the linker LOCATE command.
3. Ranged page-relocatable sections: any page-relocatable sections further restricted with a RANGE, as specified with the linker LOCATE command.
4. Ranged inpage-relocatable sections.
5. Ranged byte-relocatable sections.
6. Page-relocatable sections.
7. Inpage-relocatable sections.
8. Byte-relocatable sections.

When a memory location for a section is being chosen by the linker, the lowest memory address range that meets the relocation requirements (as well as addition restrictions presented in the LOCATE command) will be allocated to the section. For example, if the program consists only of 10 byte-relocatable sections, all 10 sections will be located in a contiguous block of memory starting at 0000.

Absolute and based sections are linked even if a conflict occurs (that is, when two or more sections have bytes at the same address). Any conflicts are noted on the linker memory map. Other section types are not linked if a conflict occurs. In any memory area where conflict has occurred, the contents are undefined.

Normally, the instructions and data for a section define a contiguous block of bytes in memory. However, some absolute sections of the program can be discontinuous as a result of the assembler ORG directive. Such sections define instructions and data for non-consecutive bytes of memory. The linker recognizes the gaps between the instructions/data of the section, and places other (relocatable) sections in these gaps. For example, if the assembler statement " ORG $+128" is present in an absolute section at assembly time, a gap of 128 (decimal) bytes is created within that section. The linker can then place any combination of relocatable sections into this gap, as long as the total number of bytes taken does not exceed 128.

ENDREL

ENDREL is a predefined global symbol. At link time, ENDREL is assigned the memory address that is one higher than the highest memory address assigned to any relocatable section (not absolute or based). Be aware that some absolute or based sections may be allocated memory that is higher than the address given by ENDREL.

If you do not reference ENDREL, no value is assigned. If you define a value for ENDREL, your value will take precedence over the predefined value.

Linking a Library File

If any undefined global symbols remain in the linker's global symbol table, and a library file has been specified, the linker examines the library file to determine of some or all of the undefined globals are defined in one of the library modules within that file.

Each module in the library contains a list of all global symbols defined within that module. Global symbols include section names, addresses within sections, and scalar values declared global at assembly time.

When a definition is found within a library module for an undefined global symbol, then that entire module is linked along with all other object modules. Only the modules that provide needed definitions for global symbols are linked; the rest are simply skipped.

The linker processes files in the order that the files are specified. If an object file requires that a library module be linked, you must specify the object file first. If a library file were to be specified first in a linker invocation, none of the library modules would be linked; when the linker processes the library, the global symbol table contains no undefined entries, causing the linker to skip the library. In general, the safest way to specify files in the command line is to list all object files **before** all library files.

Further information about libraries can be found in the Library Generator section of this manual.

LINKER OUTPUT

The linker generates two files. The **load file** contains the executable program instructions and data. The load file can be loaded into program memory with the operating system LOAD command. The **listing file** contains a summary of the actions performed by the linker. Either of these files can be omitted in any linker invocation. The listing file is described in the following paragraphs.

Listing File

The listing file summarizes the operations performed during the linking process. The listing file can be directed to any output device or file. The following information is included in the linker listing file:

	Simple Invocation	Interactive Invocation
Global Symbol List	Yes	Yes
Internal Symbol List	If selected in assembler	If selected in assembler
Map	Yes	If selected
Linker Statistics	Yes	Yes
Error Messages	If necessary	If necessary

Each of these listing items is described in the following paragraphs.

NOTE

Throughout this subsection, annotations are added to the listing samples to aid your understanding. These annotations are enclosed in square brackets ([]), and are not generated by the linker.

Global Symbol List

The global symbol list contains an alphabetic list of all global symbols and their values. These global symbols include those symbols defined with the assembler GLOBAL directive, as well as the names of SECTION, COMMON, and RESERVE sections. If a global symbol is undefined, its value field contains asterisks. In the following example, the global symbol QQQ is undefined, but was referenced by one or more object files.

```
TEKTRONIX  8080/8085 LINKER V x.x       GLOBAL SYMBOL LIST    PAGE      x

    Q1          1000  Q2          0500  QQQ          ****  X1SECT     0060
    X2_SECTA    045F  X2 SECTB    0640  [QQQ is undefined]
```

Internal Symbol List

The internal symbol list contains all symbols (other than strings and macros) defined in the assembler source file, along with their actual values after relocation. The internal symbol list parallels the assembler symbol table listing for the selected file. The list consists of three parts:

1. Alphabetical list of scalars used in the assembly.
2. Alphabetical list of labels occurring within each section.
3. Alphabetical list of labels derived from each unbound global symbol.

If there are no labels for a section, or no labels derived from an unbound global, then that section or unbound global is not indicated.

A sample internal symbol list follows.

```
TEKTRONIX  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST  PAGE      x

FILE:   QQ.OBJ  [input filespec]

MODULE:   IO_DRVR  [name assigned with the assembler NAME directive]

        SCALARS: [non-address symbols]
          A          0007     B           0000      C          0001     D          0002
          E          0003     H           0004      L          0005     M          0006
          PSW        0006     Q2          0500      R1         1E00     SP         0006
          X1VALUE    007F     X2VALUE     0030

        LABELS:   (SECTION  IO_AREA )  [all address symbols within section IO_AREA]
          L1         0100     L2          0130

        LABELS:   (SECTION  IO_AREA2)  [all address symbols within section IO_AREA 2]
          Q1         0150     Q2          0155

        LABELS:   (GLOBAL   IO_PORT )  [all symbols derived from global IO_PORT]
          IO PORT1   0070     IO PORT2    0071
```

The internal symbol list is displayed only for those object modules that were generated with the LIST DBG assembler option. Refer to the Assembler Directives section of this manual for further information about the LIST directive.

Map

The map consists of two parts: a module map, and a memory map. The map is included in the listing file only if the linker MAP command has been specified.

A **module map** lists all modules linked into the load file. The module map contains information about sections and global symbols defined in each object module.

```
TEKTRONIX  8080/8085 LINKER V x.x        MODULE MAP              PAGE     x

FILE:   FILE1.OBJ  [input filespec, as specified in LINK command]

MODULE:   MAINMOD  [module name, from assembler NAME directive]
   DO IO      SECTION BYTE 3700-3E40  [a byte-relocatable SECTION]
   INPUT      3A00  OUTPUT    3B50    [globals defined within this section]
   MAINPROG   SECTION BYTE 3E41-5141  [another byte-relocatable SECTION]
   ENTRY1     4091  ENTRY2    43A1    [globals defined within this section]
   STACK      RESERVE PAGE 3600-36FF  [a page-relocatable RESERVE section]

FILE:   FILE2.OBJ  [end of first file, beginning of second file]

MODULE:   SUBMOD  [module name, from assembler NAME directive]
   ABSECT2    SECTION ABSOLUTE 0040-0357  [absolute SECTION]
   ENTRY3     0090                        [global address within section]
   RELSECT2   SECTION PAGE 0400-2400      [page-relocatable SECTION]
   ENTRY4     0450                        [global address within section]

FILE:   FILE3.OBJ  [end of second file, beginning of third file]

MODULE:   SUBS2MOD  [module name]
   RELSECT3   SECTION PAGE 2500-3500  [a page-relocatable SECTION]
```

The module map lists all linked modules. An alphabetical list of sections and entry points (globals defined within each section) is included for each module. If no sections were linked in a module, no room for a section exists, or a section is empty, an appropriate message is included in the module map.

A **memory map** is an ordered listing of the memory allocated to sections. The list starts with the lowest allocated address and continues to the highest allocated address. For every address range, each section name and its attributes are given. An example of a typical memory map follows:

```
TEKTRONIX  8080/8085 LINKER V x.x        MEMORY MAP              PAGE     x

[beginning-ending address]
             [section name]
                       [section type]
                               [relocation type]

0040-0357  ABSECT2    SECTION ABSOLUTE
0400-2400  RELSECT2   SECTION PAGE
2500-3500  RELSECT3   SECTION PAGE
3600-36FF  STACK      RESERVE PAGE
3700-3E40  DO_IO      SECTION BYTE
3E4A-5141  MAINPROG   SECTION BYTE
```

Any address conflict (two or more sections assigned to the same address) is noted by an asterisk (*) following the address range in which the conflict occurs.

Linker Statistics

The linker statistics give the number of errors, undefined symbols, modules, sections, and the transfer address.

```
NO ERRORS      NO UNDEFINED SYMBOLS
3  MODULES     6  SECTIONS
TRANSFER ADDRESS IS 3E4A
```

The transfer address identifies the program starting location. After loading this example program, you could start execution by entering the operating system command "G 3E4A".

Error Messages

Error messages are issued wherever necessary. Three types of error messages can appear:

1. **Warnings (W)**: A problem exists but the linked program can probably be executed.
2. **Errors (E)**: A linked program probably will not execute properly.
3. **Fatal Errors (F)**: Any error directly affecting the linker's ability to continue; the linker terminates execution, and control returns to the operating system.

All errors cause a message to be displayed in the linker listing file and on the system terminal.

In the following list, each error message is indicated as being a warning (W), error (E), or fatal error (F).

ATTEMPT TO RE-DEFINE FILE TYPE FOR filespec. (W) **filespec** was specified twice: once as an object file, and once as a library file. The linker uses the first file type specified.

idname I/O ERROR #nn. (E) The linker was unable to read from or write to **idname**. (either LIST FILE, LOAD FILE, CONSOLE, COMMAND FILE, or OBJECT FILE). The error number is the corresponding operating system SVC (service call) SRB status byte. Refer to the Error Messages section of the 8550 System Users Manual for a description of the error.

IMPLICIT REORIGIN TO 0 IN SECTION sec IN MODULE mod FILE file. (W) A section has wrapped around from the last memory location to location 0.

INVALID OBJECT FORMAT AT LOCATION = nnnn.
IN FILE filespec. (E) The information contained within the file is not an object module. Verify that the designated object file has been generated by the assembler, or that LIB( ) surrounds the library filespec. **nnnn** indicates the internal linker address where the object file error was detected.

LINKER INTERNAL ERROR AT nnnn. (E) An error occurred in the linker; try linking again. If this error persists, carefully document the incident and submit an LDP Software Performance Report to Tektronix.

MACHINE REDEFINED FROM processor IN MODULE mod FILE file. (W) The current object module has been generated for a different microprocessor than the previous object modules. Incompatibilities during linking may result from differences between microprocessors, such as page length, byte order, etc.

MEMORY FULL. (E) The linker requires more memory to complete its task. The total number of globals, sections, or object modules must be reduced in order to link in the available memory.

NO ROOM IN RANGE mmmm-nnnn FOR SECTION sectname. (E) The length of the indicated section is greater than available contiguous memory in range **mmmm-nnnn** of allocated section memory.

RELOCATION TYPE OF SECTION sec MULTIPLY DEFINED IN MODULE mod FILE fln. (W) An attempt was made to redefine the section relocation type (byte, page, inpage, or absolute). This occurs when you use the LOCATE command to define a relocation type different from the type specified at assembly time. The error also occurs when relocation attributes of a COMMON or RESERVE section differ between modules. The linker uses the first-encountered relocation attribute to define the section.

SECTION sectname CHANGED FROM INPAGE TO type RELOCATABLE. (W) Section length is greater than the page size of the microprocessor. This can occur if several inpage RESERVE sections are linked together and their total size exceeds the page size of the microprocessor. A section declared to be inpage-relocatable in a LOCATE linker command generates this error if the section exceeds microprocessor page size. **type** is replaced with PAGE for sections smaller than available page size, or BYTE for sections larger than the microprocessor page size.

SECTION sectname CHANGED FROM PAGE RELOCATABLE. (W) Either the section was declared to be page-relocatable and the linker does not support paging for the microprocessor; or there was insufficient room for a paged section in available memory. In either case, the relocation type is changed to byte-relocatable.

SECTION sectname EXCEEDS MAXIMUM SIZE. (E) Section length is greater than the address space of the microprocessor. The section is not included in the load file. This error may occur when a concatenated RESERVE section is too long.

symbolname MULTIPLY DEFINED IN MODULE modname FILE filespec. (E) An attempt was made to redefine a global symbol or section. This error occurs when two modules define a global or section of the same name. All section names must be unique. The linker uses only the first definition of a section or global symbol in the load file.

TRANSFER ADDRESS MULTIPLY DEFINED IN MODULE mod FILE filespec. (W) The module has attempted to define the transfer address when an address has already been provided (either by another module or by the linker TRANSFER command). The linker uses the first-encountered transfer address to generate a transfer address for the load file.

TRANSFER ADDRESS UNDEFINED. (W) The transfer address has not been provided for this program. The transfer address can be provided either by a linker TRANSFER command, or as the optional expression value in an assembler END directive. When no transfer address is specified, the linker substitutes a transfer address of 0000.

TRUNCATION ERROR AT nnnn IN MODULE mod FILE filespec. (W) The relocated value computed for a byte is too large to fit in one byte.

UNABLE TO ASSIGN name. (E) The file or device **name** specified as an object or library file does not exist, or the output device is unavailable.

UNRESOLVED REFERENCE AT nnnn MODULE modname FILE filespec. (E) A reference to an unbound (undefined) global or section is specified at address **nnnn** in the object module. This error occurs when a global symbol is used in a module but not defined. The referenced symbol appears in the linker global symbol list with an undefined value (indicated by asterisks), and the unresolved reference is filled with zeros in the load file.

LINKER COMMANDS

Linker commands are used when you invoke the linker interactively or with a command file. Each linker command must be on a separate line.

In the following command descriptions, the same conventions are used as described in the Assembler Introduction section of this manual.

NOTE

All commands must be entered in their given form. Commands may not be abbreviated.

All filespecs in linker commands are limited to 64 characters in length. If the complete filespec is longer than 64 characters, you may use a brief name for a portion of the filespec.

DOS/50 allows any printing character (except the space) to appear in filespecs. The linker, however, allows only the following characters:

- The first character must be an uppercase letter or the slash character (/).
- Each remaining character in the filespec must be a printing character between ! (ASCII 21H) and _ (ASCII 5FH); however, plus, comma, and minus (ASCII 2BH, 2CH,and 2DH) are not permitted.

In particular, lowercase letters may not appear in filespecs. Before invoking the linker, you may use the appropriate DOS/50 commands to alter any filespecs, as necessary.

LINKER COMMAND DICTIONARY

SYNTAX

@filespec

PARAMETERS

filespec	The filespec of the command file containing a sequence of linker commands.

EXPLANATION

This command invokes **filespec** as a command file. The command file contains a series of linker commands. Commands are read from the file and processed as if you had entered them from the system terminal, until the END command is read or the end of file is reached. Commands are echoed on the system terminal as they are processed. When the end of the command file is reached, you will be prompted for additional linker commands. Nested command files are not allowed: a command file may not invoke another command file.

EXAMPLES

```
@ADD.LNKC
```

This linker command invokes command file ADD.LNKC. Additional linker commands will be read from file ADD.LNKC and processed, until the end of file is reached, or until an END command within ADD.LNKC is processed.

SYNTAX

DEBUG

EXPLANATION

The DEBUG command causes all symbols and their values to be stored in the load module. This makes your program symbols available for use in symbolic debug.

The DEBUG command may only be used in interactive invocation or command file invocation.

NOTE

*All assembly source files that have symbols to be referenced by symbolic debug must include the assembler directive **LIST DBG**. This directive causes the assembler to output the symbols to the object module, which makes the symbols available to the linker. (The size of the object module is increased appreciably.)*

*If you **relink** a load module that has been generated with the DEBUG command, the symbols will be relisted at link time.*

*For further information, see the topics, **DEBUG, LIST DBG,** and **symbolic debug** in the 8550 System Users Manual (DOS/50 Version 2).*

SYNTAX

DEFINE symbol=value[,symbol=value]...

PARAMETERS

symbol	A global symbol.
value	A hexadecimal constant.

EXPLANATION

The DEFINE command assigns values to selected global symbols. Each symbol is entered into the global symbol table and assigned the corresponding value. Even if the global symbol was previously defined (by an object module), the value you specify in a DEFINE command replaces the already-defined value.

EXAMPLES

```
DEFINE XXX=400, YYY=1FFF, IO_PORT=3E
```

This DEFINE command gives values to the global symbols XXX, YYY, and IO_PORT.

Linker Command: END

Signals end of linker command sequence

SYNTAX

```
END
```

EXPLANATION

The END command signals the end of the command sequence. You enter this command to start the linking process after you have completed entering all other linker commands.

This command must be used in interactive invocation, but can be omitted for command file invocation. If END is omitted in command file invocation, the linker begins the linking process when the end of the command file is reached.

Linker Command: LINK

Designates input file(s)

SYNTAX

$$\text{LINK} \quad \begin{Bmatrix}\textbf{LIB(library}\\ \textbf{object}\end{Bmatrix} \quad \begin{bmatrix}\textbf{,LIB(library)}\\ \textbf{,object}\end{bmatrix} \ldots$$

PARAMETERS

object	The filespec of an object file to be linked
library	The filespec of a library file to be linked

EXPLANATION

The LINK command designates the input object and library files that make up the program.

More than one LINK command can be specified in a sequence of linker commands. Successive LINK commands specify additional object and library files. For example, the command "LINK A, B, C" is identical in function to the command "LINK A" followed by commands "LINK B" and "LINK C".

An **object** filespec may not begin with the characters "LIB(". This would cause the filespec to be interpreted as a library. To prevent misinterpretation, precede the filespec with /USR/.

NOTE

> *The linker LINK command (described here) specifies a series of input files to the linker. The operating system also has a command called LINK, which invokes the linker; that operating system command is described earlier in this section under "Linker Invocation". These two commands (both called LINK) have distinctly different functions, and should not be confused.*

EXAMPLES

```
LINK MY.OBJ
```

This command selects object file MY.OBJ in the current directory to be linked.

```
LINK MY1.OBJ, MY2.OBJ
```

This command specifies object files MY1.OBJ and MY2.OBJ in the current directory to be linked.

```
LINK MY.OBJ, LIB(/SYS/MY.LIB)
```

This command specifies object file MY.OBJ in the current directory to be linked. If the object module within MY.OBJ contains any unbound global symbols, the linker searches through library file MY.LIB in the system directory for definitions of those symbols.

Linker Command: LIST

Designates listing file

SYNTAX

```
LIST     filespec
```

PARAMETERS

filespec	The filespec of the file or device of the linker listing.

EXPLANATION

The LIST command designates the file or device that is used for the linker listing. The contents of the listing file are described earlier in this section under "Linker Output."

EXAMPLES

```
LIST LPT
```

This LIST command designates the line printer (LPT) to receive the linker listing.

```
LIST MY.LNKL
```

This LIST command designates disc file MY.LNKL (in the current directory) to receive the linker listing.

SYNTAX

LOAD filespec

PARAMETERS

filespec	The filespec of the output load file.

EXPLANATION

The LOAD command designates the output file that receives the linked program. After linking, the file designated by the linker LOAD command may be brought into program memory, using the operating system LO command.

EXAMPLES

```
LOAD MY.LOAD
```

This LOAD command designates MY.LOAD in the current directory to receive the linked program.

NOTE

The linker LOAD command (described here) specifies the output file that contains the program after linking. The operating system has a command called LO, which transfers a file into program memory; that command is described in the 8550 System Users Manual. These two commands (LOAD and LO) have distinctly different functions, and should not be confused.

Linker Command: LOCATE

Changes section attributes

SYNTAX

```
                                                                     ┌,PAGE  ┐
                     ┌,BASE(starting-address)                      ┐ │,INPAGE│
LOCATE section-name  └,RANGE(starting-address,ending-address)      ┘ └,BYTE  ┘
```

PARAMETERS

section-name — The name of any section contained within the input object modules.

starting-address — A hexadecimal number representing a starting address.

ending-address — A hexadecimal number representing an ending address.

EXPLANATION

The LOCATE command alters the attributes of a SECTION, COMMON, or RESERVE section. The BASE parameter designates that the section should begin at the specified address. The RANGE parameter directs the linker to place the section anywhere within the given address range, as long as the beginning and ending addresses of the section lie within that range, and the location conforms to the relocation attribute (byte, inpage, page, or absolute).

The PAGE, INPAGE, or BYTE parameter redefines the relocation type of the designated section. When you redefine the relocation type of a section, the linked code may execute differently than you intended. Certain portions of the code may expect or require that a section of code be located in a particular memory location type (on a page boundary, within a page, etc.). Whenever you use the PAGE, INPAGE, or BYTE parameter, and the relocation type differs from the type given to the section at assembly time, the linker will generate a warning message.

EXAMPLES

```
LOCATE MYSEC.A, RANGE(2000,2FFF)
```

This command informs the linker that section MYSEC.A should be placed entirely within the range of 2000 to 2FFF (hexadecimal). If MYSEC.A is longer than 4096 bytes, or MYSEC.A cannot be located in the designated area, an error is generated.

```
LOCATE MYSEC.B, BASE(4000)
```

This linker command designates that MYSEC.B begins at memory location 4000.

```
LOCATE MYSEC.C, PAGE
```

This linker command redefines MYSEC.C to be page-boundary relocatable. The linker will attempt to place the first address of MYSEC.C at a page boundary. A warning message will be displayed if MYSEC.C was not defined to be page-relocatable at assembly time.

```
LOCATE MYSEC.D, RANGE(8000,FFFF), BYTE
```

This linker command designates that MYSEC.D will be placed somewhere in the upper 32K of memory, and redefines MYSEC.D to be byte-relocatable.

Linker Command: LOG

Enables command recording

SYNTAX

```
LOG
```

EXPLANATION

The LOG command causes all subsequent linker commands to be recorded (logged) in the linker listing file.

The NOLOG command restores the default setting: commands are not recorded in the linker listing file.

Linker Command: MAP

Includes map in listing

SYNTAX

MAP

EXPLANATION

The MAP command causes the map to be included in the linker listing file. Refer to the description of the map in the Linker Output subsection earlier in this section.

The NOMAP command restores the default setting: the map is not included in the linker listing file.

Linker Command: NOLOG

Disables command recording

SYNTAX

```
NOLOG
```

EXPLANATION

The NOLOG command disables the recording (logging) of linker commands in the linker listing file. Refer to the LOG command description for further information.

Linker Command: NOMAP

Does not include map in listing

SYNTAX

```
NOMAP
```

EXPLANATION

The NOMAP command restores the default map setting: the map is not included in the linker listing file. Refer to the description of the MAP command for further information.

Linker Command: TRANSFER

Specifies program start address

SYNTAX

TRANSFER $\begin{Bmatrix}\textbf{global-symbol}\\ \textbf{value}\end{Bmatrix}$

PARAMETERS

global-symbol A symbol appearing in the global symbol table.

value A one- to five- digit hexadecimal value that must begin with a digit (0 to 9).

EXPLANATION

The TRANSFER command defines the load file transfer address. The transfer address designates the address of the first instruction to be executed when the program is run. This address is displayed when the program is loaded into memory, and is used as the default starting address when the operating system G command is entered without an address parameter.

The transfer address can either be a fixed value (given as a hexadecimal address) or a global symbol. If a global symbol is designated, the transfer address will be taken from the symbol's value after linking.

The transfer address may have been given at assembly time by placing an expression after the END statement. If a transfer address is selected at assembly time, and the TRANSFER command is used at link time, the address specified in the TRANSFER command takes precedence.

EXAMPLES

```
TRANSFER 400
```

This linker command designates address 400 (hexadecimal) as the location of the first instruction to be executed.

```
TRANSFER MY.START
```

This command designates the value of the global symbol MY.START as the transfer address. When linking is completed, the value of MY.START is taken from the global symbol table and designated as the transfer address.

Command Processing Errors

If the linker detects an error during command entry, an up-arrow (∧) is displayed below the line, to indicate the approximate location of the error within the command line. A message defining the error is also displayed. These messages are described in the following paragraphs.

EXTRANEOUS INFORMATION IGNORED. Extra characters are on a command line that only requires an instruction (like LOG and MAP). The linker performs the appropriate action for the command, and ignores the extra characters.

ILLEGAL COMMAND. The command was not recognized.

INDIRECT FILE DEPTH EXCEEDED. A linker **@filespec** command was found during the processing of a command file. The command is ignored.

INVALID FILE NAME. The filespec specified in a LIST, LOAD, or LINK command contains illegal file characters. Refer to the Files section of the 8550 System Users Manual for information on valid filespecs.

INVALID RANGE SPECIFIED. The range in a LOCATE command is invalid. The ending address must be greater than the starting address.

SYNTAX ERROR. Statement syntax is invalid. This error occurs when a command does not precisely match the syntax for that command. For example, unmatched parentheses are found in the LOCATE command, or an operand is missing after the equals sign in a DEFINE command.

Section 8
THE LIBRARY GENERATOR

Page

Illustration

Fig.
No.

Section 8

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 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 you invoke LibGen, using the operating system LIBGEN command.
- **LibGen Execution.** Describes operations performed by LibGen.
- **LibGen Output.** Describes the listing file generated by LibGen.
- **LibGen Commands.** Presents a detailed description of each command used to control the operation of LibGen.

Some typical uses of LibGen are presented in the Operating Procedures and Programming Examples sections of this manual.

LIBGEN INVOCATION

You may invoke LibGen by either of the following methods:

- **Interactive Invocation.** Allows you to control LibGen using a series of commands. These commands direct LibGen to examine or alter the library by inserting, deleting, or replacing object modules, or copying object modules to other files.

  Interactive invocation is the most common method of invoking LibGen.
- **Command File Invocation.** Allows you to place commands normally given in interactive invocation into a file. You can then direct LibGen to process those commands when you specify only the filespec.

  Command file invocation is helpful whenever a particular sequence of LibGen commands must be used more then once. The sequence of commands can be entered once in a file, then processed many times by LibGen. If you invoke LibGen from an operating system command file, then LibGen command file invocation can be used. In this case, interactive invocation will not suffice, since it requires you to be present during LibGen's execution; this is generally not true in normal use of operating system command files.

These two methods of invocation are described on the following pages.

Interactive Invocation

SYNTAX

```
LIBGEN  [new-lib]  [list]  [old-lib]
```

PARAMETERS

new-lib	The filespec of the output library file.
list	The filespec of the LibGen listing file or device.
old-lib	The filespec of the input library file.

EXPLANATION

In interactive LibGen invocation, you designate the input and output library files, and the listing file. LibGen will display a prompt character (an asterisk), and wait for you to enter a series of LibGen commands. (These LibGen commands are described individually later in this section.) After you have entered the LibGen END command, LibGen processes the files you have specified.

LibGen can be used to create new library files, modify existing library files, or examine existing files. To create a new library file, omit the **old-lib** parameter. To modify an existing library file, include both the **old-lib** and **new-lib** parameters; any unmodified contents of the old library are copied to the new library. To examine an existing library file, omit the **new-lib** parameter.

Filespecs may not exceed 64 characters in length in the invocation line. Filespecs may not exceed 32 characters in length in interactive LibGen commands. If the complete filespec is longer, you may use a brief name for a portion of the filespec.

You may optionally specify the filespecs with the NEWLIB, OLDLIB, and LIST commands, rather than specifying them in the LIBGEN command line. Refer to the LibGen Commands subsection of this section for information on these commands.

EXAMPLES

```
LIBGEN MY.LIB LPT SYS.LIB
```

This invocation of LibGen designates MY.LIB (in the current directory) as the output library file, the lineprinter (LPT) as the device that will receive the listing, and SYS.LIB as the input library file. After invocation, LibGen prompts for a sequence of commands.

```
LIBGEN FP.LIB FP.LBGL
```

This LibGen invocation creates a new library file, FP.LIB. A listing file FP.LBGL is also created. Both files reside in the current directory. After this invocation, LibGen prompts for a series of commands.

```
LIBGEN,,LPT MY.LIB
```

When the name of the output library file is omitted, as in this invocation, no output library file is created. The output library file can be omitted when you only need a listing of the contents of a library file, or you want to extract one or more library modules to object files.

```
LIBGEN
```

In this invocation of LibGen, no input, output, or listing files are specified. LibGen commands (such as NEWLIB, OLDLIB, and LIST) must be used to specify the appropriate input and output files.

Command File Invocation

SYNTAX

LIBGEN @command-filespec

PARAMETERS

command-filespec	The filespec of the file or device from which LibGen will read a series of commands.

EXPLANATION

This invocation of LibGen is similar to interactive invocation. In this case, however, commands are read from the designated file or device, instead of from the system terminal. Commands are read from the specified file until the END command is read, or until the end of the file is reached (whichever comes first).

EXAMPLES

```
LIBGEN @LBGC
```

This invocation line executes the LibGen commands contained in file LBGC in the current directory.

LIBGEN EXECUTION

LibGen performs operations on library files by copying an old library file into a new one. Changes, as specified by LibGen commands, are made during the copying process. This process is illustrated in Fig. 8-1.

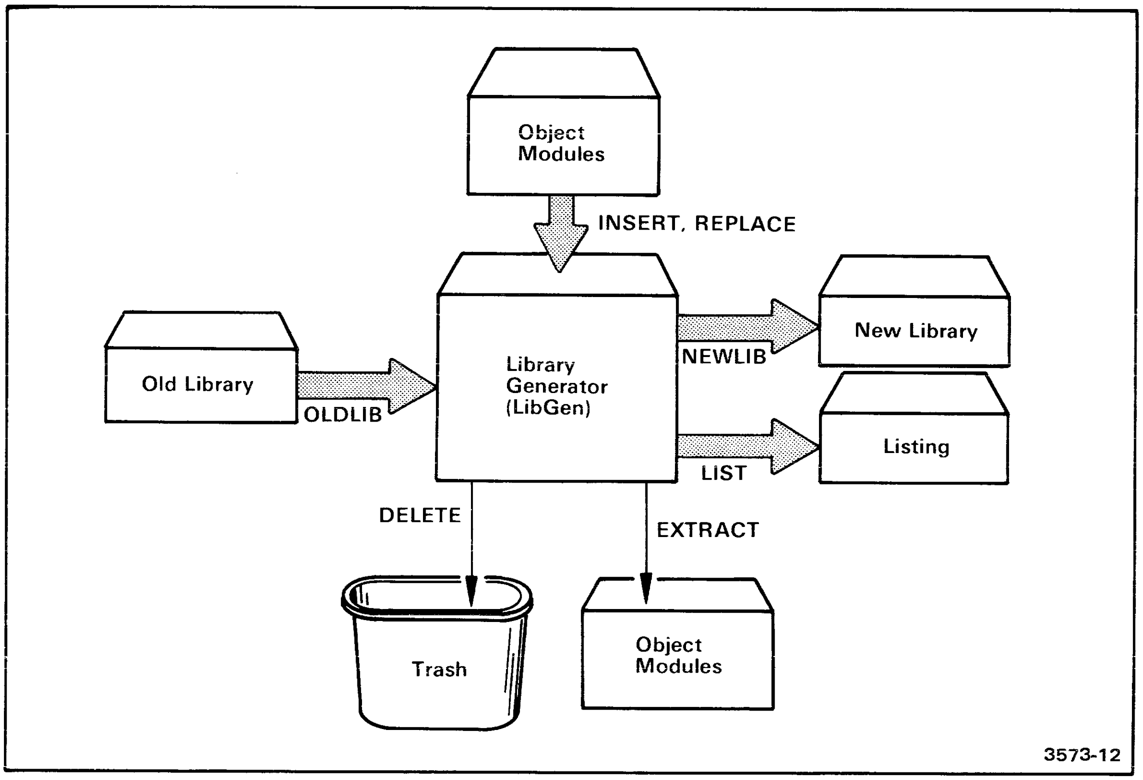


Fig. 8-1. LibGen information flow.

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 commands that designate the filespecs used for each file are given along each data path line. The END, LOG and NOLOG commands are not shown, since they do not control the direction of information flow in LibGen.

Any of the information paths in Fig. 8-1 can be omitted when they are not necessary. For example, if you are creating a new library, then no old library is needed. If you are examining an old library, then no new library need be created. If you do not need a listing, do not specify one.

Three of the filespecs may be specified in the LibGen invocation line: the old library file, the new library file, and the listing file. Other filespecs and operations may be specified only with the indicated LibGen commands.

LibGen does not process each command at the time you enter it, but saves all commands to be processed in a specific order. LibGen processes commands in this order:

1. INSERT BEFORE
2. EXTRACT
3. DELETE
4. INSERT AFTER

The REPLACE command is processed as a combination of the DELETE and INSERT AFTER commands.

LIBGEN OUTPUT

LibGen produces three different types of output files: the new library file, a listing file, and zero or more object files (if specified with the EXTRACT command).

The New Library File

The new library file is the primary product of the library generator. The new library 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 symbol 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. These error messages are described at the end of this subsection.

Command Log

The command log lists each LibGen command used in the current invocation. The command log is optional; you can enter the LOG command to include the log in the listing, or the NOLOG command to omit the log. When you specify neither of these commands, the command log is included by default.

Symbol List

In this part of the listing, LibGen records the names of all modules contained in the output library, and the global symbols contained within each module.

Global symbols within each module are divided into three categories:

- **Section names:** The name of a SECTION, COMMON, or RESERVE contained within the module.
- **Entry points:** An address (within the most-recently-listed section) declared global with the assembler GLOBAL directive.
- **Global symbols:** A scalar value declared global with the assembler GLOBAL directive.

These symbols are preceded in the listing with either a (S), (E), or (G), indicating section name, entry point, or global symbol, 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 global symbol named "P9". At link time, if any one of the symbols "P", "P1", or "P9" has been referenced (through an unbound GLOBAL reference), and 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 Action

The summary of action 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 a library module to an object file.

Error Messages

Error messages are issued wherever necessary. Two types of error messages can appear:

1. **Non-Fatal Errors (N):** LibGen cannot process the command as entered, due to syntax errors, or improper file/module specifications. Processing will continue, but the result may not be exactly what you had expected.
2. **Fatal Errors (F):** LibGen has encountered a major problem that prevents further processing. The error message is displayed, and control returns to the operating system.

All errors cause a message to be displayed on the system terminal. The error message will also appear in the LibGen listing file, if one is being generated.

In the following list, each error message is indicated as being a non-fatal error (N), or a fatal error (F).

CAN NOT FIND END BLOCK FOR MODULE IN FILE filespec. (F) **filespec** is not a valid object file. Verify that you have specified the correct filespecs in your INSERT and REPLACE commands.

CAN NOT FIND END BLOCK FOR MODULE modname OF LIBRARY oldlib. (F) **oldlib** is not a valid library.

comtype DATA STRUCTURE OVERFLOW. (F) Too many **comtype** (INSERT, DELETE, or EXTRACT) commands were specified in the current LibGen invocation. LibGen allows a maximum of 100 commands of any given type.

COULD NOT FIND MODULE modname IN oldlib, newmod INSERTED AT END OF newlib. (N) The BEFORE/AFTER parameter of an INSERT command specified a library module not present in the old library. The module will be added to the end of the library.

FILE filespec IS NOT AN OBJECT FILE. (F) **filespec** is not a valid object file. Verify that you have specified the correct filespecs in your INSERT and REPLACE commands.

filespec I/O ERROR #nn. (F) The operating system has reported an I/O error during the access of the specified file. The error number is the service call (SVC) status byte value in hexadecimal. Refer to the Error Messages section of the 8550 System Users Manual for a description of the error and for possible actions to take to correct the situation.

ILLEGAL COMMAND. (N) The command specified is not a valid LibGen command. Refer to the list of valid LibGen Commands later in this section. The command line is ignored.

INDIRECT FILE DEPTH EXCEEDED. (N) An **@filespec** command was read from a command file. Command files may not invoke other command files. The command is ignored.

INVALID FILE NAME. (N) A filespec contains an invalid character. The invalid character(s) are deleted, and processing continues.

INVALID OBJECT FORMAT FOR FILE filespec LOCATION = nnnn. (F) **filespec** is not a valid object file. Verify that you have specifed the correct filenames in your INSERT and REPLACE commands.

MODULE(S) NOT FOUND IN oldlib. (N) The modules specified in an EXTRACT or DELETE command were not found in the old library. The command is ignored.

NO OLD LIBRARY GIVEN, filespec INSERTED AT END OF newlib. (N) The BEFORE/AFTER parameter of an INSERT command specified a library module, but no old library was given. The module will be added to the end of the new library.

oldlib NOT A LIBRARY. (F) **oldlib** is not a library file. Verify that you have specified the proper filespec in the LibGen invocation line, or the parameter of an OLDLIB command.

SYNTAX ERROR. (N) The command does not conform to the proper syntax for that command. The command line is ignored.

UNABLE TO ASSIGN filespec. (N) **filespec** cannot be located. Verify that you have entered the proper filespec.

WARNING. DUPLICATE MODULE NAME: modname. (N) Two or more modules within the library file have the name **modname**. This condition does not affect the performance of the linker when selecting modules, but will make future modification and maintenence of the library difficult. When creating a library, be sure to give each object module a unique name with the assembler NAME directive.

LIBGEN COMMANDS

LibGen commands allow you to control the operations that LibGen will perform. When you invoke LibGen interactively, you must enter one (and only one) command each time LibGen prompts with an asterisk. When you invoke LibGen with a command file, each line of the command file should contain one LibGen command.

Whenever you enter a series of LibGen commands, the last command must be the END command. If you invoke LibGen with a command file, you may omit the END command.

In the following command descriptions, the same conventions are used as described in the Assembler Introduction section of this manual. Additionally, the following abbreviation convention is used.

Most commands can be entered either of two ways:

1. using the full name of the command (INSERT), or
2. using the designated abbreviation (I).

The designated abbreviation for each command is indicated by the underlined portion of the command in the syntax description. If all letters in the command are underlined, then no abbreviation is permitted. Partial abbreviations are **never** permitted.

All filespecs in interactive LibGen commands are limited to 32 characters in length. If the complete filespec is longer than 32 characters, you may use a brief name for a portion of the filespec.

DOS/50 allows any printing character (except the space) to appear in a filespec. The library generator, however, allows only the printing characters from ! (ASCII 21H) to_(ASCII 5FH), with the exception of plus, comma, and minus (ASCII 2BH, 2CH, and 2DH). In particular, lowercase letters may **not** appear in LibGen filespecs. Before invoking the LibGen, you may use the appropriate DOS/50 commands to alter any filespecs, as necessary.

LibGen Command: @filespec

Invokes a LibGen command file

SYNTAX

@filespec

PARAMETERS

filespec — The filespec of the command file containing a sequence of LibGen commands.

EXPLANATION

This command invokes **filespec** as a command file. The command file contains a series of LibGen commands. Commands are read from the file and processed as if you had entered them from the system terminal, until the END command is read or the end of file is reached. Commands are echoed on the system terminal as they are processed. When the end of the command file is reached, you will be prompted for additional LibGen commands. Nested command files are not allowed: a command file may not invoke another command file.

EXAMPLES

```
@ADD.LBGC
```

This LibGen command invokes command file ADD.LBGC (in the current directory). Additional LibGen commands will be read from file ADD.LBGC and processed, until the end of file is reached, or until an END command within ADD.LBGC is processed.

LibGen Command: DELETE

Deletes library modules

SYNTAX

```
DELETE module-name [,module-name] . . .
```

PARAMETERS

module-name The name of an input library module that you want to delete from the output library.

EXPLANATION

The DELETE command prevents the designated modules from being copied from the old library file into the new library file.

If two or more modules with the designated name exist, every module with that name is deleted.

EXAMPLES

```
DELETE MYMOD
```

This DELETE command removes MYMOD from the output library.

```
DELETE IO.OPS, FPOINT, RANDOM$$
```

This DELETE command removes modules IO.OPS, FPOINT, and RANDOM$$ from the output library.

SYNTAX

```
END
```

EXPLANATION

The END command signals the end of the command sequence. You enter this command to start the library generation process after you have completed entering all other LibGen commands.

This command must be used in interactive invocation, but it can be omitted for command file invocation. If END is omitted, LibGen begins the library generation process when the end of the command file is reached.

LibGen Command: EXTRACT

Copies module to object file

SYNTAX

EXTRACT module-name TO filespec

PARAMETERS

module-name The name of a library module to be copied to a file.

filespec The filespec of the file that is to receive copy of the library object module.

EXPLANATION

The EXTRACT command copies the designated library object module to a file. The designated object module remains in the library (unless it has also been designated in a DELETE command). If the specified file already exists, it is replaced by the designated library object module; the old contents are lost without warning.

EXAMPLES

```
EXTRACT FP$MULT TO FPMULT.OBJ
```

This EXTRACT command copies the library module FP$MULT to the file FPMULT.OBJ.

```
EXTRACT IO.MOD TO IO.OBJ
```

This EXTRACT command copies the library module IO.MOD to the file IO.OBJ in the current directory.

SYNTAX

```
                                  ┌BEFORE lib-module-name┐
INSERT filespec [,filespec] . . . └AFTER lib-module-name ┘
```

PARAMETERS

filespec — The filespec of an object file containing one of the object modules to be inserted.

lib-module-name — The name of an object module already present in the library.

EXPLANATION

The INSERT command adds new object modules into the library. Each specified object file contains one object module. These modules are placed into the new library file according to the BEFORE/AFTER parameter (or its absence). If more than one object file is specified, all designated object modules are placed together in the given order, with the entire group located according to the BEFORE/AFTER parameter.

The BEFORE/AFTER parameter controls the placement of the module(s) in the following manner:

- If the BEFORE/AFTER parameter is omitted, the object module(s) are placed at the beginning of the library.
- If the BEFORE parameter is given, the object module(s) are placed immediately **before** the designated library module (**lib-module-name**).
- If the AFTER parameter is given, the object module(s) are placed immediately **after** the designated library module (**lib-module-name**).

If the BEFORE/AFTER parameter is entered, but the designated library module cannot be found in the library, an error is generated, and the object module(s) are placed at the end of the library.

EXAMPLES

```
INSERT IO.OBJ
```

This INSERT command adds the contents of file IO.OBJ (located in the current directory) to the beginning of the library.

```
INSERT FPADD.OBJ, FPSUB.OBJ, FPMULT.OBJ
```

This INSERT command adds the contents of files FPADD.OBJ, FPSUB.OBJ and FPMULT.OBJ to the beginning of the library

```
INSERT FPDIV.OBJ BEFORE FP$MULT
```

This INSERT command adds the contents of file FPDIV.OBJ to the library file. The object module contained in FPDIV.OBJ is placed immediately before the library object module named FP$MULT.

```
INSERT FPCLR.OBJ, FPROT.OBJ, FPSIGN.OBJ AFTER FP$ADD
```

This INSERT command adds the contents of object files FPROT.OBJ, FPROT.OBJ, and FPSIGN.OBJ immediately after the library module FP$ADD.

SYNTAX

LIST filespec

PARAMETERS

filespec — The filespec of the LibGen listing file or device.

EXPLANATION

The LIST command specifies the file or device for the LibGen listing. Refer to the LibGen Output subsection of this section for information on the contents of the listing.

The listing file may also be specified by the second parameter of the LIBGEN command during interactive invocation.

EXAMPLES

```
LIST LPT
```

This LIST command designates the line printer (LPT) to receive the LibGen listing.

```
LIST MY.LBGL
```

The LIST command designates file MY.LBGL (located in the current directory) to receive the LibGen listing.

LibGen Command: LOG

Enables command recording

SYNTAX

LOG

EXPLANATION

The LOG command causes all subsequent LibGen commands to be recorded (logged) in the LibGen listing file. Each command, as entered, appears in a section of the LibGen listing file for future reference.

The NOLOG command disables the recording of LibGen commands in the LibGen listing file.

The default setting is logging enabled (identical to the effect of the LOG command).

SYNTAX

NEWLIB filespec

PARAMETERS

filespec	The filespec of the new library file.

EXPLANATION

The NEWLIB command designates the output file that is to receive the updated library. If the specified file currently exists, that file is replaced (without warning) with the new library file. If more than one NEWLIB command is entered in a command sequence, or NEWLIB commands are specified during an interactive invocation, only the file specified in the last NEWLIB command processed is used as the output library file.

When LibGen is invoked with a command file, the NEWLIB command is essential for specifying the output library file. However, when LibGen is invoked interactively, the output library file may be specified either as the first parameter of the operating system LIBGEN command, or as the parameter of a LibGen NEWLIB command.

EXAMPLES

```
NEWLIB FPPACK.LIB
```

This NEWLIB command designates FPPACK.LIB (in the current directory) as the output library file.

LibGen Command: NOLOG

Disables command recording

SYNTAX

NOLOG

EXPLANATION

The NOLOG command disables the recording (logging) of LibGen commands in the LibGen listing file. Refer to the LOG command for further information.

The default setting is logging enabled (identical to the effect of the LOG command).

LibGen Command: OLDLIB

Specifies input library

SYNTAX

OLDLIB filespec

PARAMETERS

filespec	The filespec of the old library file.

EXPLANATION

The OLDLIB command designates the input file that contains the source library. If more than one OLDLIB command is entered in a command sequence, or OLDLIB commands are specified during an interactive invocation, only the file specified in the last OLDLIB command processed is used as the input library file.

When LibGen is invoked with a command file, the OLDLIB command is essential for specifying the input library file. However, when LibGen is invoked interactively, the input library file may be specified either as the third parameter after the operating system LIBGEN command, or as the parameter of a LibGen OLDLIB command.

EXAMPLES

```
OLDLIB FPPACK.LIB
```

This OLDLIB command designates FPPACK.LIB (in the current directory) as the input library file.

LibGen Command: REPLACE

Replaces old module with new module

SYNTAX

REPLACE lib-module-name BY filespec

PARAMETERS

lib-module-name The name of an object module already present in the library.

filespec The filespec of a file containing an object module that will replace **lib-module-name**.

EXPLANATION

The REPLACE command replaces the designated library module with the contents of an object file. The old library module is deleted (as if the appropriate DELETE command were entered), and the object module contained within the object file is inserted in its place (as if the appropriate INSERT AFTER command were entered).

If more than one library module has the specified module name, then all modules with that name are deleted, and the new object module replaces the first library module with that name.

If the specified file does not exist, the library module is deleted and an error occurs.

EXAMPLES

```
REPLACE FP$ADD BY NEWADD.OBJ
```

This REPLACE command deletes module FP$ADD from the library and inserts the contents of object file NEWADD.OBJ in its place. in its place.

Section 9
PROGRAMMING EXAMPLES

Page

ILLUSTRATIONS

Fig.
No.

Section 9

PROGRAMMING EXAMPLES

INTRODUCTION

NOTE

This section supports DOS/50 Version 1 and DOS/50 Version 2.

This section contains examples of some typical uses of the assembler, linker, and library generator. These examples range from a simple macro invocation, to the creation and use of a complex floating-point library.

These examples assume that you have some familiarity with assembly language programming, and with the Tektronix Assembler, Linker, and Library Generator. You can use these examples as "application notes" for the assembler, linker, and library generator's features. These examples are **not** intended to be used during your initial familiarization with these subsystems.

The following examples are included in this section:

- **Using a simple assembler macro.** This example creates a small, general-purpose assembler macro, and shows some typical ways you can create, define, and invoke a macro.
- **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.
- **DOS/50 SVC generation.** This example shows how the macro and conditional assembly features of the assembler can make it easier to use SVCs (service calls) under DOS/50.
- **Creating constant values.** This example shows how to use an assembler macro to declare a constant value in a separate assembler section. You could use this technique to keep your instructions, fixed data values, and variable data values separate, so that you could eventually place your program into ROM.
- **Save-and-restore macro.** This example shows a typical application of an intelligent macro to perform a common programming operation: saving registers on the stack and later restoring the registers from the stack.
- **Conditional assembly.** This example suggests ways of using the IF assembler directive to include or omit various program segments, based on various conditions.
- **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.

USING A SIMPLE ASSEMBLER MACRO

This example illustrates the use of a small, general-purpose assembler macro. The macro generates multiple copies of an assembler statement.

First, the macro is defined. Then, the example shows alternative ways of defining the macro, using various assembler features. Finally, a few sample invocations of the macro are presented.

The macro itself is simple. The macro is invoked with two parameters: an integer and an assembler statement. The first parameter designates how many copies to generate. For example, if the macro is given the two parameters of 16 and " WORD 0", the macro will generate 16 lines of " WORD 0". Other invocations are given later in this example.

The COPY Macro

```
        MACRO   COPY                    ; line 1
COPY$   SET     1                       ; line 2
        REPEAT  COPY$<='1'              ; line 3
'2'                                     ; line 4
COPY$   SET     COPY$+1                 ; line 5
        ENDR                            ; line 6
        ENDM                            ; line 7
```

The macro is named COPY (in line 1), to remind you of its function: generate multiple copies of a designated assembler statement. Generally, you should give a macro a name that reflects its purpose.

Line 2 sets the assembler variable COPY$ to 1. This variable (COPY$) is used later in the body of the macro to keep track of the number of copies generated.

Line 3 begins a REPEAT loop. The REPEAT assembler directive causes all statements between this directive and the matching ENDR directive (line 6) to be repeatedly assembled. The assembler stops assembling these statements when the condition of the REPEAT loop (the first operand in the REPEAT directive) is false (zero).

For this macro, the condition expression is zero when the value of the assembler variable COPY$ is **not** less than or equal to (<=) the first parameter ('1') specified when COPY is invoked. In other words, the two statements within the REPEAT loop are repeatedly assembled as long as the assembler variable COPY$ is not greater than the first parameter.

Line 4 is a placeholder for the second parameter specified in the macro invocation line. When the assembler processes this statement, it replaces the '2' with the the assembler statement that is to be copied.

Line 5 increments the "number-of-copies" counter, COPY$. This counter is incremented once each time the statements within the REPEAT loop are assembled, to keep track of the number of copies generated.

Line 6 terminates the REPEAT loop. As long as the condition of the REPEAT loop is non-zero (true), the assembler will return to the REPEAT statement for another pass through the REPEAT loop. When the condition is zero (false), the assembler proceeds with assembly following the ENDR statement.

Line 7 terminates the definition of macro COPY$.

Defining the Macro

The macro can be defined in three different ways:

1. The macro can be placed at the beginning of the assembly source file that needs to use the macro.
2. The macro can be placed in a separate file, and brought into the source file with an INCLUDE assembler directive.
3. The macro can be placed in a separate file, and concatenated to the beginning of the source file when you specify the operating system ASM command.

These alternatives are described in the following paragraphs.

Defining the Macro As Part of Source File

If the macro is needed for only one assembler source file, this method of definition is easiest. Simply place the lines forming the macro definition somewhere in your source file before the first invocation of the macro. A typical place would be somewhere near the beginning of the file.

This method is illustrated in Fig. 9-1.

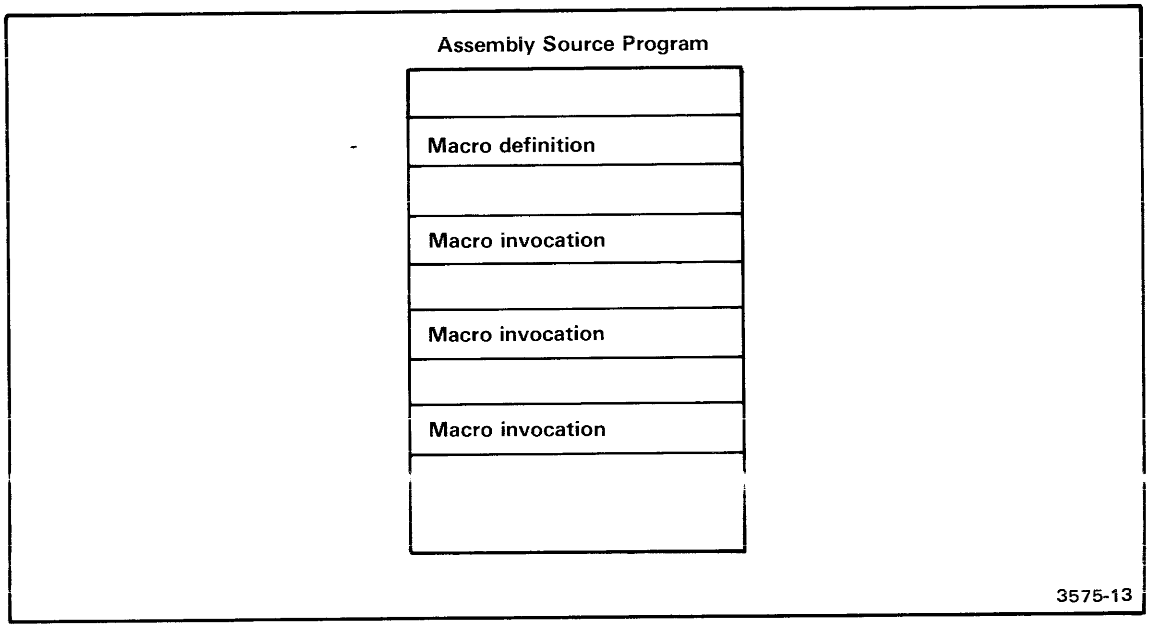


Fig. 9-1. Defining a macro as part of the source file.

In this method, the macro is defined once, near the beginning of the file. The macro may then be invoked as needed.

Defining the Macro Using the INCLUDE Directive

If the same macro is needed in several assembler source files, you can place the macro in a separate file, then refer to the filespec with an assembler INCLUDE directive.

For example, you can place the lines defining the macro into a file named CPYM.ASM. Then, you'd place the INCLUDE statement in your assembler source file (before your first invocation of the macro):

```
Label   Operation        Operand          Comment

        INCLUDE          "CPYM.ASM"       ; Obtain COPY macro definition
```

When the assembler processes this statement, it will examine the contents of file CPYM.ASM, which defines the macro COPY. This method is illustrated in Fig. 9-2.

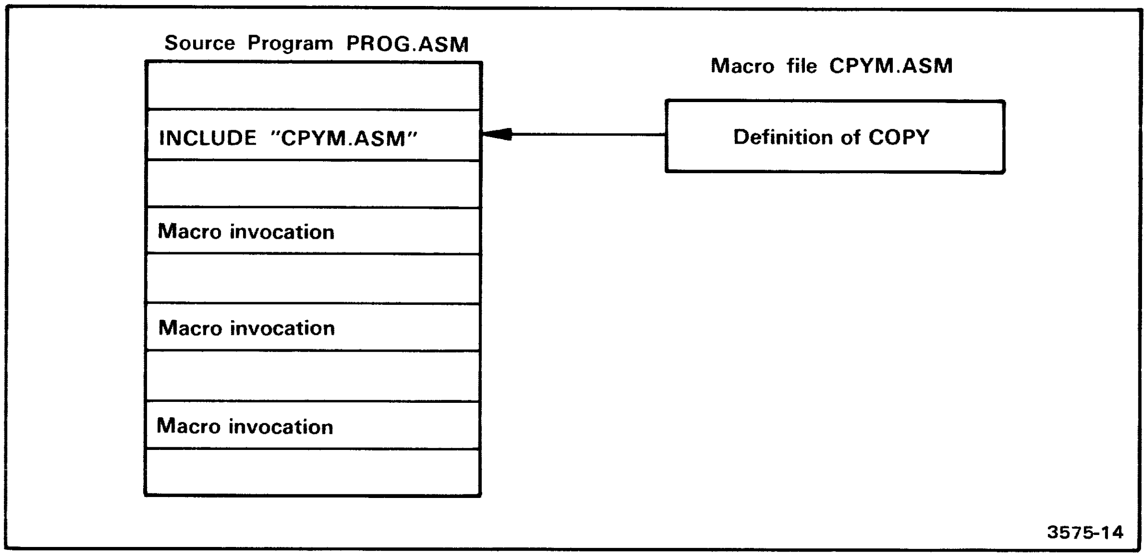


Fig. 9-2. Defining a macro with an INCLUDE directive.

In this method, the contents of file CPYM.ASM are brought into the assembler source file PROG.ASM, at the point indicated by the INCLUDE directive. This way, the macro is defined before its first invocation.

Defining the Macro in a Concatenated Prefix File

This definition method is much like the INCLUDE method. However, in this case, the filespec containing the macro definition is **not** specified by an assembler statement, but is specified at the time you enter the ASM command.

Let's assume again that the macro resides in a file named CPYM.ASM, and that your source program is named PROG.ASM. To assemble your source program, you enter the following operating system command line:

```
> ASM PROG.OBJ PROG.LOAD CPYM.ASM PROG.ASM
```

Prefix file

Notice what happens: the file CPYM.ASM is effectively "glued" to the front of the source program PROG.ASM by this operating system ASM command line. The assembler will read and process the contents of CPYM.ASM before processing the statements of PROG.ASM, thus ensuring that the COPY macro will be defined before its first use. This process is illustrated in Fig. 9-3.

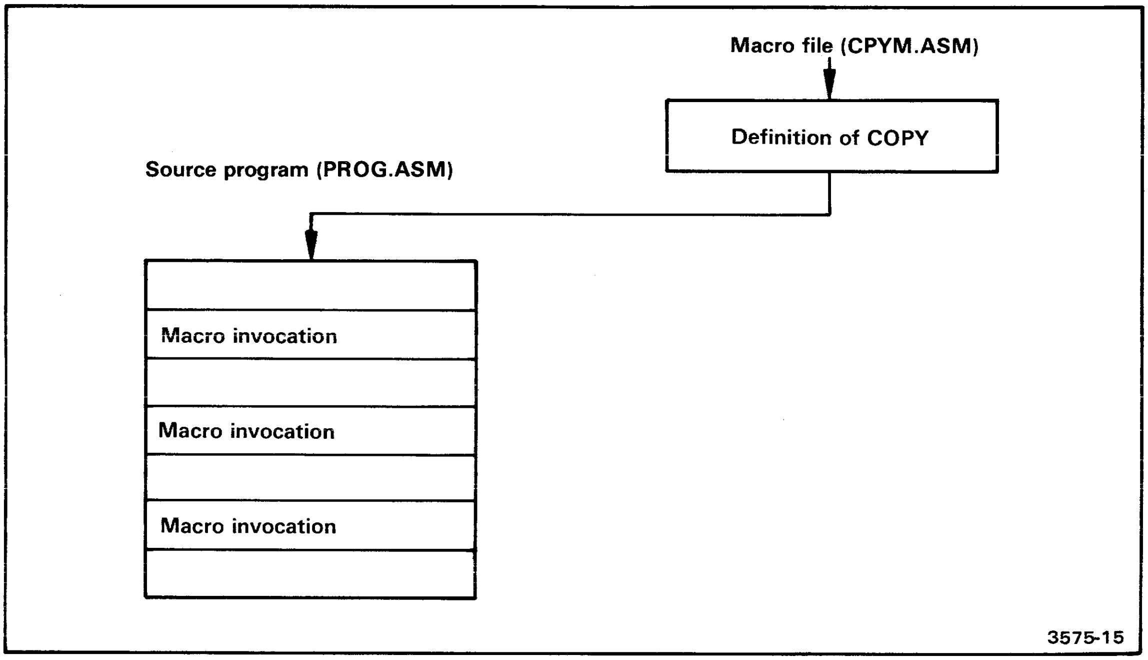


Fig. 9-3. Defining a macro in a concatenated prefix file.

In this method, the macro definition file (CPYM.ASM) is attached by the assembler to the beginning of the main program (PROG.ASM).

This method has two major advantages:

1. You specify the name of the macro definition file when you assemble the file, instead of when you edit the file. Sometimes you may not know the complete filespec of the definition file when you're entering the program. This method allows you to change the name or volume without editing the file.
2. Your macros are guaranteed to have been defined before their first use; the assembler processes the prefix file before it assembles any statements in the main part of the program.

This method has two disadvantages that you should be aware of:

1. The name of the macro definition file **must** be specified each time you assemble the file. This disadvantage can be minimized if you create an operating system command file containing the assembler invocation line.
2. The line numbers in the assembler listing are incremented for any assembled line; therefore, the line number of an error in the listing will not necessarily compare correctly with the line number of the main program source statements. In our example, an error appearing on line 50 of the assembler listing would actually refer to line 43 (in our example) of PROG.ASM, because the first seven lines of the listing have been obtained from the file CPYM.ASM.

Sample Invocations of the COPY Macro

Now that COPY has been defined (by one of the three methods mentioned above), you may use the macro in your program. For example, suppose that you need 20 (decimal) consecutive constant-value bytes—each byte containing the value 47 (decimal). Without the aid of this macro, you would need to enter the BYTE directive with 20 operands (each being the value 47), or 20 BYTE directives, each with an operand of 47, or some combination of the above entries. With the aid of the macro, however, you only need to write one assembler statement:

```
COPY    20,[ BYTE 47]
```

Notice that the second parameter is enclosed in matching square brackets. These brackets are not part of the parameter, but indicate the part of the statement line that belongs to the parameter. Without the brackets, the essential leading space (before the word BYTE) would have been discarded, and the assembler statements (generated within the macro) would have been in error.

Another example of a need for multiple copies of an assembler statement can be taken from the microprocessor instruction set. An 8080A/8085A RLC instruction rotates the accumulator (A register) one bit-position to the left. You may need to rotate the accumulator four bit-positions to the left; the 8080A/8085A does not provide this as a primitive instruction. Ordinarily, you would have to generate four consecutive RLC instructions; with the COPY macro, you can enter these four statements with one line:

```
COPY    4,[ RLC]
```

Again, the brackets surround the second parameter to retain the required leading space.

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 like 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 in this example: 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. (See the Add Module and Subtract Module discussions.)

This example, then, consists of seven major tasks:

1. The library ADD module is presented.
2. The library SUBTRACT module is presented.
3. The modules are entered and assembled.
4. The library generator is invoked to create the floating-point library from the two object modules.
5. A sample mainline program using the library ADD module is presented.
6. The sample mainline program is entered, assembled, and linked.
7. A parallel mainline program using the library SUBTRACT module is presented, entered, assembled, and linked.

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.

The ADD Module Statements

```
        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
```

Explanation of the ADD Module

Line 1 enables the linker to generate a listing of all internal (non-global) symbols with their relocated values. Although you wouldn't normally enable this feature in a library module, you can use it here to observe the normally invisible linker operations.

Line 2 declares the name of the object module generated by the assembler from these source statements. 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 the result (DEST).

Lines 8 through 10 define the executable-instruction section named FP_ ADD. This assembler section contains 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. (See the following discussion.)

Line 11 designates the end of this assembler module.

Entry Points

This library module defines two entry points:

- Your program can call this subroutine at FP.ADD to add SCR1 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 SCR2, then falls through to the routine at 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.

The SUBTRACT Module Statements

```
         LIST     DBG                     ; line 1
         NAME     FP$SUB                  ; line 2
         GLOBAL   FP.SUB, FP.SU2          ; line 3
         GLOBAL   FP.AD2                  ; line 4
         COMMON   FP$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
         CALL     FP.AD2                  ; line 12
         BLOCK    35                      ; line 13
         END                              ; line 14
```

Explanation of the SUBTRACT Module

Line 1 enables the linker to generate a listing of all internal (non-global) symbols with their relocated values.

Line 2 declares the name of the object module generated by the assembler from these source statements: 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 into 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 named differently 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 executable-instruction section named FPSUB. This assembler section contains the instructions that perform the subtraction. Two entry points are defined here: FP.SUB and FP.SU2. (See the following discussion.)

Line 14 designates the end of this assembler routine.

Entry Points

This library module defines two entry points:

- Your program can call this subroutine at FP.SUB to subtract SORC1 from DST, leaving the result in DST.
- Your program can call the subroutine at FP.SU2 to subtract SORC1 from SORC2, leaving the result in DST.

The routine starting at FP.SUB copies the contents of DST to SORC2, then falls through to FP.SU2. The routine beginning at FP.SU2 changes the sign of SORC1, and calls FP.AD2 to complete the subtraction. (The 8080A/8085A instruction at line 12 is a call to a subroutine, and returns to the address following the instruction.)

Entering the Modules

You may use the operating system Editor to enter these two modules into their respective assembler source files. The ADD module will be placed in a file named FPA.ASM, and the SUBTRACT module will be placed in a file named FPS.ASM. The underlined entries indicate what you enter.

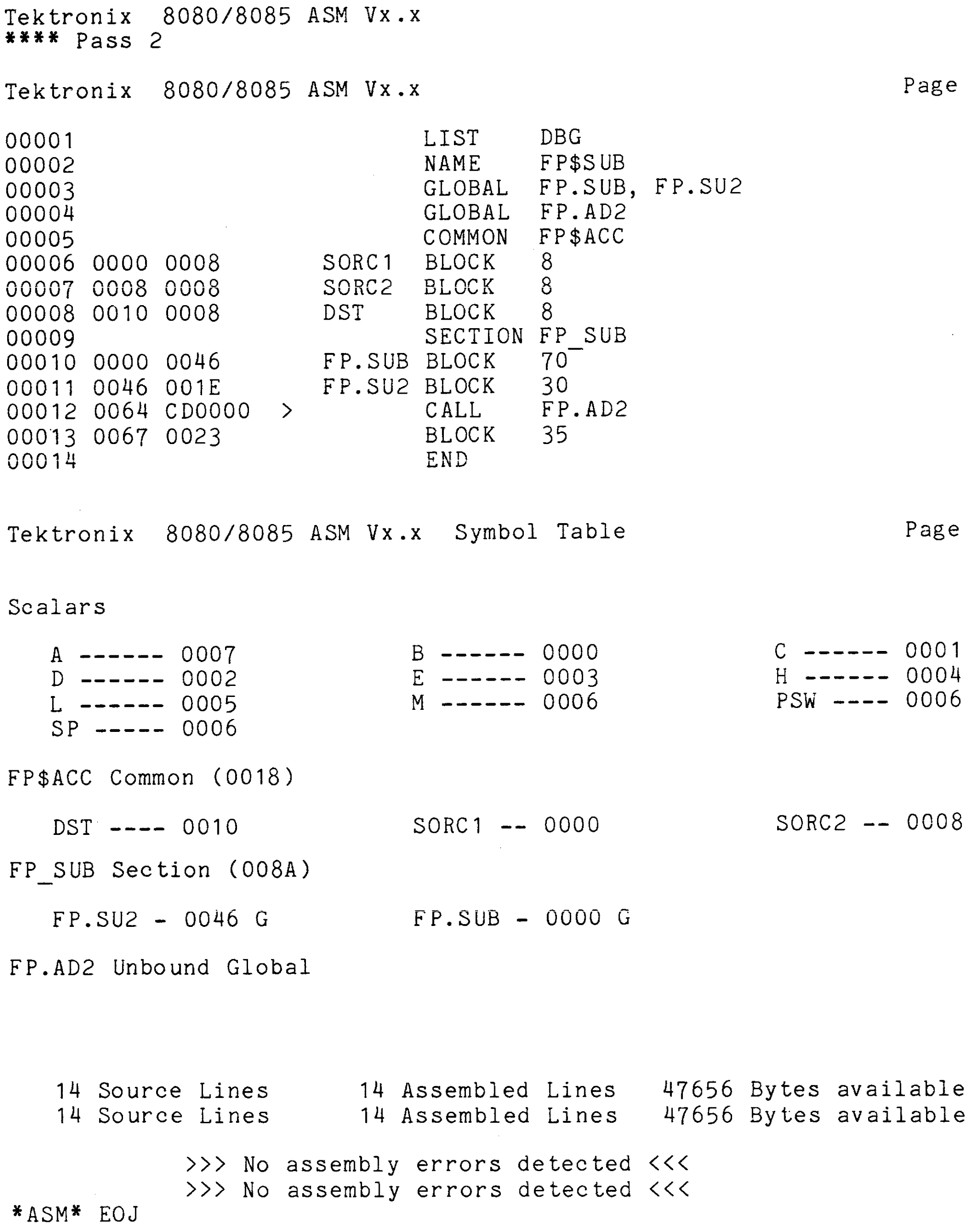
```
[Create the addition source file with the editor.]
> EDIT FPA.ASM

 ** EDIT VERSION x.x
 ** NEW FILE
[Define a visible tab character, and enter the assembly statements.]
*XTABS ON:TAB \:INPUT
INPUT:
\LIST\DBG
\NAME\FP$ADD
\GLOBAL\FP.ADD, FP.AD2
\COMMON\FP$ACC
SRC1\BLOCK\8
SRC2\BLOCK\8
DEST\BLOCK\8
\SECTION\FP_ADD
FP.ADD\BLOCK\40
FP.AD2\BLOCK\350
\END

[Display the statements with the tab characters expanded to spaces.]
*TYPE B-E
              LIST     DBG
              NAME     FP$ADD
              GLOBAL   FP.ADD, FP.AD2
              COMMON   FP$ACC
SRC1          BLOCK    8
SRC2          BLOCK    8
DEST          BLOCK    8
              SECTION  FP_ADD
FP.ADD        BLOCK    40
FP.AD2        BLOCK    350
              END
*FILE
 ** END OF TEXT
 ** EOF
```

```
[Now follow the same procedure for the subtraction source file.]
> EDIT FPS.ASM

 ** EDIT VERSION x.x
 ** NEW FILE
*XTABS ON:TAB \:INPUT
INPUT:
\LIST\DBG
\NAME\FP$SUB
\GLOBAL\FP.SUB, FP.SU2
\GLOBAL\FP.AD2
\COMMON\FP$ACC
SORC1\BLOCK\8
SORC2\BLOCK\8
DST\BLOCK\8
\SECTION\FP_SUB
FP.SUB\BLOCK\70
FP.SU2\BLOCK\30
\CALL\FP.AD2
\BLOCK\35
\END

*TYPE B-E
       LIST    DBG
       NAME    FP$SUB
       GLOBAL  FP.SUB, FP.SU2
       GLOBAL  FP.AD2
       COMMON  FP$ACC
SORC1  BLOCK   8
SORC2  BLOCK   8
DST    BLOCK   8
       SECTION FP_SUB
FP.SUB BLOCK   70
FP.SU2 BLOCK   30
       CALL    FP.AD2
       BLOCK   35
       END
*FILE
 **END OF TEXT
 **EOF
```

Assembling the Modules

Now that you've entered the programs, you may assemble them to generate the necessary object modules for the library.

```
[Assemble the source FPA.ASM into the object FPA.OBJ.  The listing is
output  to CONO (the system terminal), so that you may examine it.]
> ASM FPA.OBJ CONO FPA.ASM

Tektronix  8080/8085 ASM Vx.x
**** Pass 2

Tektronix  8080/8085 ASM Vx.x                                 Page     1

00001                         LIST     DBG
00002                         NAME     FP$ADD
00003                         GLOBAL   FP.ADD, FP.AD2
00004                         COMMON   FP$ACC
00005 0000 0008        SRC1   BLOCK    8
00006 0008 0008        SRC2   BLOCK    8
00007 0010 0008        DEST   BLOCK    8
00008                         SECTION  FP_ADD
00009 0000 0028        FP.ADD BLOCK    40
00010 0028 015E        FP.AD2 BLOCK    350
00011                         END

Tektronix  8080/8085 ASM Vx.x  Symbol Table                   Page     2


Scalars

   A ------ 0007            B ------ 0000            C ------ 0001
   D ------ 0002            E ------ 0003            H ------ 0004
   L ------ 0005            M ------ 0006            PSW ---- 0006
   SP ----- 0006

FP$ACC Common (0018)

   DEST --- 0010            SRC1 --- 0000            SRC2 --- 0008

FP_ADD Section (0186)

   FP.AD2 - 0028 G          FP.ADD - 0000 G



   11 Source Lines       11 Assembled Lines   47672 Bytes available
   11 Source Lines       11 Assembled Lines   47672 Bytes available

            >>> No assembly errors detected <<<
            >>> No assembly errors detected <<<
*ASM* EOJ


  [Now do the same for the subtract module: assemble FPS.ASM into FPS.OBJ.]
  > ASM FPS.OBJ CONO FPS.ASM
```



```
Tektronix  8080/8085 ASM Vx.x
**** Pass 2

Tektronix  8080/8085 ASM Vx.x                                          Page      1

00001                              LIST     DBG
00002                              NAME     FP$SUB
00003                              GLOBAL   FP.SUB, FP.SU2
00004                              GLOBAL   FP.AD2
00005                              COMMON   FP$ACC
00006 0000 0008           SORC1    BLOCK    8
00007 0008 0008           SORC2    BLOCK    8
00008 0010 0008           DST      BLOCK    8
00009                              SECTION  FP_SUB
00010 0000 0046           FP.SUB   BLOCK    70
00011 0046 001E           FP.SU2   BLOCK    30
00012 0064 CD0000  >               CALL     FP.AD2
00013 0067 0023                    BLOCK    35
00014                              END

Tektronix  8080/8085 ASM Vx.x  Symbol Table                            Page      2


Scalars

   A ------ 0007              B ------ 0000              C ------ 0001
   D ------ 0002              E ------ 0003              H ------ 0004
   L ------ 0005              M ------ 0006              PSW ---- 0006
   SP ----- 0006

FP$ACC Common (0018)

   DST ---- 0010              SORC1 -- 0000              SORC2 -- 0008

FP_SUB Section (008A)

   FP.SU2 - 0046 G            FP.SUB - 0000 G

FP.AD2 Unbound Global



   14 Source Lines      14 Assembled Lines    47656 Bytes available
   14 Source Lines      14 Assembled Lines    47656 Bytes available

            >>> No assembly errors detected <<<
            >>> No assembly errors detected <<<
*ASM* EOJ
```

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.

```
[Invoke LibGen in interactive mode.]
> LIBGEN

[Select the listing file name.]
*LIST FP.LBGL

[Designate the name of the library.]
*NEWLIB FP.LIB

[Now enter the list of object files to be included in this library.]
*INSERT FPS.OBJ
*INSERT FPA.OBJ

[All finished...  terminate with the END command.]
*END

NEW LIBRARY GENERATED:  FP.LIB

MODULE: FP$SUB   FROM FPS.OBJ   INSERTED
MODULE: FP$ADD   FROM FPA.OBJ   INSERTED
*LIBGEN* EOJ

[Display the listing on the system terminal.]
> COP FP;N

Tektronix  Library Generator  Vx.x     COMMAND LOG          Page     1


LIST FP.LBGL
NEWLIB FP.LIB
INSERT FPS.OBJ
INSERT FPA.OBJ
END


Tektronix  Library Generator  Vx.x   SYMBOLS DEFINED        Page     2



MODULE: FP$SUB
  (S) FP$ACC    (S) FP_SUB    (E) FP.SUB    (E) FP.SU2


MODULE: FP$ADD
  (S) FP$ACC    (S) FP_ADD    (E) FP.ADD    (E) FP.AD2

Tektronix  Library Generator  Vx.x   SUMMARY OF ACTION      Page     3


NEW LIBRARY GENERATED:  FP.LIB

MODULE: FP$SUB   FROM FPS.OBJ   INSERTED
MODULE: FP$ADD   FROM FPA.OBJ   INSERTED
*COPY* EOJ
```

Notice that the subtraction routine is placed **before** the addition routine in the library. The sample mainline programs (presented later) show why the modules are inserted in this order.

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    DBG                     ; line 1
        NAME    MAIN.ADD                ; line 2
        GLOBAL  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
        CALL    FP.ADD                  ; line 10
MORE    BLOCK   50                      ; line 11
        END     ENTRY                   ; line 12
```

Explanation of the Mainline Add Program

Line 1 enables the linker to display all internal (non-global) symbols with their relocated values. This feature of the linker enables you to examine normally invisible operations.

Line 2 gives the name MAIN.ADD to the object module.

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. 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 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 gathered into the MAIN section.

Line 9 sets aside memory space for an unspecified number of processor instructions; these instructions load values into S1 and DESTN for processing. In a real program, this BLOCK directive would be replaced with microprocessor instructions, such as data transfer instructions or I/O operations.

Line 10 is an 8080A/8085A instruction. The subroutine FP.ADD (contained in the floating-point library) is invoked. The contents of S1 are added to the contents of DESTN, and the subroutine returns to the memory location following the CALL 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 operand ENTRY is designated as the starting address of the instructions. The value of this address will be passed along to the linker; the linker then determines its relocated address, and displays this final value as a transfer address.

Entering, Assembling, and Linking the Program

The mainline add program can be entered, assembled, and linked using the following command entries:

```
[Invoke the editor to enter the program into file MNA.ASM.]
> EDIT MNA.ASM

** EDIT VERSION x.x
** NEW FILE

[Select a visible tab character and enter the assembly statements.]
*XTABS ON:TAB \:INPUT
INPUT:
\LIST\DBG
\NAME\MAIN.ADD
\GLOBAL\FP.ADD
\COMMON\FP$ACC
S1\BLOCK\8
S2\BLOCK\8
DESTN\BLOCK\8
\SECTION\MAIN
ENTRY\BLOCK\40
\CALL\FP.ADD
MORE\BLOCK\50
\END\ENTRY

[Display the lines with tab characters expanded to spaces.]
*TYPE B-E
       LIST    DBG
       NAME    MAIN.ADD
       GLOBAL  FP.ADD
       COMMON  FP$ACC
S1     BLOCK   8
S2     BLOCK   8
DESTN  BLOCK   8
       SECTION MAIN
ENTRY  BLOCK   40
       CALL    FP.ADD
MORE   BLOCK   50
       END     ENTRY
*FILE
 **END OF TEXT
 **EOF
```

```
[Assemble MNA.ASM into MNA.OBJ.]
> ASM MNA.OBJ CONO MNA.ASM

Tektronix  8080/8085 ASM Vx.x
**** Pass 2

Tektronix  8080/8085 ASM Vx.x                              Page     1

00001                        LIST    DBG
00002                        NAME    MAIN.ADD
00003                        GLOBAL  FP.ADD
00004                        COMMON  FP$ACC
00005 0000 0008       S1     BLOCK   8
00006 0008 0008       S2     BLOCK   8
00007 0010 0008       DESTN  BLOCK   8
00008                        SECTION MAIN
00009 0000 0028       ENTRY  BLOCK   40
00010 0028 CD0000  >         CALL    FP.ADD
00011 002B 0032       MORE   BLOCK   50
00012      0000    >         END     ENTRY

Tektronix  8080/8085 ASM Vx.x  Symbol Table                Page     2


Scalars

   A ------ 0007          B ------ 0000          C ------ 0001
   D ------ 0002          E ------ 0003          H ------ 0004
   L ------ 0005          M ------ 0006          PSW ---- 0006
   SP ----- 0006

FP$ACC Common (0018)

   DESTN -- 0010          S1 ----- 0000          S2 ----- 0008

MAIN Section (005D)

   ENTRY -- 0000          MORE --- 002B

FP.ADD Unbound Global



   12 Source Lines      12 Assembled Lines     47669 Bytes available
   12 Source Lines      12 Assembled Lines     47669 Bytes available

          >>> No assembly errors detected <<<
          >>> No assembly errors detected <<<
*ASM* EOJ


[Link the mainline program together with the floating-point library.]
> LINK MNA.LOAD MNA.LNKL MNA.OBJ LIB(FP.LIB)

    NO ERRORS     NO UNDEFINED SYMBOLS
    2  MODULES    3  SECTIONS
    TRANSFER ADDRESS IS 019E
*LINK* EOJ
```

```
[Display the linker listing file on the system terminal.]
> COP MNA.LNKL

Tektronix  8080/8085 LINKER V x.x       GLOBAL SYMBOL LIST    Page      1


   FP$ACC     0000  FP.AD2    0040  FP.ADD    0018  FP_ADD    0018
   MAIN       019E

Tektronix  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST  Page      2



FILE:  MNA.OBJ

MODULE:  MAIN.ADD

     SCALARS:
      A          0007     B          0000     C          0001     D          0002
      E          0003     H          0004     L          0005     M          0006
      PSW        0006     SP         0006

      LABELS:  (SECTION  FP$ACC   )
       DESTN     0010     S1         0000     S2         0008

      LABELS:  (SECTION  MAIN     )
       ENTRY     019E     MORE       01C9

Tektronix  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST  Page      3



FILE:  FP.LIB

MODULE:  FP$ADD

     SCALARS:
      A          0007     B          0000     C          0001     D          0002
      E          0003     H          0004     L          0005     M          0006
      PSW        0006     SP         0006

      LABELS:  (SECTION  FP$ACC   )
       DEST      0010     SRC1       0000     SRC2       0008

      LABELS:  (SECTION  FP_ADD   )
       FP.AD2    0040     FP.ADD     0018
```

```
Tektronix  8080/8085 LINKER V x.x       MODULE MAP              Page      4

FILE:  MNA.OBJ

MODULE:  MAIN.ADD
   FP$ACC     COMMON BYTE 0000-0017
   MAIN       SECTION BYTE 019E-01FA


FILE:  FP.LIB

MODULE:  FP$ADD
   FP$ACC     COMMON BYTE 0000-0017
   FP_ADD     SECTION BYTE 0018-019D
   FP.AD2     0040  FP.ADD     0018

Tektronix  8080/8085 LINKER V x.x       MEMORY MAP              Page      5


   0000-0017  FP$ACC     COMMON BYTE
   0018-019D  FP_ADD     SECTION BYTE
   019E-01FA  MAIN       SECTION BYTE

    NO ERRORS      NO UNDEFINED SYMBOLS
    2  MODULES     3  SECTIONS
    TRANSFER ADDRESS IS 019E
*COPY* EOJ
```

Linking Explanation

The library module containing the floating-point addition routine is automatically linked in with the mainline program. The linker determined that a global symbol (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$ADD provided a value for this global symbol. The linker included module FP$ADD in the load module. This process is illustrated in Fig. 9-4.

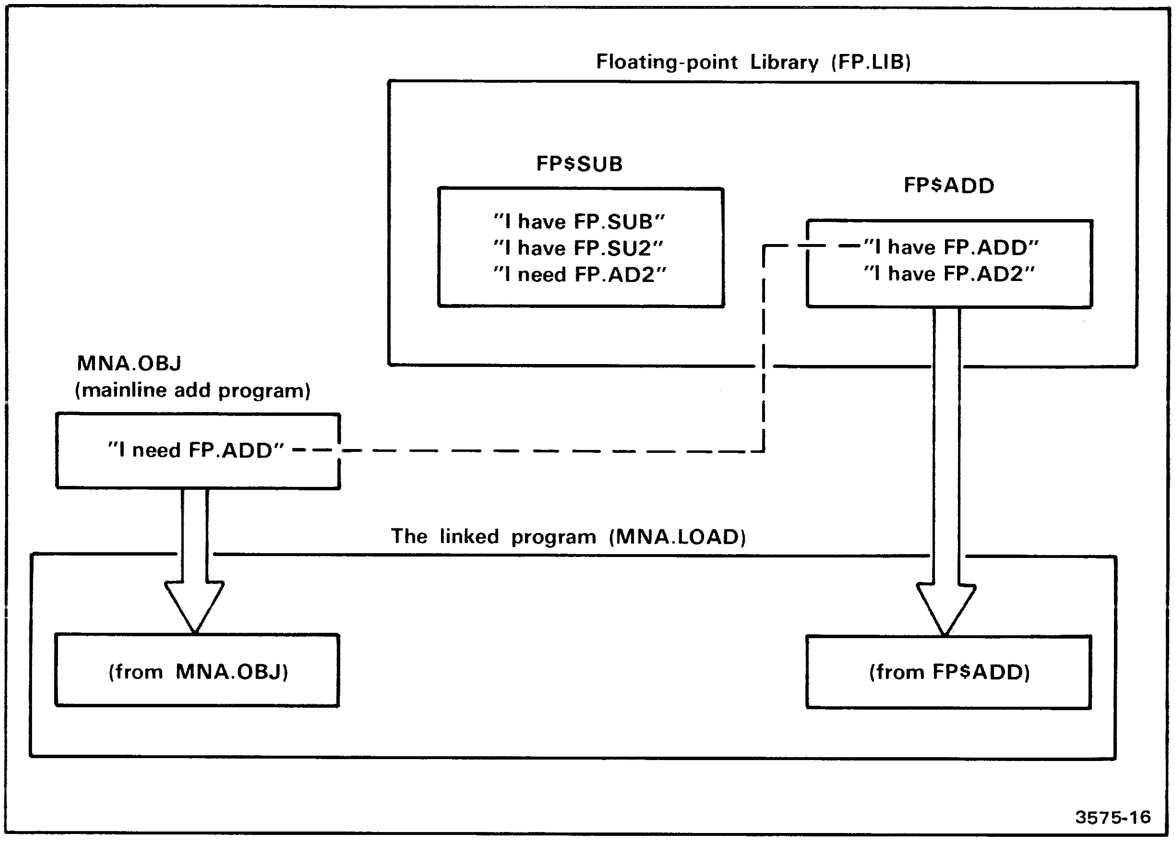


Fig. 9-4. Linking the add program to the library.

In this example, MNA.OBJ needs a definition for its unbound global symbol, FP.ADD. The linker examines the contents of the library FP.LIB, and locates module FP$ADD, which provides a definition for FP.ADD. Both MNA.OBJ and module FP$ADD are then included in the final load file. FP$SUB does not provide definitions for any unbound globals, so 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 satisfy the request made by the mainline program, and another module from the library to satisfy the first library module.

The Mainline Subtract Program

```
        LIST    DBG                     ; line 1
        NAME    MAIN.SUB                ; line 2
        GLOBAL  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
        CALL    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 symbol requested in lines 3 and 10 is FP.SUB, not 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.

Entering, Assembling,and Linking the Program

Like the mainline add program, the mainline subtract program can be entered, assembled, and linked using the following command entries:

```
[Invoke the editor to create MNS.ASM.]
> EDIT MNS.ASM

** EDIT VERSION x.x
** NEW FILE

[Select a visible tab character, and enter the assembly statements.]
*XTABS ON:TAB \:INPUT
INPUT:
\LIST\DBG
\NAME\MAIN.SUB
\GLOBAL\FP.SUB
\COMMON\FP$ACC
S1\BLOCK\8
S2\BLOCK\8
DESTN\BLOCK\8
\SECTION\MAIN
ENTRY\BLOCK\45
\CALL\FP.SUB
MORE\BLOCK\35
\END\ENTRY
```

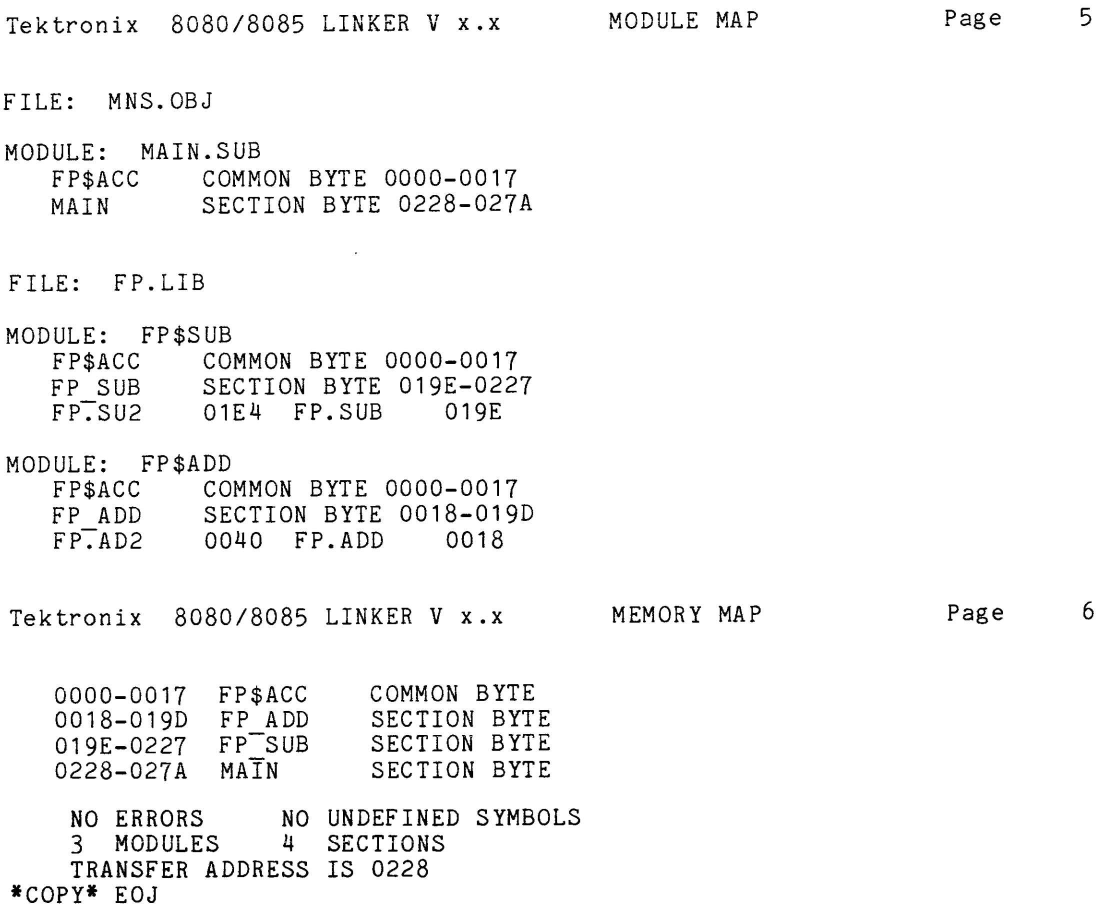
```
[Display the expanded statements.]
*TYPE B-E
        LIST    DBG
        NAME    MAIN.SUB
        GLOBAL  FP.SUB
        COMMON  FP$ACC
S1      BLOCK   8
S2      BLOCK   8
DESTN   BLOCK   8
        SECTION MAIN
ENTRY   BLOCK   45
        CALL    FP.SUB
MORE    BLOCK   35
        END     ENTRY
*FILE
 ** END OF TEXT
 ** EOF

[Assemble the source file into an object file.]
> ASM MNS.OBJ CONO MNS.ASM

Tektronix  8080/8085 ASM Vx.x
**** Pass 2

Tektronix  8080/8085 ASM Vx.x                                         Page      1

00001                           LIST    DBG
00002                           NAME    MAIN.SUB
00003                           GLOBAL  FP.SUB
00004                           COMMON  FP$ACC
00005 0000 0008          S1     BLOCK   8
00006 0008 0008          S2     BLOCK   8
00007 0010 0008          DESTN  BLOCK   8
00008                           SECTION MAIN
00009 0000 002D          ENTRY  BLOCK   45
00010 002D CD0000  >            CALL    FP.SUB
00011 0030 0023          MORE   BLOCK   35
00012      0000    >            END     ENTRY
```

```
Tektronix  8080/8085 ASM Vx.x  Symbol Table                    Page     2

Scalars

   A ------ 0007            B ------ 0000            C ------ 0001
   D ------ 0002            E ------ 0003            H ------ 0004
   L ------ 0005            M ------ 0006            PSW ---- 0006
   SP ----- 0006

FP$ACC Common (0018)

   DESTN -- 0010            S1 ----- 0000            S2 ----- 0008

MAIN Section (0053)

   ENTRY -- 0000            MORE --- 0030

FP.SUB Unbound Global


   12 Source Lines      12 Assembled Lines   47669 Bytes available
   12 Source Lines      12 Assembled Lines   47669 Bytes available

            >>> No assembly errors detected <<<
            >>> No assembly errors detected <<<
*ASM* EOJ
```

```
[Link the mainline program with the library.]
> LINK MNS.LOAD MNS.LNKL MNS.OBJ LIB(FP.LIB)

    NO ERRORS     NO UNDEFINED SYMBOLS
    3  MODULES    4  SECTIONS
    TRANSFER ADDRESS IS 0228
*LINK* EOJ
```

```
[Display the resulting linker listing file.]
> COP MNS.LNKL

Tektronix  8080/8085 LINKER V x.x       GLOBAL SYMBOL LIST     Page      1


   FP$ACC     0000  FP.AD2     0040  FP.ADD     0018  FP.SU2     01E4
   FP.SUB     019E  FP_ADD     0018  FP_SUB     019E  MAIN       0228
```

```
Tektronix  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST   Page      2



FILE:  MNS.OBJ

MODULE:  MAIN.SUB

     SCALARS:
       A         0007     B          0000     C          0001    D          0002
       E         0003     H          0004     L          0005    M          0006
       PSW       0006     SP         0006

      LABELS:  (SECTION   FP$ACC    )
       DESTN      0010    S1         0000     S2         0008

      LABELS:  (SECTION   MAIN      )
       ENTRY      0228    MORE       0258

Tektronix  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST   Page      3



FILE:  FP.LIB

MODULE:  FP$SUB

     SCALARS:
       A         0007     B          0000     C          0001    D          0002
       E         0003     H          0004     L          0005    M          0006
       PSW       0006     SP         0006

      LABELS:  (SECTION   FP$ACC    )
       DST        0010    SORC1      0000     SORC2      0008

      LABELS:  (SECTION   FP SUB    )
       FP.SU2     01E4    FP.SUB     019E
Tektronix  8080/8085 LINKER V x.x       INTERNAL SYMBOL LIST   Page      4



FILE:  FP.LIB

MODULE:  FP$ADD

     SCALARS:
       A         0007     B          0000     C          0001    D          0002
       E         0003     H          0004     L          0005    M          0006
       PSW       0006     SP         0006

      LABELS:  (SECTION   FP$ACC    )
       DEST       0010    SRC1       0000     SRC2       0008

      LABELS:  (SECTION   FP_ADD    )
       FP.AD2     0040    FP.ADD     0018
```



```
Tektronix  8080/8085 LINKER V x.x       MODULE MAP            Page     5


FILE:  MNS.OBJ

MODULE:  MAIN.SUB
   FP$ACC    COMMON BYTE 0000-0017
   MAIN      SECTION BYTE 0228-027A


FILE:  FP.LIB

MODULE:  FP$SUB
   FP$ACC    COMMON BYTE 0000-0017
   FP_SUB    SECTION BYTE 019E-0227
   FP‾.SU2   01E4  FP.SUB    019E

MODULE:  FP$ADD
   FP$ACC    COMMON BYTE 0000-0017
   FP_ADD    SECTION BYTE 0018-019D
   FP‾.AD2   0040  FP.ADD    0018

Tektronix  8080/8085 LINKER V x.x       MEMORY MAP            Page     6


   0000-0017  FP$ACC    COMMON BYTE
   0018-019D  FP_ADD    SECTION BYTE
   019E-0227  FP‾SUB    SECTION BYTE
   0228-027A  MAI‾N     SECTION BYTE

    NO ERRORS      NO UNDEFINED SYMBOLS
    3  MODULES     4  SECTIONS
    TRANSFER ADDRESS IS 0228
*COPY* EOJ
```

Linking Explanation

For the mainline subtract program, the library module FP$SUB is referenced using the global symbol FP.SUB. This brings module FP$SUB into the final load module. However, FP$SUB itself contains a reference to an unbound global symbol, FP.AD2. The definition for this unbound global symbol is found in the FP$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. 9-5.

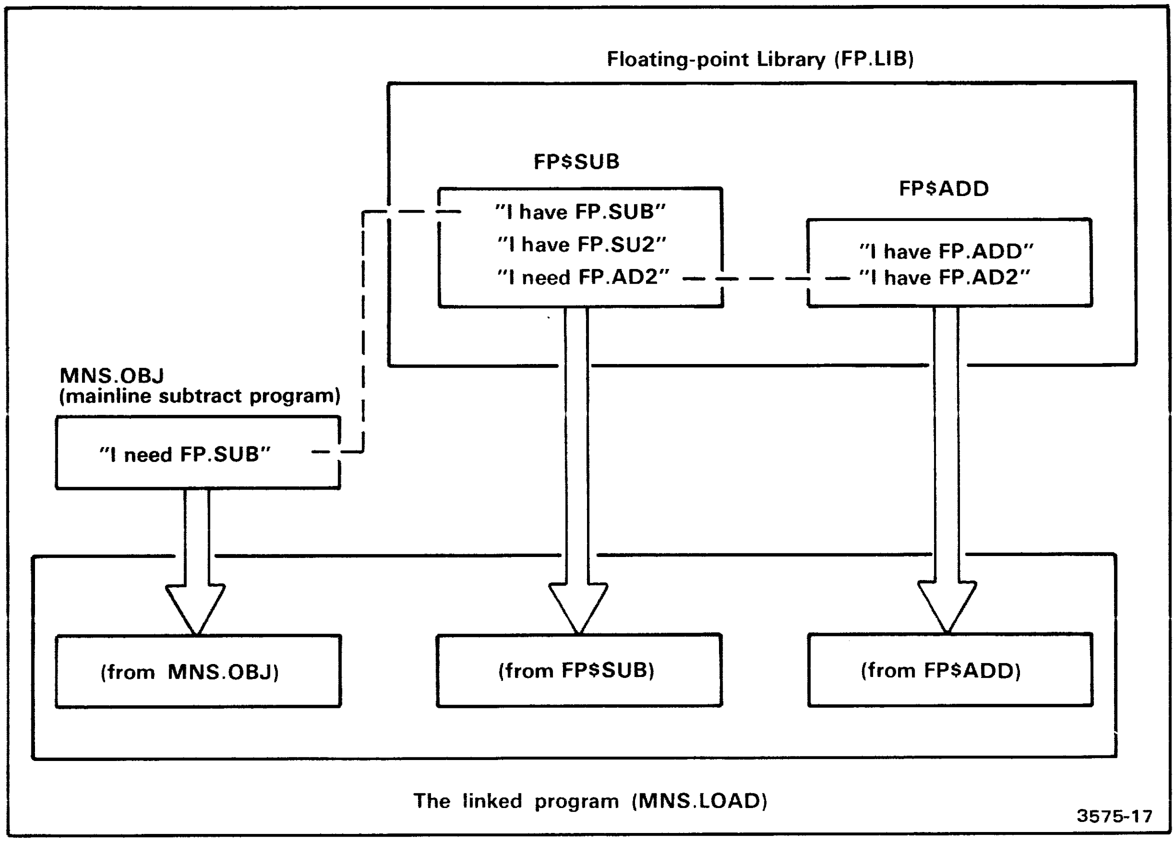


Fig. 9-5. Linking the subtract program to the library.

In this linking example, MNS.OBJ requests definition for an unbound global, FP.SUB. The linker scans the library (starting at the left in this figure), and locates a definition for FP.SUB, in module FP$SUB. However, FP$SUB itself contains a reference to an unbound global symbol, FP.AD2. The linker continues to scan the library, and finds a definition for FP.AD2 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.

This example illustrates why the subtract module (FP$SUB) was placed **before** the add module (FP$ADD). The linker scans the modules of a library only once, in a front-to-back order. If FP$SUB had been located after FP$ADD, then the linker would not have found the definition for the FP.AD2 symbol after linking in FP$SUB.

DOS/50 SVC GENERATION

This example explores two areas of DOS/50 service calls: creating service request blocks (SRBs), and generating the required microprocessor service call (SVC) instructions. This example uses the 8080A/8085A instruction set, but similar techniques can be applied to most processors.

This example assumes you are familiar with the use of SVCs under DOS/50, as described in the Service Calls section of the 8550 System Users Manual.

Creating Service Request Blocks

The first task in using an SVC is to create an appropriate SRB. The SRB consists of eight bytes:

1. a function code,
2. a channel number,
3. a status byte,
4. a single-byte data item,
5. a byte count for I/O operations,
6. a buffer length,
7. the high-order byte of the buffer address, and
8. the low-order byte of the buffer address.

The buffer (specified by the last three bytes) is used for I/O operations.

Setting up the SRB in your source program can be made easier when you use an "intelligent" macro. The macro can decide (based on the parameters you give it) whether to generate a SRB location vector, what the names of the SRB components are, what the size of the I/O buffer is, and other miscellaneous items. 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$SEC  SET     "'%'"                            ; line 3
         IF      DEF(SRB.SEC)                     ; line 4
         RESUME  SRB.SEC                          ; line 5
         ELSE                                     ; line 6
         SECTION SRB.SEC                          ; line 7
         ENDIF                                    ; line 8
'1'.FUN  BLOCK   1                                ; line 9
'1'.CHN  BLOCK   1                                ; line 10
'1'.STA  BLOCK   1                                ; line 11
'1'.DAT  BLOCK   1                                ; line 12
'1'.CNT  BLOCK   1                                ; line 13
         IF      "'3'"=""                         ; line 14
'1'.LEN  BLOCK   1                                ; line 15
'1'.HIB  BLOCK   1                                ; line 16
'1'.LOB  BLOCK   1                                ; line 17
         ELSE                                     ; line 18
         IF      "'4'"=""                         ; line 19
SRB$BUF  SET     "'1'.BUF"                        ; line 20
         ELSE                                     ; line 21
SRB$BUF  SET     "'4'"                            ; line 22
         ENDIF                                    ; line 23
'1'.LEN  BYTE    '3'                              ; line 24
'1'.HIB  BYTE    HI('SRB$BUF')                    ; line 25
'1'.LOB  BYTE    LO('SRB$BUF')                    ; line 26
         IF      DEF(BUF.SEC)                     ; line 27
         RESUME  BUF.SEC                          ; line 28
         ELSE                                     ; line 29
         SECTION BUF.SEC                          ; line 30
         ENDIF                                    ; line 31
'SRB$BUF' BLOCK '3'                               ; line 32
         ENDIF                                    ; line 33
         IF      "'2'"<>""                        ; line 34
         IF      DEF(SRB.VEC)                     ; line 35
         RESUME  SRB.VEC                          ; line 36
         ELSE                                     ; line 37
         SECTION SRB.VEC, ABSOLUTE                ; line 38
         ENDIF                                    ; line 39
         ORG     40H+2*('2'-1)                    ; line 40
         BYTE    HI('1'.FUN)                      ; line 41
         BYTE    LO('1'.FUN)                      ; line 42
         ENDIF                                    ; line 43
         RESUME  'SRB$SEC'                        ; line 44
         ENDM                                     ; line 45
```

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 four 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, the parameter designates the SVC number (1 to 6) that will be used with this SRB. If you provide this number, the macro will create the appropriate pointer to this SRB in the 40H to 4BH 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, the parameter designates the size of I/O buffer to be associated with this SRB. The last three bytes of the SRB are correctly altered to describe the buffer's size and location, and the buffer of this size is created. The name of the buffer is controlled by the fourth parameter. If you omit the third parameter, the last three bytes of the SRB are left empty.
4. The fourth parameter is optional; if present, the parameter selects the name of the buffer associated with this SRB. If you omit this parameter, the macro chooses a name derived from the SRB name. This parameter will be ignored if the third parameter is not present.

Line 1 creates two string assembler 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 further discussed 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 whether or not the section SRB.SEC was previously started: if so, a simple RESUME statement is processed, to continue object code generation; if not, the section is begun with a SECTION statement, as its first definition. This technique of using IF DEF(section-name) to conditionally resume a section is used twice again, starting in lines 27 and 35.

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.DAT (byte data), and QQ.CNT (I/O count).

Lines 14 through 33 generate the last three bytes of the SRB, and create the buffer (if necessary). Three possible combinations exist:

1. **No third parameter:** the last three 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 three 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 three 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 absent, lines 15 through 17 are assembled; if present, lines 19 through 32 are assembled. In either case, the other block of statements is skipped.

Lines 15 through 17 generate the last three bytes of the SRB when the third parameter is absent. Again, the names of the three bytes are derived from the SRB name given in the macro invocation line. If the SRB name is QQ, for example, three bytes are generated: QQ.LEN (length of buffer), QQ.HIB (high byte of the buffer address), and QQ.LOB (low byte of the buffer address).

Lines 19 through 23 determine the name of the buffer. 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. If a fourth parameter is present, then it is used as the buffer name. In either case, the buffer name is assigned to the assembler string variable SRB$BUF. This variable is used later in the macro.

Lines 24 through 26 generate the last three 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.HIB, and QQ.LOB. 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 27 through 31 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 32 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 34 through 43 generate a pointer to the SRB in the SRB vector (fixed locations 40H to 4BH) only if the second parameter is present.

Lines 35 through 39 define SRB.VEC as the current section. This section is absolute (non-relocatable), because the vectors must be in fixed locations in memory.

Line 40 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 6.

Lines 41 and 42 generate a pointer to the SRB's first entry, the function byte.

Line 44 restores the current section to the section name that was saved upon entry into this macro.

Line 45 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 name of the SRB is specified in the first parameter, QQQ. 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, like this:

```
SRB     RRR, 4
```

The SRB macro automatically places the appropriate pointer to the SRB (named RRR) at locations 46H and 47H. 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, include the third and fourth parameters. For example,

```
SRB     SSS, , 128, BUFFER
```

specifies a BUFFER that is 128 bytes long. This buffer is created automatically by the macro. The macro also places values (describing the location and length of the buffer) into the last three bytes of the SRB, relieving your program of this responsibility.

If you do not require a specific buffer name, omit the fourth parameter. The 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 44H and 45H (the SVC 3 vector location), and an 80-byte buffer named MYBUF.

Generating Service Calls

The task of generating the service call consists of placing two microprocessor-dependent instructions in your program. The first instruction is usually a data transfer instruction, while the second is a no-operation instruction. For an 8080A/8085A 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       "'1'"=""                            ; line 2
WARNING ; Missing SVC Number                 ; line 3
ELSE                                         ; line 4
OUT      OF8H-'1'                            ; line 5
NOP                                          ; line 6
ENDIF                                        ; line 7
ENDM                                         ; line 8
```

Explanation of the SVC Macro

This macro is invoked with one parameter, the SVC number: a single digit between 1 and 6.

Line 1 defines the name of the SVC-generation macro.

Lines 2 through 7 form an IF..ELSE..ENDIF block. If the first parameter is absent, line 3 is processed. If the first parameter is present, lines 5 and 6 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 and 6 generate an 8080A/8085A service call instruction sequence. Line 5 generates an OUT instruction; the address of the OUT instruction is computed from the first macro parameter. Line 6 generates the required NOP (no-operation) instruction.

Line 8 terminates the macro definition.

Sample Invocation of the SVC Macro

The SVC macro is simple to invoke: simply provide the SVC number as the first parameter. For example,

```
SVC      4
```

generates the proper instruction sequence for SVC 4. If the first 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. It ends by switching 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$CNT'                            ; line 13
 CON$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 it 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 will then process the next 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.

Line 19 through 21 define macro ENDCONSTANT. The macro ENDCONSTANT simply switches the current section 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 would like 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. The subroutine prints each successive character pointed to by the HL 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
LXI        H,MES1
CONSTANT   MES1 ASCII "HELLO THERE",[ BYTE 13]

                 1st parameter        2nd parameter

CALL       PRINT
```

The first line declares section PRINCON.

The second line is an 8080A/8085A instruction that loads the HL 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, CALL PRINT, invokes the subroutine PRINT.

Macro CONSTANT simply switches to section ROM.CODE when you do not supply any parameters. 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
          LXI        H,MES1
          CONSTANT
MES1      ASCII      "HELLO THERE"
          BYTE       13
          ENDCONSTANT
          CALL       PRINT
```

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:

```
          LXI       H,DATA.TAB
          VARIABLE  DATA.TAB BLOCK 512
          CALL      PROCESS
```

or this sequence:

```
          LXI       H,DATA.TAB
          VARIABLE
DATA.TAB  BLOCK     512
          ENDVARIABLE
          CALL      PROCESS
```

SAVE-AND-RESTORE MACRO

This example uses two assembler macros in a common assembly language operation: saving and restoring microprocessor registers. The example uses the 8080A/8085A instruction set; however, the techniques illustrated here can be applied to nearly all stack-oriented microprocessors.

The SAVE Macro

```
        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    B, D, H, PSW      ; line 9
        ENDIF                     ; line 10
        ENDM                      ; line 11
```

This macro is used to save one or more registers on the stack. The parameters of the macro invocation line designate the registers to be saved on the stack. The body of this macro examines those parameters and generates the appropriate 8080A/8085A PUSH instructions.

Line 1 begins the macro definition, and gives the macro the name SAVE. This name will be used later 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 non-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 invocation statement consisted solely of the word SAVE), 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. The macro must generate one PUSH instruction for each parameter provided; each parameter is the name of one 8080A/8085A register pair. A REPEAT..ENDR loop processes each parameter in turn.

Line 3 initializes the assembler variable SAVE$ to 1. This assembler variable will be incremented once for each parameter given in the macro invocation line.

Line 4 designates the beginning of the REPEAT..ENDR loop. The loop is repeated as long as the assembler variable SAVE$ is not greater than the number of parameters passed to the macro ('#').

Line 5 generates an 8080A/8085A 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 H, this statement generates an 8080A/8085A PUSH H instruction.

Line 6 increments the value of the assembler variable SAVE$.

Line 7 terminates the definition of the REPEAT..ENDR loop. As long as the expression specified 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 (zero). If SAVE is entered with no parameters, the SAVE macro reinvokes itself with all four possible parameters, thereby saving all four register pairs.

Line 10 terminates the IF..ELSE..ENDIF block.

Line 11 statement terminates the definition of the macro.

The RESTORE Macro

```
         MACRO    RESTORE
         IF       '#'
RESTORE$ SET      1
         REPEAT   RESTORE$ <= '#'
         POP      'RESTORE$'
RESTORE$ SET      RESTORE$ + 1
         ENDR
         ELSE
         RESTORE PSW, H, D, B
         ENDIF
         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 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             ; 8080A/8085A 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    H, PSW          ; Save H-L and PSW-A
        ...
        ...                     ; Body of subroutine
        ...
        RESTORE PSW, H          ; Restore PSW-A, H-L
        RET                     ; Return from subroutine
```

CONDITIONAL ASSEMBLY

This example illustrates some uses of the IF-ELSE-ENDIF or IF-ENDIF constructs for conditional assembly.

Three typical examples are provided: (1) processor-independent programming, (2) use of conditional assembly in macros, and (3) assembly based on relative memory locations.

Processor-Independent Programming

A program may be written to run on two or more similar processors by placing the processor-dependent instructions in conditional blocks. These conditional blocks check the processor type and assemble the correct instructions.

```
        MACRO   SUBTRACT
; This is a processor-independent macro written for either the Z80 or the 8085
        IF      PROC="Z80"      ; If processor is a Z80...
        OR      A               ; Clear the accumulator
        SBC     HL,DE           ; Subtract DE from HL
        ELSE                    ; But, if not a Z80...
        IF      PROC="8085"     ; If processor is an 8085A
        LD      A,L             ; Transfer L to A
        SUB     E               ; Subtract E from A
        LD      L,A             ; Move A to L
        LD      A,H             ; Move H to A
        SBC     D               ; Subtract (carry+D) from A
        LD      H,A             ; Move A to H
        ELSE                    ; But, if not 8085...
        WARNING                 ; WRONG PROCESSOR - 'PROC'
        ENDIF
        ENDIF
```

Use of 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 OF 100
        ENDIF
```

These instructions will 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 of 100 characters, is assembled.

Another use of conditional assembly in macros is to verify that a symbol has previously been declared as a global symbol.

```
        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
        LD      A, '1'          ; Move first parameter to A
        LD      BC, '2'         ; Move second parameter to BC
        CALL    PRINT
        ENDM
```

The conditional block in this macro checks if PRINT has been defined as a global symbol. If PRINT has not been defined, the statement GLOBAL PRINT is assembled. If PRINT has been defined previously, the statement in the conditional block is skipped.

Assembly Based on Relative Memory Locations

Conditional assembly can also be used to keep track of the relative distance between bytes of memory; this information may then be used to decide which instructions to assemble. For example, the Z80 JR (jump relative) instruction will not allow a jump of more than 128 bytes. The relative distance between bytes must be less than +128 and greater than −127. If the range is exceeded, a warning will be issued. The following example checks to see if the relative distance between the location counter and the jump address is within the specified range. If it is, the JR instruction is assembled: if not, the JP instruction, which has no range restriction, is assembled.

```
IF      DEF(LABEL)      ; If the label is not a forward reference
IF      (($-LABEL) > -127) & (($-LABEL) < +128) ; and it's near
JR      LABEL           ; do a JR
ELSE                    ; Not close enough, so tell user..
WARNING  ; JR RANGE EXCEEDED. JP INSTRUCTION USED INSTEAD
JP      LABEL           ; and generate the JP
ENDIF
ELSE                    ; Forward reference?
JP      LABEL           ; Always a JP (can't tell where it is)
ENDIF
```

This conditional block may be inserted into a macro and invoked when the jump instruction is needed. When the macro is assembled and executed, the correct jump instruction will be assembled into the object file.

```
NAME     MAINPRO
.
.
MACRO    JUMP       ; BEGINS MACRO JUMP DEFINITION
IF       DEF('1')
IF       (($-'1') > -127) & (($-'1') < +128)
JR       '1'
ELSE
WARNING  ; JR RANGE EXCEEDED.  JP INSTRUCTION USED
JP       '1'
ENDIF
ELSE
JP       '1'
ENDIF
ENDM       ; END OF MACRO DEFINITION
.
.
JUMP     LABEL  ; INVOKES THE MACRO JUMP WITH THE PARAMETER "LABEL"
.
.
```

USING THE '@' CONSTRUCT WITHIN MACROS

This example illustrates the use of the '@' (at) construct in macros. Each time a macro is invoked, any '@' construct appearing in the macro body is replaced with a unique four-character value. When this value is appended to a one-to-four character symbol within the macro body, a unique five-to-eight character label is created. With this construct, you can 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 two separate label symbols. Even though this macro is invoked more than once, DEL1'@' and DEL2'@' will be unique each time the macro is invoked.

The number of delay loops (0 to 0FFH) is passed to the macro DELAY as the single parameter.

This example uses the 8080A/8085A instruction set, but similar techniques can be applied to most processors.

Delay Loop Macro

```
            MACRO     DELAY       ; line 1
            MVI       H, '1'      ; line 2
  DEL2'@'   MVI       L,0FFH      ; line 3
  DEL1'@'   DCR       L           ; line 4
            JNZ       DEL1'@'     ; line 5
            DCR       H           ; line 6
            JNZ       DEL2'@'     ; line 7
            ENDM                  ; line 8
```

Line 1 defines the beginning of the macro and names it DELAY.

Line 2 is an assembly language instruction that moves the value of the parameter (number of delay loops) to the H register.

Line 3 moves the value 0FFH into the L register. The label DEL2'@' is replaced by a unique eight-character symbol each time the macro is invoked.

Line 4 decrements the L register. The label DEL1'@' is replaced by another unique symbol.

Line 5 tests the L register. If it is not zero, the program jumps to the DCR L (line 4) instruction. When the L register becomes zero, the program proceeds to the instruction DCR H (line 6).

Line 6 decrements the H register (the number of delay loops requested).

Line 7 tests the H register. If it is not zero, the program jumps to the MVI L,0FFH instruction (line 3), thus repeating the DEL1'@' loop the number of delay loops requested. When the H register becomes zero, the macro is terminated.

Macro Invocation

```
        DELAY     10H        ; SHORT DELAY
          .
          .
          .
        DELAY     0FFH       ; LONG DELAY
```

In this example, the first time macro DELAY is invoked the number of delay loops is 10H. The label symbols DEL2'@' and DEL1'@' represent one set of address values. When DELAY is invoked again with OFFH delay loops requested, DEL2'@' and DEL1'@' represent another set of addresses.

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 is then assembled into the object module.

This example illustrates four ways in which the INCLUDE directive may typically be used: (1) defining constants, (2) defining COMMON sections, (3) defining macros, and (4) providing authorship notices in your listings.

Including Constant Declarations

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
        MVI      B,ROWS      ; Number of rows to B
        MVI      C,COLS      ; Number of columns to C
          .
          .
          .
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
          .
          .
         MVI      B,ROWS      ; Number of rows to B
         MVI      C,COLS      ; Number of columns to C
          .
          .
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
          .
          .
```

The INCLUDE Directive in Macros

The INCLUDE directive can be very helpful in assembly language programming. A frequently used macro may be defined in a file for later invocation.

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 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 has been INCLUDEd in MAINPRO, each invocation of macro ABC will cause the macro to be expanded at assembly time.

Authorship and Copyright Notices for Listings

The INCLUDE directive may also 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) 1980 BY                         *
;*                                                                      *
;************************************************************************
;*                                                                      *
;*       *******       *      *                       *                 *
;*          *          *      *                                         *
;*          *   ***    *   *  ****  ****  ****  ****  * *   *           *
;*          *  *   *   *  *   *    *     *    * *   * *  * *    ...     *
;*          *  ****    ***    *    *     *    * *   * *   *    . R .    *
;*          *  *       *  *   *    *     *    * *   * *  * *    ...     *
;*          *   ********   *   *** *      ****  *   * * *   *           *
;*                                                                      *
;*                                   COMMITTED TO EXCELLENCE            *
;*                                                                      *
;************************************************************************
;*                                                                      *
;*       TEKTRONIX, INCORPORATED, BEAVERTON, OREGON  97077              *
;*                                                                      *
;*                  ALL RIGHTS RESERVED                                 *
;*                                                                      *
;************************************************************************
;************************************************************************
;*                                                                      *
;*                   AUTHOR: KEN DEDATE                                 *
;*                                                                      *
;************************************************************************
```

By using a single statement, INCLUDE "CPYR.ASM", your assembler listing will take the following format.

```
        NAME    MAINPRO
        INCLUDE "CPYR.ASM"
;************************************************************************
;*                                                                      *
;*                        COPYRIGHT (C) 1980 BY                         *
;*                                                                      *
;************************************************************************
;*                                                                      *
;*       *******       *      *                       *                 *
;*          *          *      *                                         *
;*          *   ***    *   *  ****  ****  ****  ****  * *   *           *
;*          *  *   *   *  *   *    *     *    * *   * *  * *    ...     *
;*          *  ****    ***    *    *     *    * *   * *   *    . R .    *
;*          *  *       *  *   *    *     *    * *   * *  * *    ...     *
;*          *   ********   *   *** *      ****  *   * * *   *           *
;*                                                                      *
;*                                   COMMITTED TO EXCELLENCE            *
;*                                                                      *
;************************************************************************
;*                                                                      *
;*       TEKTRONIX, INCORPORATED, BEAVERTON, OREGON  97077              *
;*                                                                      *
;*                  ALL RIGHTS RESERVED                                 *
;*                                                                      *
;************************************************************************
;************************************************************************
;*                                                                      *
;*                   AUTHOR: KEN DEDATE                                 *
;*                                                                      *
;************************************************************************
         .
         .
         .
```

Section 10
TABLES

Section 10
TABLES

Table 10-1
Source Module Character Set

Symbols	Definition
A..Z	letters used in symbols; 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
;	precedes a comment
,(comma)	delimiter for operand items
"	string delimiter
:	string concatenation operator
'	string substitution delimiter
#	total number of arguments passed to current macro expansion
[ ]	treat everything within brackets as a single argument
@	provide unique labels for each macro expansion
%	replaced by name of current section in a macro expansion
*	binary arithmetic operation, multiplication
/	binary arithmetic operation, division
+	unary or binary arithmetic operator, addition
-	unary or binary arithmetic operator, subtraction
( )	override precedence of operators

Table 10-1 (cont)

Symbols	Definition
\	unary logical operator, NOT
&	binary logical operator, AND
!	binary logical operator, inclusive OR
!!	binary logical operator, exclusive OR
SPACE	field delimiter
TAB	field delimiter
CARRIAGE 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 10-2
Assembler Directives

Directive	Operation
ASCII	generates ASCII data
BLOCK	reserves a data block
BYTE	generates byte(s) of data
COMMON	declares program section, assigns name, defines type to be common
ELSE	when the expression in the IF statement is false (zero), causes assembly of alternate source lines between ELSE and ENDIF directives
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
GLOBAL	declares global symbol
IF	when expression is true (non-zero), causes assembly of source lines between IF and ENDIF (or ELSE, if present) directives
INCLUDE	inserts text from another source file
LIST	turns on assembler listing options
MACRO	defines the beginning of a macro source block
NAME	declares object module name
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

Table 10-2(cont)

Directive	Operation
RESERVE	reserves a work space 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 lines 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	generates word(s) of data

Table 10-3
ASCII Codes (Hexadecimal)

HIGH-ORDER BITS

	0	1	2	3	4	5	6	7
LOW-ORDER BITS	CONTROL		SYMBOLS		UPPERCASE		LOWERCASE	
0	NUL	DLE	SP	Ø	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL BELL	ETB	'	7	G	W	g	w
8	BS BACK SPACE	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR RETURN	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	∧	n	~
F	SI	US	/	?	O	_	o	DEL RUBOUT

Table 10-4
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 10-5
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

HEX	F + 8	=	17	
HEX	10	=	16	DEC
HEX	7	=	7	DEC
HEX	17	=	23	DEC

Table 10-6
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

HEX	9 x 8	=	48	
HEX	40	=	64	DEC
HEX	8	=	8	DEC
HEX	48	=	72	DEC

Section 11

TECHNICAL NOTES

This section is reserved for technical information about the Tektronix Assembler, Linker, and Library Generator (LibGen). At the time of this writing, no technical notes are included. Technical notes will be incorporated into later versions of this manual, as necessary.

Section 12

ASSEMBLER SPECIFICS

Processor-specific information is contained in the Assembler Specifics supplement that accompanies each assembler. Each supplement is designed as a subsection to this manual.

These Assembler Specifics supplements are numbered as if they were separate sections of this manual. For example, the 8080A/8085A supplement is labeled "Section 12A," and the first illustration is numbered "Fig. 12A-1." Similarly, other supplements are labeled Sections 12B, 12C, etc. Figures, pages, and tables are numbered accordingly.

Each subsection presents the following information:

- A demonstration run that parallels the one given for the 8080A/8085A in the Learning Guide of this manual.
- A brief summary of the processor's addressing modes and registers.
- A list of notational conventions used to describe the instruction set.
- The microprocessor instruction set in a notation acceptable to the given assembler.
- A list of reserved words for the given assembler.
- The page size for the assembler, as defined in the Linker section of this manual.
- Any processor-specific assembler error messages.
- Any irregularities that should be noted.

Section 13

ERROR MESSAGES

INTRODUCTION

This section lists the assembler error messages in numeric order. The assembler error messages with numbers above 91 are described in the Assembler Specifics section for your microprocessor. Refer to the Linker section for the linker error messages and to the Library Generator section for the LibGen error messages. Each error message is followed by a description of possible causes.

***** **ERROR 001:.** (No message) This error is generated by a user-specified WARNING directive. For more information, see the WARNING directive in the Assembler Directives section of this manual.

***** **ERROR 002: Symbol already defined.** A symbol has been redefined. This error may occur if the same symbol is equated to two different values (with EQU directives) or if two different instructions have the same label.

***** **ERROR 003: Symbol value Phase Error.** There is a difference between pass 1 and pass 2 in the value or section number of a label or symbol. This message may be caused by a SET directive with a forward reference.

***** **ERROR 004: Illegal EQU of GLOBALs.** An unbound global has been assigned the value of another unbound global (with the EQU directive).

***** **ERROR 005: Global definition may not use HI, LO, or ENDOF.** The values of HI, LO or ENDOF have been assigned to a global symbol. This error may occur when a global symbol is equated to HI(x) or LO(x), where x is an address; or ENDOF(y), where y is section name whose ending address is yet to be found.

***** **ERROR 006: String expression required.** A numeric value appears where a string is required. Concatenation, SEG or NCHR functions, and ASCII, TITLE, or STITLE directives all require string operands.

***** **ERROR 007: Undefined BLOCK or ORG expression.** The operand of an ORG or BLOCK directive is either undefined or a forward reference. This error may occur if a misspelled or undefined symbol appears in an ORG or BLOCK directive, or if these directives reference a symbol that has not yet been assigned a value.

***** **ERROR 008: Invalid ORG out of section.** The section of an ORG expression is either not a scalar or 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.

***** **ERROR 009: Negative block length.** The BLOCK operand is either negative or greater than 32767.

***** **ERROR 010: Macro already defined.** The same name appears in two or more MACRO directives.

***** **ERROR 011: Macro definition phase error.** There are two possible errors: the macro has been called before being defined, or the macro has been defined in the second (but not the first) pass of the assembler. This error may be caused by a forward reference used with a SET directive.

***** **ERROR 012: Memory full on Macro call.** There is insufficient memory space to perform macro expansion. This error may occur if no limit is set for macro recursion, if too many symbols are used in a macro definition, or if too many actual parameters are specified.

***** **ERROR 013: Missing ENDR or ENDIF.** A conditional assembly (IF or REPEAT) block is not properly terminated. This error may occur if a conditional assembly block begins within a macro definition. This error may also occur if a macro ends (with the ENDM directive) before termination of conditional assembly (by the ENDR or ENDIF directive).

***** **ERROR 014: Duplicate definition of section name.** A section name is already in use as a symbol.

***** **ERROR 015: END directive invalid within an INCLUDE file.** An END directive is present within an INCLUDE file. See the Assembler Directives section in this manual for information on INCLUDE files.

***** **ERROR 016: ENDR or ENDIF mis-matched.** An incorrect termination directive is used on a conditional assembly block. This error may occur if an ENDR is used to terminate a IF block, if an ENDIF is used to terminate a REPEAT block, or if REPEAT and IF blocks overlap each other.

***** **ERROR 017: Iteration limit 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 this parameter is not specified, the error message is displayed after 256 repeat cycles are completed.

***** **ERROR 018: Misplaced ELSE.** Either an ELSE directive is outside an IF-ENDIF block, or more than one ELSE directive is within an IF-ENDIF block.

***** **ERROR 019: Operation invalid for address.** An operation requiring scalar operands has been applied to an address value.

***** **ERROR 020: Divisor is zero.** A division or a MOD operation attempted to use zero as a divisor.

***** **ERROR 021: Text following "" ignored.** 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 Macro section of this manual for further information on parameters.

***** **ERROR 022: ENDOF operand is scalar.** The specified parameter of an ENDOF function is a scalar or non-global symbol.

***** **ERROR 023: ENDOF already applied.** An attempt has been made to perform an ENDOF function upon an address resulting from a previous ENDOF function.

***** **ERROR 024: ENDOF operand is not global.** The specified parameter of an ENDOF function is a scalar or non-global symbol.

***** **ERROR 025: Operation on HI or LO of address.** An attempt has been made to add or subtract the result of a HI or LO function.

***** **ERROR 026: Addition of addresses.** An attempt has been made to add two addresses.

***** **ERROR 027: Conflicting section bases.** An attempt has been made to subtract or compare addresses based on different sections.

***** **ERROR 028: Address subtracted from scalar.** An attempt has been made to subtract an address from a scalar value.

***** **ERROR 029: Negative string length.** A declaration in the STRING directive specifies a maximum length that is either negative or greater than 32767.

***** **ERROR 030: 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 a SET directive with a forward reference.

***** **ERROR 031: Redeclaration of string variable.** An attempt has been made to redeclare a string variable. This error may occur if a STRING directive is inside a REPEAT loop or inside a macro which is expanded more than once.

***** **ERROR 032: String declaration phase error.** A string value has been defined during the second assembler pass but not during the first pass. This error may be caused by a SET directive with a forward reference.

***** **ERROR 033: Invalid string name.** An invalid symbol is used as a string variable name in a STRING directive. See the Assembler Directives section for more information on the STRING directive.

***** **ERROR 034: END inside an unclosed block.** An END statement occurs within an IF block, a REPEAT block, or a MACRO definition block. This error may occur if the ENDM, ENDR, or ENDIF directives are either missing or misspelled.

***** **ERROR 035: Value truncated to byte.** The value entered exceeds one byte (allowable range -128 to +255). The value is truncated to fall within this range.

***** **ERROR 036: Invalid character follows label.** A label has not been followed by a space or tab character.

***** **ERROR 037: Extra operands ignored.** One or more extra operands appear in a statement. The statement is assembled and the extra operands are ignored. This error may occur if a statement is miscoded, if an invalid delimiter is used for an operand list, or if a semicolon does not precede a comment. This error may also occur if an invalid character occurs within a symbol (for example, AB%C).

***** **ERROR 038: String variable used as label.** A string variable is present in the label field of a statement other than a SET directive. The label field is ignored.

***** **ERROR 039: Invalid operation code.** 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 operation is misspelled, if a macro invocation precedes its definition, or if an invalid delimiter follows a label.

***** **ERROR 040: Invalid character.** A character not in the assembler character set has been used outside of double quotes. Refer to the "Source Module Character Set" in the Tables section of this manual.

***** **ERROR 041: Syntax error.** A statement does not conform to the required syntax. Refer to the Language Elements section of this manual for the correct syntax.

***** **ERROR 042: Invalid option or separator.** An invalid delimiter has been used between listing control options in the LIST or NOLIST directive operand field. Spaces are not valid delimiters. This error may occur if spaces are used in place of commas to delimit the options, or if an invalid listing control option is used.

***** **ERROR 043: No label on EQU or SET.** An EQU or SET directive has a missing or invalid label field.

***** **ERROR 044: Invalid macro name.** A macro name is missing or invalid. The macro body is ignored. This error may occur if the macro name is already defined or if an invalid delimiter is used before the macro name.

***** **ERROR 045: Invalid relocation option.** An invalid relocation option has been used in a section directive. (Valid options are: PAGE, INPAGE, and ABSOLUTE.) The assembler ignores the invalid option and assumes the section to be byte-relocatable. This error may occur if the option is misspelled or is not preceded by a comma.

***** **ERROR 046: MACRO within a macro.** A MACRO directive occurs within a macro definition block. The MACRO directive is ignored.

***** **ERROR 047: Invalid except in Macro.** An EXITM, ENDM, REPEAT, or ENDR directive appears outside of a macro definition block.

***** **ERROR 048: Invalid operand.** The specified operand is either incomplete or inaccurate for the BYTE, WORD, ASCII, BLOCK, ORG, or TITLE directives.

***** **ERROR 049: Address assigned to string.** An attempt has been made to assign an address value to a string variable.

***** **ERROR 050: Section definition Phase error.** The specified section relocation option differs between pass 1 and pass 2. This error may occur if a SET directive has a forward reference.

***** **ERROR 051: Section definition Phase error.** The specified section is defined during the second, but not the first, pass of the assembler. This error may occur if a SET directive has a forward reference.

***** **ERROR 052: Too many Sections or Globals.** The number of declared sections and other global symbols exceeds 254 during the processing of a SECTION directive. The current section declaration is not accepted by the assembler.

***** **ERROR 053: Invalid relocation option.** An ABSOLUTE relocation option has been specified within a RESERVE directive operand field.

***** **ERROR 054: Negative RESERVE length.** The RESERVE operand is either negative or greater than 32767.

***** **ERROR 055: Invalid section name.** An invalid symbol has been declared as a SECTION, COMMON, or RESERVE name. This error may occur if a section name is misspelled, contains invalid characters, or is a previously defined label or reserved word.

***** **ERROR 056: Invalid RESERVE length.** The required RESERVE expression is either specified incorrectly, specified without a comma before the expression, or missing from the RESERVE directive.

***** **ERROR 057: RESUME section undefined.** 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 use invalid characters.

***** **ERROR 058: RESUME of RESERVE section.** A RESUME directive has been used with a RESERVE section name.

***** **ERROR 059: Resumed section invalid.** A resumed section has been defined after the number of declared sections and other global symbols exceeded 254. The section being resumed is discarded.

***** **ERROR 060: Global operand already defined.** A global symbol has been defined more than once. See the GLOBAL directive in the Assembler Directives section for correct usage of global symbols.

***** **ERROR 061: GLOBAL declaration Phase error.** A global symbol has been used before it is defined. This message may be caused by a SET directive with a forward reference.

***** **ERROR 062: Too many Sections and Globals.** The number of declared sections and other global symbols exceeded 254 during the processing of a GLOBAL directive. The current global declaration is not accepted by the assembler.

***** **ERROR 063: Invalid radix.** An invalid radix follows a constant. The assembler recognizes hexadecimal (H), octal (O or Q), and binary (B) constants, and defaults to decimal when no radix is specified.

***** **ERROR 064: Invalid digit.** An invalid digit is associated with a specified radix. For example, the binary number 10031B contains an invalid digit (3).

***** **ERROR 065: Unmatched string or parameter delimiter.** An opening quotation mark or bracket is not matched by a closing quotation mark or bracket.

***** **ERROR 066: Line too long after replacement.** The expanded line (containing single quotes used as replacement indicators) is too long. Only 127 characters are allowed.

***** **ERROR 067: Extra replacement identifier.** One or more characters follow the replacement identifier (an item enclosed in single quotes) within a macro definition block. For example, '#bug' generates this error.

***** **ERROR 068: Replacement invalid outside of macro.** Replacement identifiers (# and @) are used outside of a macro definition block.

***** **ERROR 069: Undefined replacement string.** A symbol in single quotes (' ') is not yet defined as a string.

***** **ERROR 070: Invalid replacement identifier.** An invalid symbol or symbols have been used for the replacement specification. For example, '???' is invalid.

***** **ERROR 071: Scalar value required.** An address value has been used where a scalar is required.

***** **ERROR 072: Invalid expression.** The expression is either invalid or incomplete for the specified operation.

***** **ERROR 073: Section size Phase error.** The number of bytes generated for this section during the first pass is not the same as the number of bytes generated during the second pass. This error may occur if a SET directive is used with a forward reference.

***** **ERROR 074: Undefined symbol.** No value has yet been assigned to a symbol used in an expression.

***** **ERROR 075: String truncated.** More characters are assigned to a string than allowed by its definition.

***** **ERROR 076: Negative SEG operand.** The parameter of the SEG function is either negative or greater than 32767.

***** **ERROR 077: SEG starting operand is zero.** The starting position parameter of the SEG function is zero.

***** **ERROR 078: Insufficient workspace.** An internal work area of the assembler is full. This error may occur if string functions or conditional assembly leave insufficient memory to perform the required functions.

***** **ERROR 079: Value too large.** The operand of the SPACE directive exceeds 255.

***** **ERROR 080: Invalid NAME symbol.** The NAME symbol is invalid because it does not begin with a letter.

***** **ERROR 081: Illegally substituted ENDM.** An ENDM directive within a macro expansion precedes the normal macro ending.

***** **ERROR 082: Nested INCLUDE directive.** The source code inserted into the program with an INCLUDE directive contains another INCLUDE directive. See the Assembler Directives section for information on the INCLUDE directive.

***** **ERROR 083: Missing ENDIF**. The ENDIF directive is missing from a conditional IF block.

***** **ERROR 084: Missing ENDM for included macro.** The ENDM directive is missing from a macro definition block.

***** **ERROR 085: String value too large**. The length of a string used as a number exceeds two characters.

***** **ERROR 086: Shift count exceeds 16.** An attempt has been made (using SHL or SHR) to shift more than 16 bit positions in one operation.

***** **ERROR 087: Too many symbols.** The assembler symbol table is filled. This is a fatal error; assembly is aborted. This error occurs when too many symbols have been used. The source module must be divided into smaller pieces to permit assembly.

***** **ERROR 088: Invalid transfer label.** The label used for a transfer address on an END directive is not an address defined in the current source module.

***** **ERROR 090: ENDOF function applied to a bound global.** An ENDOF function has been used with a bound global instead of a section.

***** **ERROR 091: Unable to assign INCLUDE file.** DOS/50 is unable to access an INCLUDE file. This occurs when an illegal filespec is specified. (For example, INCLUDE "LPT" specifies a reserved device.) This error message is preceded by an SRB status code indicating the reason that the specified file cannot be accessed. Refer to the Error Codes section of the 8550 System Users Manual for individual descriptions of the SRB status codes.

***** **ERROR 092: Illegal operation on a global**. An attempt has been made to assign a value to a global symbol with the SET directive.

Section 14

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 16-bit address has a value in the range 0 to 65535 (FFFF hexadecimal).

Assembler. The 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. For example, the decimal number 29 may be written as 11101B.

Bound Global. See **Global**.

Brief Name. A temporarily defined shorthand name for a file, used as an alternative to a complete filespec. The BRIEF command defines brief names.

Byte-Relocatable. See **Relocatable**.

Carriage Return. See **Return**.

Code. To translate a sequence of operations into a series of statements in a programming language. The statements of a program are called **source code**. The machine instructions produced by assembling source code are called **object code**.

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.

Command File Invocation. A method of invoking the linker or library generator.

```
LINK @comfile or LIBGEN @comfile
```

invokes the linker or library generator and specifies that commands are to be read from the designated command file rather than from the system terminal.

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.

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 many 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 in quotes (for example, "TEXT", "P.O. Box 500", "").

Converter. A system program that translates an assembly language program written for another assembler into a format suitable for processing by the Assembler.

Current Directory. The directory that contains the file(s) you are currently using. A filespec that does not begin with a slash specifies either a standard device (such as CONI or REMO) or a file in the current directory. The operating system USER command selects a new current directory.

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 ASCII, BLOCK, BYTE, or WORD directive.

Default. 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.

Directory. A file that may contain only pointers to other files. A file that is not a directory is called a **data file**. A file that is pointed to by a directory is said to reside **in** the directory; every file resides in at least one directory. Likewise, a directory is said to **contain** each file it points to. The operating system CREATE command creates a new directory.

DOS/50. The Disc Operating System of the 8550 Microcomputer Development Lab.

End Address. The address of the last byte in a section.

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.

Filespec. A sequence of names, separated by slashes, that defines a path to a file. A file that is pointed to by the current directory may be specified with a single name.

Formal Parameter. See **Parameter**.

Forward Reference. Use of a symbol that is not defined until later 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 referred to in another. A **bound** global is defined in the current module. An **unbound** global is undefined in the current module; its value must be supplied by another module or by the linker command DEFINE.

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 and begins with a decimal digit to distinguish it from a symbol. For example, the decimal number 29 may be written as 1DH. The decimal number 15 may be written as 0FH (but not 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 into the corresponding machine instruction.

Interactive Invocation. A method of invoking the linker or library generator. When you enter the LINK command without parameters, or the LIBGEN command without specifying a command file, you must enter further linker or library generator commands from the system terminal.

Label. A symbol that represents an address, variable, or constant in an assembly language program.

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). The system program used to create and maintain libraries of object modules.

Linker. The 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 8080A mnemonic MOV represents a machine instruction that moves a value into a register.

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 of the same type. (2) To include a subexpression within an expression.

Null String. An empty character string without quotes: nothing.

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 LOAD command. 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 the suffix letter O or Q. For example, the decimal number 29 may be written as 35O 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 assembly language statement LDA NUM1 performs a load operation on the byte addressed by the operand NUM1.

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 microcomputer development lab 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 **absolute** section must start at a specified address.

Reserved Word. A predefined symbol that has a special meaning to the assembler and may not be used as a label, section name, or module name. Reserved words include mnemonics, register names, and assembler directives and functions.

Return. The RETURN character (ASCII code 13), also called CR or carriage return. This character marks the end of a command or an assembly language statement.

Scalar. A 16-bit signed numeric value not used as an address. A scalar takes a value in the range -32768 to +32767.

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.

Simple Invocation. A method of invoking the linker in which all actions to be taken by the linker are specified in the LINK command line.

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.

Start Address. The address of the base, or first byte, of a section.

String. A sequence of ASCII characters. A string enclosed in quotes (for example, "ELEPHANT") is called a **string constant**.

Symbol. A string of one to eight 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, sections, or modules.

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 TRANSFER.

Unbound Global. See **Global**.

Variable. In an assembly language program, a value that may be altered during assembly. The SET directive creates or redefines a variable.

Section 15
INDEX