64000

HP64000 Logic Development System

Assembler/Linker Reference Manual

CERTIFICATION

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

WARRANTY

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service. Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country.

HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

LIMITATION OF WARRANTY

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environmental specifications for the product, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

EXCLUSIVE REMEDIES

THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES. HP SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

ASSISTANCE

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

CW&A 9/79

Tab Index

Printing History

Each new edition of this manual incorporates all material updated since the previous edition. Manual change sheets are issued between editions, allowing you to correct or insert information in the current edition.

The part number on the back cover changes only when each new edition is published. Minor corrections or additions may be made as the manual is reprinted between editions.

First Printing	January 1980	(Part Number 64980-90990)
Second Edition	June 1980	(Part Number 64980-90992)
Third Edition	November 1980	(Part Number 64980-90994)
Reprinted	July 1981	
Fourth Edition	December 1981	(Part Number 64980-90997)
Reprinted	March 1982	
Reprinted	July 1982	
Reprinted	July 1983	

Assembler/Linker Reference Manual

Table of Contents

Table of Contents (Cont'd)

Table of Contents (Cont'd)

List of Tables

List of Illustrations

Chapter 1

How to Use the HP Assembler

General

The assembler processes the source program modules and produces an output that consists of a source program listing, a relocatable object file, and a symbol cross-reference list. Errors detected by the assembler will be noted in the output listing as error messages. Refer to Appendix D for a listing of all error codes and their definitions.

NOTE

Refer to Chapter 2 in the Overview Manual for BOOT-UP operations and SOFTWARE UPDATING PROCEDURE from a tape cartridge.

Input/Output Files

Source Input File

Input to the assembler is a source file that is created through the editor. It consists of the following:

Example	Description
"8080"	- Assembler directive.
Source Code	- Source statements consisting of:
.	Assembler Pseudos - refer to
.	Chapter 3
.	
.	Microprocessor Instructions -
	refer to the Assembler
	Supplement Manual

Assembler Output Files

The assembler produces relocatable object modules that are stored under the same name as the source file but in a format that can be processed by the linker. If an object file does not exist at assembly time, the assembler will create one. If an object file does exist, the assembler will replace it.

List File. The list file is a formatted file that is output to a line printer. It can also be stored in another file or applied to the system CRT display. The list can include:

a. Source statements with object code.

b. Error messages.

c. Summary of errors with a description list.

d. Symbol cross-reference list.

Symbol Cross-reference List. All symbols are cross-referenced except local macro labels and parameters. A cross-reference listing contains:

a. Alphabetical list of program symbols.

b. Line numbers where symbols are defined.

c. All references (by line numbers) to symbols in the program.

Entering a Source Program

Once a source program has been developed, it can be entered into the HP Model 64000 by way of the system editor. The first line of the source program must be the assembler directive statement since it tells the assembler what type of assembly source follows in the file. This first line of the source program is also used to set options that control the assembler output listing. The assembler directive format is:

"processor" options

Example:

"8080" XREF EXPAND

NOTE

Options may be listed in either upper or lower case characters.

The list options that may be selected in the assembler directive statement are list/nolist, expand, nocode, and xref. A brief description of each option follows:

nolist - no listing, except for error messages. All LIST pseudo instructions in the source program are ignored.

list - listing of source program with no macro or data expansions. All NOLIST pseudo instructions in the source program are ignored.

expand - listing of all source and macro generated codes. All LIST pseudo instructions in the source program are ignored.

nocode - the nocode option suppresses the generation of object code.

xref - the xref option activates the symbol cross-reference feature of the assembler.

NOTE

If an invalid option is assigned, the assembler will indicate the error within the directive statement as follows:

"8080" REFX<–invalid EXPAND

This type of directive error is not counted with the source program errors detected by the assembler.

The output listing can be controlled to limit the number of lines appearing on each page of the output. The following rules apply.

1. The format is: LIST <decimal number>.

2. The instruction must be embedded in the program; but cannot be included on the directive line. The instruction will not be accepted by the assembler if it is entered from the keyboard.

3. The effective limits for <decimal number> are 5 thru 127. If a number less than five is used, the first page of output will have six lines, and succeeding pages will have five lines.

Pseudo instructions LIST, NOLIST, and EXPAND may be assigned in the body of the source program (refer to Chapter 3). However, if the assembler directive statement specifies any list option, that option will override all embedded list instructions.

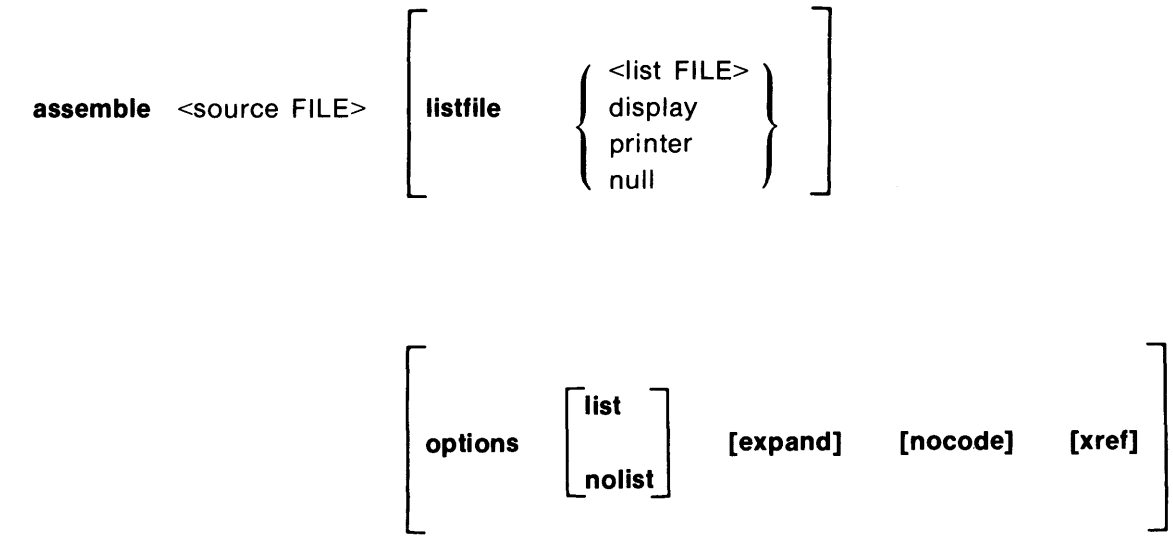
Assembling the Program

Once a source program module has been entered into the system by way of the editor, it can be assembled using the assemble function of the system. A syntax description follows for assembler activation.

NOTE

Refer to table 1-4 for syntax conventions.

SYNTAX



where:

<source FILE> - name of the file containing the source program.

listfile - soft key used to specify a destination for output listing other than the system default list file.

<list FILE> - name of the file where the assembler output listing will be stored. If the file does not exist, a new file will be created using the name assigned.

display - designates the system CRT as the output listing destination.

printer - designates the system line printer as the output listing destination.

null - specifies that no listing is to be generated.

options - soft key used to specify type of output listing.

DEFAULT VALUES

listfile: Assembler output listing defaults to the device specified by the userid statement. If the userid statement does not specify an output location, the assembler defaults to the null listing function.

options: If no entry is made following the options soft key prompt, the output listing default will be as follows:

a. Output listing of source program with object codes and error messages.

b. There will be no expansion of macros and multiple-byte pseudo instructions.

c. There will be no listing of the symbol cross-reference list.

EXAMPLES:

a. **assemble** SAM

Assembles source file SAM; output listing to listfile default.

b. **assemble** SAM **listfile** CHARLEY

Assembles source file SAM; output listing to file name CHARLEY.

c. **assemble** SAM **listfile** display options nolist nocode

Assembles source file SAM; only error codes will be listed on the CRT display; no object code will be generated.

FUNCTION

The assembler translates source program inputs into relocatable object modules that may be linked and loaded into the system. Absolute addresses are assigned by the linker.

How to Use the Assembler

NOTE

In the following paragraphs, the soft key prompts are indicated as follows:

[name]

The name listed in the soft key symbol indicates the soft key prompt or the soft key that is to be pressed.

Program Assembly

To assemble a source program and list the assembler output to a disc file or some other device, proceed as follows:

a. Ensure that the following soft key prompts are displayed on the system CRT:

[edit] [compile] [assemble] [link] [emulate] [prom_prog] [<run>] [---ETC---]

b. Press [assemble] soft key. The soft key configuration will change to:

[<FILE>] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

c. Your next prompt is FILE. Type in the name of the source program to be assembled.

d. The soft key configuration will change to:

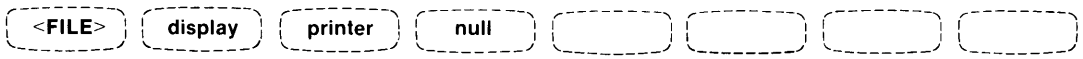
[ ] [listfile] [options] [ ] [ ] [ ] [ ] [ ]

e. The user now has the opportunity to select an output listing device or the listfile default device (see SYNTAX description block). If the listfile default device is selected, press the [RETURN] key and proceed to step k.

NOTE

If the required output listing device is the device specified by the listfile default but the output listing options need revised, press the [options] soft key and preceed to step i.

If another output listing device is desired, press the [listfile] soft key. The soft key configuration will change to:



f. Route the assembler output listing to the desired location by pressing the [display] soft key, or the [printer] soft key.

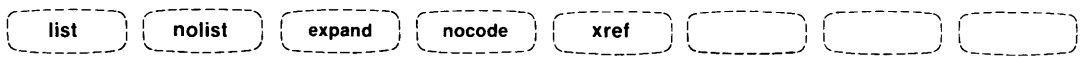
NOTE

Pressing the [null] soft key results in no output listing. Error messages will be displayed on the system CRT.

g. If the output listing is to be stored in another file, type in the name of the file to accomplish step f.

h. The soft key configuration will change to:

i. The user now has the opportunity to select new list options or the option default condition (see SYNTAX description block). If the option default condition is selected, press the [RETURN] key and proceed to step k. If the output listing options are to be changed, press the [options] soft key. The soft key configuration will change to:



j. Press any of the soft keys to change the output listing for this assembly run (more than one soft key may be pressed starting with the left-most key).

k. After accomplishing step j, press the [RETURN] key to assemble and list the source program.

Output Listing

An example of an assembler output listing is given in table 1-2, using the source program example listed in table 1-1. To illustrate an assembler output listing that contains error messages refer to table 1-3.

NOTE

Tables 1-1, 1-2, and 1-3 use the target system program for their examples. The file names have been changed slightly to prevent program duplication.

Table 1-1. Source Program Example

```
'8080' list xref
                EXT     DSPL,KYBD
                ORG     0B00H
EXEC            LXI     H,0C00H
                SPHL
                LXI     H,0805H
                MVI     A,03H
LP1             MOV     M,A
                DCR     L
                JNZ     LP1
                MVI     B,06H
LP2             CALL    KYBD
                JC      LIGHT
                PUSH    PSW
                DCR     B
                JP      LP2
                MVI     B,-01H
                LXI     H,0805H
                LXI     D,0804H
GO              LDAX    D
                MOV     M,A
                DCR     L
                DCR     E
                JNZ     GO
                POP     PSW
                MOV     M,A
                JMP     LIGHT
                POP     PSW
                MOV     M,A
                DCR     L
LIGHT           CALL    DSPL
                JMP     LP1
                END
```

Table 1-2. Assembler Output Listing

```
FILE: EXCT:                                    HEWLETT-PACKARD:
                                               INTEL 8080 ASSEMBLER

LINE   LOC     CODE  ADDR          SOURCE STATEMENT

  1                          '8080' list xref
  2                          EXT       DSPL,KYBD
  3    0B00                  ORG       0B00H
  4    0B00    21    0C00    EXEC      LXI       H,0C00H
  5    0B03    F9              LP1     SPHL
  6    0B04    21    0805              LXI       M,A
  7    0B07    3E    03                MVI       A,03H
  8    0B09    77              LP1     MOV       M,A
  9    0B0A    2D                      DCR       L
 10    0B0B    C2    0B09              JNZ       LPI
 11    0B0E    06    06                MVI       B,06H
 12    0B10    CD    0000    LP2       CALL      KYBD
 13    0B13    DA    0B32              JC        LIGHT
 14    0B16    F5                      PUSH      PSW
 15    0B17    05                      DCR       B
 16    0B18    F2    0B10              JP        LP2
 17    0B1B    06    FF                MVI       B,-01H
 18    0B1D    21    0805              LXI       H,0805H
 19    0B20    11    0804              LX!       D,0804H
 20    0B23    1A              GO      LDAX      D
 21    0B24    77                      MOV       M,A
 22    0B25    2D                      DCR       L
 23    0B26    1D                      DCR       E
 24    0B27    C2    0B23              JNZ       GO
 25    0B2A    F1                      POP       PSW
 26    0B2B    77                      MOV       M,A
 27    0B2C    C3    0B32              JMP       LIGHT
 28    0B26    F1                      POP       PSW
 29    0B30    77                      MOV       M,A
 30    0B31    2D                      DCR       L
 31    0B32    CD    0000    LIGHT     CALL      DSPL
 32    0B35    C3    0B09              JMP       LP1
 33                                    END

Errors=       0
```

Table 1-2. Assembler Output Listing (Cont'd)

FILE: EXCT: CROSS REFERENCE TABLE

LINE#	SYMBOL	TYPE	REFERENCES
2	DSPL	E	31
4	EXEC	A	
20	GO	A	24
2	KYBD	E	12
31	LIGHT	A	13,27
8	LP1	A	10,32
12	LP2	A	16

NOTE: In the cross-reference table, the letter listed under the TYPE column has the following definition:

A = Absolute
C = Common (COMN)
D = Data (DATA)
E = External
M = Multiple Defined
P = Program (PROG)
R = Predefined Register
S = Special
U = Undefined

Table 1-3. Assembler Output Listing with Errors

```
FILE:  EXCT:                                    HEWLETT-PACKARD:
                                                INTEL 8080 ASSEMBLER

LINE   LOC     CODE  ADDR        '8080' list xref   SOURCE STATEMENT
1                                                
2                                          EXT      DSPL,KYBD
3      0B00                                ORG      0B00H
4      0B00    21    0C00        EXEC     LXI      H,0C00H
5      0B03    F9                         SPHL
6      0B04    21    0805                 LXI      M,A
7                                          MVU      A,03H
ERROR-UO                                   ^
8      0B07    77                 LP1     MOV      M,A
9      0B08    2D                         DCR      L
10     0B09    C2    0B07                 JNZ      LPI
11     0B0C    06    06                   MVI      B,06H
12     0B0E    CD    0000        LP2      CALL     KYBD
13     0B11    DA    0B2F                 JC       LIGHT
14     0B14    F5                         PUSH     PSW
15     0B15    05                         DCR      B
16     0B16    F2    0B0E                 JP       LP2
17     0B19    06    FF                   MVI      B,-01H
18     0B1B    21    0805                 LXI      H,0805H
19     0B1E    11    0804                 LXI      D.0804H
20     0B21    1A                 GO      LDAX     D
21     0B22    77                         MOV      M,A
22     0B23    2D                         DCR      L
23     0B24    1D                         DCR      E
24     0B25    C2    0B21                 JNZ      GO
25                                         POO      PSW
ERROR-UO, see line 7                       ^
26     0B28    77                         MOV      M,A
27     0B29    C3    0B2F                 JMP      LIGHT
28     0B2C    F1                         POP      PSW
29     0B2D    77                         MOV      M,A
30     0B2E    2D                         DCR      L
31     0B2F    CD    0000        LIGHT    CALL     DSPL
32     0B32    C3    0B07                 JMP      LP1
33                                         END
Errors=2, previous error at line 25
UO-Unidentified Opcode, Opcode encountered is not defined for this microprocessor
```

Table 1-3. Assembler Output Listing with Errors (Cont'd)

FILE:	EXCT:		CROSS REFERENCE TABLE
LINE#	SYMBOL	TYPE	REFERENCES
2	DSPL	E	31
4	EXEC	A	
20	GO	A	24
2	KYBD	E	12
31	LIGHT	A	13,27
8	LP1	A	10,32
12	LP2	A	16

NOTE: Error messages are inserted immediately following the statement where the error occurs. All error messages (after the first error message) will contain a pointer to the statement where the last error occurred. At the end of the source program listing, an error summary statement will be printed. The summary will contain a statement as to the total number of errors noted, along with a line reference to the previous error. It will also define all error codes listed in the source program listing.

Refer to Appendix D for a listing of all error codes.

Table 1-4. Syntax Conventions

[ ]	Parameters enclosed in square brackets are optional. Several parameters stacked inside a set of brackets indicate an either/or situation. You may select any one or none of the parameters. The use of square brackets implies that a default value exists. **Example:** [A B] This indicates A or B may be selected.
< >	Angle brackets denote a syntactical variable. A syntactical variable is a defined parameter that you supply. **Example:** < FILE > This example says FILE is a variable that is supplied by you.
{ }	Braces specify that the parameter enclosed is required information. When several parameters are stacked within a set of braces, you must select one and only one of the parameters. **Example:** { A B C } This example says one and only one of A, B, or C must be selected.

Table 1-4. Syntax Conventions (Cont'd)

[ ] [ ]	Stacked square brackets indicate that enclosed parameters are optional and may be selected in any single occurrence, any combination, or may be omitted. **Example:** [A] [B] [C] A and/or B and/or C may be selected, or this option may be omitted.
=>	Arrow indicates - "is defined as"
...	An ellipsis indicates a previous bracketed element can be repeated.
lower-case bold type	Key words (commands) are always lower-case on the System 64000. These key words will always be represented in text with lower-case bold type. **Example:** **edit** <FILE>
UPPER-CASE PARAMETERS	Literal information which are parameters of a command are represented in text with upper-case type. Literal information parameters are information that you enter as shown in text. An exception to this is any parameter enclosed with angle brackets, < >, (e.g. <FILE> is a syntactical variable, not literal information.
Symbols in color	Syntax symbols that are in color indicate that they are used for definition purposes and do not appear on the CRT display.

Chapter 2

HP Model 64000 Assembler Rules and Conventions

Introduction

The HP Model 64000 Assembler recognizes three types of source statements: microprocessor instructions, assembler pseudo opcodes, and macro definitions and calls. This chapter describes the coding rules and conventions that must be followed when using the assembler.

Source Statement Format Rules

Each microprocessor instruction, assembler pseudo opcode, or macro call is divided into four fields: the label field, the operation field, the operand field, and the comment field. The format rules to be followed when constructing a line of source program are:

a. Field sequence cannot be changed. The correct order of field sequence is:

Example:

Label	Operation	Operand	Comment
SAVE	EQU	SAM	;SAVE EQUATES ;TO SAM

NOTE

It is recommended that each field in the source statement start at a fixed position (column) in the source line. This type of format may be constructed using the tab setting capabilities of the system editor to define each field's starting position. The presentation of the program listing in a fixed format improves readability.

b. One or more spaces (blanks) must separate the fields in a source statement.

c. A label field, if used, must begin in column 1 of the source statement. If column 1 is blank, the assembler assumes that the label field is omitted.

Additional rules and conventions governing the source statement fields are given in the following paragraphs.

Statement Length

A source statement may contain up to 110 characters (including spaces), and is terminated by a RETURN. A statement containing more than 110 characters will be truncated to 110 characters.

Blank lines will not affect the object modules and may be used to improve readability of the source program listing.

Label Field

Labels may be used in all microprocessor instructions, some assembler pseudo opcodes, and macro calls. Since the label assigned identifies that particular statement, and since this label may be used as a reference point by other statements in the program, every label must be unique within each source program.

NOTE

Specific symbols are predefined and cannot be used as labels. The symbols that are predefined will depend upon the microprocessor being supported. Refer to the Assembler Supplement Manual for a list of predefined symbols.

The label field starts in column 1 of the source statement and must be terminated by a space or a colon (:).

NOTE

A colon (:) cannot be used to terminate a macro label. Refer to Chapter 4 for construction of Macros.

A label may contain any number of characters. The first character must be an upper case alphabetic character. The remaining characters may be either alphabetic or numeric. The alphanumeric character set includes the letters of the alphabet (upper and lower case), underline symbol (_), and the numeric digits Ø through 9.

Valid Symbols:

Ab_cd
AB_CD
A5rHi

Invalid Symbols:

ab.cd?
$BCDEF
4UVWXY

If more than fifteen characters are entered in the label field, the assembler will print all characters in the output listing; however, it will use the first 15 characters only for label identification. Therefore, the assembler will recognize:

STATEMENTLABELA1

and

STATEMENTLABELA2

as being identical and will issue a duplicate-symbol error message.

The only statements requiring labels are macro definitions and EQU pseudo instructions. For all other statements, assignment of a label is optional.

Operation Field

The operation field contains a mnemonic code for a microprocessor instruction, an assembler pseudo opcode (see Chapter 3), or a macro call (see Chapter 4). The opcode specifies the operation or function to be performed. The operation field follows the label field and is separated from it by at least one space, a tab, or colon (:). If there is no label, the opcode may begin in any column position after column 1.

The operation field is terminated by one or more spaces, by a tab, by a carriage return, or by a semicolon (;) indicating the start of the comment field.

Assembler pseudo and control statements provide the following capabilities:

a. Assembler control

b. Object program linkage

c. Address and Symbol definitions

d. Constant definition

e. Assembly listing control

f. Storage allocation

If a label is specified and the operation field does not contain a microprocessor instruction, an assembler pseudo opcode, or a macro call, the label will be assigned to the current program counter location.

Operand Field

The operand field specifies values or locations required by the microprocessor instruction, assembler pseudo opcode, or macro call. The microprocessor uses various modes of addressing for obtaining the operands and saving the results of the execution.

The addressing mode will be determined by the mnemonic instruction and the information in the operand field. The operand field, if present, follows the operation field and must be separated from it by at least one space.

An operand may contain an expression consisting of a single symbolic term, a single numeric term, or a combination of symbolic terms and numeric terms, enclosed in parentheses, and joined by the expression operators +, –, *, and /.

The types of information that are permitted in the operand field are summarized in Assembler Supplement Manual. Each instruction determines the operand type and their proper sequence.

Comment Field

The optional comment field may contain any information that the user deems necessary to identify portions of the program. The delimiter for the comment field is the semicolon (;), a tab, or a space following the operand field. A semicolon in any column of the source statement will starts the comment field (except when used in an ASCII string). In situations where more than one line of programming is needed for the comment field, an asterisk (*) in column 1 of a source statement indicates that the information following is part of a comment field and should not be acted on as if it were part of the program.

Delimiters

Certain characters are used to indicate the end of fields or labels, and the beginning of others. These characters, referred to as delimiters, should not be used as ordinary characters. For example, a space cannot be used as part of a label name. A list of delimiters follows:

Delimiter	Use
space	Separates fields or operands; ends a label
tab	Separates fields; ends a label
Semicolon (;)	Indicates start of comment field
Asterisk (*)	When used in column 1 of source statement, indicates that comment field follows
Colon (:)	Indicates end of label field
Parentheses ((...))	Used in expression for precedence
Apostrophes ('...')	Indicates a character string
Quotation Marks ("...")	Indicates a character string
Ampersand (&)	Indicates macro parameters
Double Ampersand (&&)	Index macro parameters

Symbolic Terms

A symbol used in the operand field must be a symbol that has been defined in the program, such as a symbol in the label field, a machine instruction, or a symbol in the label field of an EQU pseudo instruction (must be defined prior to referencing).

A symbol may be either absolute or relocatable and depends on the type of assembly selected. The assembler will assign a value to a symbol when it is encountered in a label field of a source statement. If the program is to be loaded in absolute form, the values assigned by the assembler remain fixed. If the program is to be relocated, the actual value of a symbol will be established by the linker (refer to Chapter 5).

A symbolic term may be preceded by a plus (+) or minus (−) sign. If preceded by a plus (+) sign or no sign, the symbol refers to its associated value. If preceded by a minus (−) sign, the symbol refers to the 2's complement of its associated binary value.

Program Counter ($)

The program counter symbol ($) is a symbolic term used to indicate the current value of the program counter.

Numeric Terms

A numeric term may be binary, octal, decimal, or hexadecimal. A binary term must have the suffix "B" (for example: 1Ø11Ø1B). Octal values must have either an "O" or a "Q" suffix (for example: 26O, 26Q). A hexadecimal term must have the suffix "H" (for example: ØBBH, 2CDH, 36H). When no suffix is assigned, the decimal value is assumed.

NOTE

It is necessary to start a hexadecimal term with a decimal digit since the assembler will identify a term that starts with an alphabetic character as a label or an expression.

String Constants

Besides numeric and symbolic constants, an operation may contain string constants. String constants are produced by using ASCII (American Standard Code for Information Interchange) characters. String constants, combined with other symbols and constants, are written by enclosing ASCII characters within quotation marks (".....''), apostrophe marks ('.....'), or carets (^.....^).

The numeric value of a string is defined as follows:

a. A null string (" ") (' ') or (^ ^) has a numerical value of zero.

b. A 16-bit value of a one-character string is one whose high-order nine bits are zeros and whose low-order seven bits contain the ASCII code for the character (refer to Appendix B for ASCII character conversion table).

Example:

'C' = "C" =

ØØØØØØØØB = ØØH= High order byte

Ø1ØØØØ11B = 43H= Low order byte

c. A 16-bit value of a two-character string is the 16-bit value where the ASCII code for the first character is the high-order byte and the ASCII code for the second character is the low-order byte.

Example:

'AB' = "AB" =

A = Ø1ØØØØØ1B = 41H= High order byte

B = Ø1ØØØØ1ØB = 42H= Low order byte

NOTE

The MASK pseudo instruction allows the user to alter ASCII strings. Refer to the MASK pseudo description given in Chapter 3.

---

d. For a string longer than two characters, the value of the string will be the last two characters.

Example:

'ABCDE' = "ABCDE" =

D= Ø1ØØØ1ØØB = 44H= High order byte

E= Ø1ØØØ1Ø1B = 45H= Low order byte

Expression Operators

The assembler contains two groups of operators that permit the following operations: arithmetic and relational comparison.

Arithmetic Operators. The arithmetic operators are:

Operator	Interpretation
+	Addition
–	Subtraction
*	Multiplication
/	Division

Examples:

The following expressions generate the bit pattern for ASCII character W (Ø1Ø1Ø111B):

1+28*2

1+(–28*–2)

1+(84/3)*2

Logical Operators. Logical operators are used to form logical expressions and a logical expression may be used any place that an expression can legally be used. The logical operators are as follows:

Operator	Interpretation
.AN.	Logical AND
.NT.	Logical one's complement
.OR.	Logical OR
.SL.	Shift left
.SR.	Shift right

Examples:

```
SAM.SL.1
.NT.CHAR
SAM.OR.CHARLIE
```

Operator Precedence. The operators have a precedence to define which operator is evaluated first in an expression. The operators are listed below in a descending order of precedence.

Parenthesis ( ) override all precedence.

```
.NT.
.SL.,.SR.
.OR.,.AN.
*,/
+,-
```

Relational Comparison (Macros Only). When the assembler processes an ".IF" instruction, the logical expression in the operand field is evaluated. The relational operators are:

Operator	Interpretation
.EQ.	equal
.NE.	not equal
.LT.	less than
.GT.	greater than
.LE.	less than or equal
.GE.	greater than or equal

Relocatable Expressions

Three program counters are provided for identifying areas of relocatable code. The three areas are identified as data (DATA), program (PROG), and common (COMN) and can be changed from one relocatable area to another by using these assembler pseudo codes (refer to Chapter 3 for more detail). Some rules governing relocatable expressions are given in the following paragraphs.

The value of a relocatable term will be assigned during the linking process. The assigned value will depend upon:

a. The relocatable areas (PROG, DATA, or COMN) to which it is assigned,

and

b. Where the area is located in memory during the link operation.

It should be remembered that expressions may be formed from absolute and relocatable terms using arithmetic operators and parentheses. The expression resulting from this type of operation must be either absolute or one of the three relocatable types.

An absolute term is an expression whose value is not dependent upon the location of the program module in memory. The following rules apply to the formation of absolute expressions:

a. Each absolute term or constant is an absolute expression.

b. If AD and BD are relocatable symbols in the same relocatable area, then (AD–BD) is an absolute expression. This is so because the difference between AD and BD remains constant regardless of the relocation factor of the program. That is, if the program is relocated, the values of AD and BD are offset by the same amount.

c. If A2 and B2 are absolute symbols, then:

(A2+B2)
(A2*B2)
(A2–B2)
and (A2/B2)

are absolute expressions.

A relocatable term is an expression whose value is undefined at link time. The following rules apply to the formation of relocatable expressions:

a. Any relocatable term is a relocatable expression.

b. If DA is an absolute expression and DR is a relocatable expression, then:

(DA+DR)
(DR+DA)
and (DR-DA)

are relocatable expressions and are the only relationship permitted. That is, an absolute expression may be subtracted from a relocatable expression but not vice versa.

2

Certain relocatable terms are invalid and will generate error messages. A few examples of invalid relocatable terms are as follows:

a. Two relocatable symbols - same area (PROG, DATA, or COMN). If DA and DB are two relocatable symbols, then:

(DA+DB)
(DA*DB)
and (DA/DB)

are invalid expressions because the assembler does not know where these symbols are stored in memory.

b. Two relocatable symbols - different areas (PROG, DATA, or COMN). If DA and DB are two relocatable symbols, then:

(DA+DB)
(DA-DB)
and (DA*DB)

are invalid expressions because, again, the assembler does not know where these symbols are stored in memory.

c. Relocatable symbols in different areas (PROG, DATA, or COMN) can be combined if the expression results in one relocatable type. For example, if relocatable symbols DA and DB are PROG type and relocatable symbol DC is DATA type, the expression:

(DA+DC-DB)

is valid since (DA-DB) is an absolute offset to DC.

Chapter 3

Assembler Pseudo and Control Instructions

Introduction

This chapter describes the HP Model 64000 assembler pseudo instructions. The pseudo instructions are used for listing control, program counter, linkage control, and constant definitions.

An assembler pseudo may be either an instruction to the assembler or a request for some special service. Most pseudos require no memory space because, unlike microprocessor instructions, they produce no object code.

Table 3-1 is supplied to help you quickly locate the description of a specific pseudo instruction.

Table 3-1. Pseudo Instruction Index

Table 3-1. Pseudo Instruction Index (Cont'd)

8-Bit Microprocessors

The label field of each numerical constant instruction listed above is the address of the first byte of data. The value of the constant is an 8-bit number for the binary, decimal, hexadecimal, and octal instructions. For the ASCII instruction, each character in the string expression represents one byte of data.

16-Bit Microprocessors

The label field of each numerical constant instruction listed above is the address of the first word (two bytes) of data. The value of the constant is a 16-bit number for binary, decimal, hexadecimal, and octal instructions. For the ASCII instruction, two characters will be put into each 16-bit word (high and low bytes). If an odd number of characters exist in the string then the assembler will pad the last word with ASCII spaces.

Pseudo Instruction Syntax

The following paragraphs list and define each assembler and control instruction in detail. They are listed alphabetically. Once familiar with the instructions, use Appendix C, 'Assembler Pseudo Instructions Summary', as a quick-reference guide when constructing program modules.

ASCII Constant

SYNTAX:

Label	Operation	Operand	Comment
[symbol]	ASC	string expression	
	or		
[symbol]	ASCII	string expression	

The ASC pseudo instruction allows the user to store ASCII text in memory using quotation marks or apostrophes as delimiters. The first delimiter must be used as the terminating delimiter.

The ASCII character(s) specified in the operand field may be in the form of a string expression.

Example:

	Label	Operation	Operand	Comment
a.		ASC	"XYZ"	
b.		ASCII	"THE EAGLE'S BEAK"	
		or		
c.		ASCII	'G. H. "BABE" RUTH'	

Binary Constant

SYNTAX:

Label	Operation	Operand	Comment
[symbol]	BIN	binary number	
	or		
[symbol]	BINARY	binary number	

The BIN pseudo instruction allows the user to store data in binary format in memory.

The number(s) specified in the operand field is (are) written in binary format. If more than one operand is specified, each one must be separated from the other by a comma.

Example:

	Label	Operation	Operand	Comment
a.		BIN	1Ø1	
b.		BINARY	1Ø1,1Ø11Ø1ØØ	
c.	SAM	BIN	1Ø1,1Ø11,1Ø11Ø1ØØ	

Designated Memory Storage Area

SYNTAX:

Label	Operation	Operand	Comment
	COMN		
	or		
	DATA		
	or		
	PROG		

Three program counters are used to identify areas of relocatable code. The areas are designated as data (DATA), program (PROG), and common (COMN). You can change from one relocatable area to another by the use of these pseudo instructions.

The PROG and DATA instructions function identically and are merely two names that identify two separate, relocatable memory areas. Common (COMN) allows construction of a common block of data that is used by different program modules. The default area is PROG.

Example:

Operation	Operand
DATA	
.	
.	
.	
.	
PROG	
.	
.	
.	
DATA	
.	
.	

Normally, the default memory area (PROG) will be used when constructing a source program. The DATA memory area might occupy another part of memory and can be used for storing data, tables, instructions, etc.

The COMN pseudo can be used to group information that is common to a number of program modules. Assigning these type of items to a specific area in memory facilitates modification and referencing.

NOTE

All information assigned to the COMN area in memory must be grouped in one program file. If two or more files assign information to the COMN area, the linker will overlay the first data stored with the second block of data assigned, thereby erasing the first block of data.

Refer to Chapter 2 for rules and conventions covering construction of relocatable expressions. Refer to Chapter 5 for more details concerning relocatable areas in memory.

Decimal Constant

SYNTAX:

Label	Operation	Operand	Comment
[symbol]	DECIMAL	decimal number	

The DECIMAL pseudo instruction allows the user to store data in decimal format in memory.

The number(s) specified in the operand field is (are) written in decimal format. If more than one operand is specified, each one must be separated from the other by a comma.

Example:

	Label	Operation	Operand	Comment
a.		DECIMAL	153	
b.		DECIMAL	1Ø,2Ø,3Ø	
c.	SAM	DECIMAL	1ØØØ	

NOTE

The DECIMAL pseudo instruction may be replaced with the DEC pseudo if it does not conflict with the microprocessor's mnemonic instruction set.

Program Module Termination

SYNTAX:

Label	Operation	Operand	Comment
	END	[expression]	

The END instruction terminates the logical end of a program module. It is optional. If it is omitted, the program will be automatically terminated after the last statement in the program module being edited.

The optional expression in the operand field represents the starting address in memory for program execution. This address initializes the program counter when the file is loaded during emulation.

Example:

Operation	Operand
END	
or	
END	1000H
or	
END	(symbol)

Equate

SYNTAX:

Label	Operation	Operand	Comment
symbol	EQU	expression	

The EQU instruction is used to establish a relationship between a symbol and an expression. The symbol in the label field acquires the same value as the expression in the operand field. Redefinition of the symbol is not permitted.

If the operand field of an EQU instruction contains another symbol, it must be defined previously in the source program.

Example:

Label	Operation	Operand
SAM	EQU	3

This statement assigns an absolute decimal value of 3 to symbol SAM.

The EQU instruction may also be used to equate symbols of certain relocatable types and add an offset to an external.

Example:

Label	Operation	Operand
	EXT	TABLE
INDEX	EQU	TABLE+4

EXPAND

Listing of Macro Expansions

SYNTAX:

Label	Operation	Operand	Comment
	EXPAND		

The EXPAND instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will generate, within the output listing, all macro and data expansions that follow it.

You may exit the EXPAND output listing mode by embedding the LIST directive in the proper location within the source program.

EXT

External

SYNTAX:

Label	Operation	Operand	Comment
	EXT	SYMBOL1,SYMBOL2	
	or		
	EXTERNAL	SYMBOL1,SYMBOL2	

Symbols used in one program module, but defined in another program module, must be declared external with an EXT or EXTERNAL statement. After assembling the source program, the linker will connect identical symbols.

GLB

Global

SYNTAX:

Label	Operation	Operand	Comment
	GLB	SYMBOL1,SYMBOL2	
	or		
	GLOBAL	SYMBOL1,SYMBOL2	

Symbols that are defined in one program module and referenced by other program modules must be declared global in the program module where they are defined.

HEX

Hexadecimal Constant

SYNTAX:

Label	Operation	Operand	Comment
[symbol]	HEX	hexadecimal number	

The HEX pseudo instruction allows the user to store data in hexadecimal format in memory.

The number(s) specified in the operand field is (are) written in hexadecimal format. If more than one operand is specified, each one must be separated from the other by a comma.

Example:

	Label	Operation	Operand	Comment
a.		HEX	FF	
b.		HEX	A,FE,Ø5	
c.	SAM	HEX	B,5F,7,81	

SYNTAX:

Label	Operation	Operand	Comment
	IF	<absolute expression>	
		.	
		.	
		.	
	ELSE		
		.	
		.	
		.	
	IFEND or ENDIF		
or			
	IF	<absolute expression>	
		.	
		.	
		.	
	IFEND or ENDIF		

The IF pseudo instruction allows sections of code to be conditionally assembled. Sections of code are assembled or skipped based on an absolute expression. This expression is treated as a Boolean function with either a true or false value.

The IF instruction evaluates an absolute expression as a logical function with the value zero false and a nonzero value true. When the expression evaluates to a nonzero (true) condition, the code following the IF instruction is assembled until an ELSE or IFEND or ENDIF instruction is encountered. If the expression evaluates to zero (false), then the ELSE part of the IF instruction is assembled until an IFEND or ENDIF is found. The expression type must be absolute (type = 0) and all symbolic references must be defined before being used with an IF instruction. Only the lower 16 bits of the expression value are used to determine the true or false condition. The IFEND or ENDIF instructions are used to terminate the IF instruction. They must either follow the ELSE instruction or the IF instruction if no ELSE portion is desired.

Conditional IF instructions can be nested up to 20 levels deep. If the nesting levels exceed 20, then an IO (invalid operand) error will be flagged on the IF instruction. If an error is flagged on an ELSE or IFEND or ENDIF instruction, then a nesting level error has occurred. One of these three instructions was encountered before an IF instruction or more IFEND or ENDIF instructions were found than IF instructions. The end of the assembly source is treated as an IFEND or ENDIF instruction and no error is flagged if the assembler is currently in an IF instruction.

Example:

Label	Operation	Operand	Comment
TRUE	EQU	-1	
FALSE	EQU	0	
SAM	EQU	TRUE	
	IF	SAM	
		.	;This code will be assembled.
	ELSE	.	
		.	;This code will not be assembled.
	ENDIF	.	

NOTE

This instruction is only available with the 1802, 6800 series, 6805/6809, 8080/8085, 8086/8088, 68000, Z8, Z80, Z8000, and user definable assemblers.

3

INCLUDE

Include Secondary File in Source Input

SYNTAX:

Operation	Operand
INCLUDE	<FILENAME> [:<USERID>] [:<DISC#>]

The INCLUDE pseudo instruction allows a secondary file to be included in the source input stream. Only one level of inclusion is allowed. Nested INCLUDE files will result in an error message.

Example:

Operation	Operand
INCLUDE	SAM:ID:0

List

SYNTAX:

Label	Operation	Operand	Comment
	LIST		

The LIST instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will generate one line of output for each line of source code that follows it.

The output listing can be controlled so that a desired number of lines per output page can be achieved. Refer to the section in chapter 1 dealing with list options for details.

NOTE

All LIST instructions embedded in the source program will be overridden if any list option is specified in the assembler directive statement (refer to Chapter 1 for assembler directive statement definition).

Set Mask

SYNTAX:

Label	Operation	Operand	Comment
	MASK	(AND),(OR)	

The MASK instruction permits masking of ASCII strings. The instruction affects ASCII strings only and will produce a logical 'AND' operation with each ASCII character followed by a logical 'OR' operation. (The OR operand is optional.)

Example:

Operation	Operand
MASK	77H,1Ø1B
or	
MASK	77H

The default condition of a MASK directive is:

AND = FFH

OR = Ø

NAME

Name

SYNTAX:

Label	Operation	Operand	Comment
	NAME	"SALPHA"	;character string

The NAME instruction is used to add comments to the object module for reference on the load map listing. The name string may contain any combination of characters, numbers, or special characters but is limited to a maximum of 22 characters.

NOLIST

No Output Listing

SYNTAX:

Label	Operation	Operand	Comment
	NOLIST		

The NOLIST instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will suppress the output listing of all source statements following it. If used in the assembler directive statement, it will suppress all output listings except error messages.

Octal Constant

SYNTAX:

Label	Operation	Operand	Comment
[symbol]	OCT	octal number	
	or		
[symbol]	OCTAL	octal number	

The OCT pseudo instruction allows the user to store data in octal format in memory.

The number(s) specified in the operand field is (are) written in octal format. If more than one operand is specified, each one must be separated from the other by a comma.

Example:

	Label	Operation	Operand	Comment
a.		OCT	37	
b.		OCTAL	37,24,71	
c.	SAM	OCT	77	

Origin

SYNTAX:

Label	Operation	Operand	Comment
	ORG	address	

The ORG instruction is used for absolute programming. It sets the contents of the location counter to the address entered in the operand field. The next statement, following the ORG instruction, will be located at the address specified.

NOTE

The ORG instruction cannot be used to alter the relocatable area counters associated with the DATA, PROG, and COMN instructions. The relocatable area instructions do not contain operands and their associated counters start at zero and are initialized at linking time.

Example:

Operation	Operand
ORG	ØB111H

The object code of the source statement following the ORG instruction will begin at location B111H. When using the ORG directive care should be taken to ensure that the assigned memory location will not result in memory overlap during the link operation.

A label symbol is generally not used in the operand field of this instruction; however, if a symbol is entered it must be defined in a label field of a prior statement in the source program and must be an absolute expression.

Real

Convert real decimal to binary floating point

SYNTAX:

Label	Operation	Operand	Comment
	REAL	real decimal number	

The REAL instruction converts real decimal numbers to IEEE binary floating point constants. Short (32 bit) or long (64 bit) IEEE binary floating point values can be generated by the REAL instruction.

A real decimal number must start with a decimal digit(s), followed by a decimal point, and end with a decimal digit(s). Powers of 10 are added after the last decimal digit with an "E" or "L" qualifier. Real decimal numbers specified with an E qualifier or with no qualifier are converted to short real binary floating point (32 bits). The L qualifier indicates a long real number.

Numbers are converted to the IEEE standard for real numbers and stored high to low; where, the highest byte (containing the sign bit) is stored at the lowest address value and the lowest byte is stored at the highest address.

Examples:

Label	Operation	Operand	Comment
	REAL	1.0	;Short real, decimal value = 1.
	REAL	1.0E2	;Short real, decimal value = 100.
	REAL	1.0E-2	;Short real, decimal value = .01.
	REAL	-1.0E-2	;Short real, decimal value = -.01.
	REAL	1.0L2	;Long real, decimal value = 100.

REPT

Repeat

SYNTAX:

Label	Operation	Operand	Comment
	REPT	number	

The REPT instruction is used to repeat the next source statement any given number of times.

Example:

	Operation	Operand
	REPT	5

will repeat the next source statement five times.

SET

Define Symbol

SYNTAX:

Label	Operation	Operand	Comment
symbol	SET	expression	

The SET pseudo instruction allows a symbol to be defined and assigned a value. It is similar to the EQU pseudo, except with SET the value can be changed during the assembly process. The expression used must be absolute (type = 0) and all symbolic references must be defined before they are used.

Examples:

Label	Operation	Operand	Comment
SAM	SET	0	
SAM	SET	SAM+1	

NOTE

This instruction is only available with the 1802, 6800 series, 6805/6809, 8080/8085, 8086/8088, 68000, Z8, *Z80, *Z8000, and user definable assemblers.

*With these microprocessors, the pseudo instruction is called DEFL.

SKIP

Skip

SYNTAX:

Label	Operation	Operand	Comment
	SKIP		

The line of output listing that follows a SKIP instruction will be placed at the top of the next page, following the page heading.

The SKIP instruction is not printed in the program listing.

SPC

Line Space

SYNTAX:

Label	Operation	Operand	Comment
	SPC	[number]	

Whenever a SPC instruction is encountered in the source program, the assembler will space downward (line feed) a specified number of lines.

The number of line feeds required is indicated in the operand field. If the operand field is left blank, the assembler will generate one blank line.

The SPC instruction is printed in the output listing only if an error exists in the operand field.

Title

SYNTAX:

Label	Operation	Operand	Comment
	TITLE	"Name"	

The TITLE instruction will initiate a page eject and create a "Name" line at the top of each page listing for the source program that follows. The title may be 70 characters in length and may be changed any number of times during the program.

Example:

Operation	Operand
TITLE	"This is the Title"

This statement, if inserted as the second statement in the source program (directly after the assembler directive), will cause the title to be printed on the first page listing of the source program and on the top of each page thereafter. Alternatively, if the TITLE instruction is inserted in the program at some place other than the second statement of the source program, the instruction will initiate a page eject and the new title will be printed at the top of the new page and each page thereafter.

Warning No Warning

SYNTAX:

Label	Operation	Operand	Comment
	WARN or NOWARN		

The NOWARN instruction turns off the warning message in the source listing. The WARN instruction restores it.

Example:

```
                    Operation

                    NOWARN
                    EXT SAM
                    WARN
                    NOP
                    EXT SAM
                        ^
WARN_DS
```

Chapter 4
Macros

Introduction

This chapter discusses the use of macro directives and their construction. Using macro definitions eliminates the repetitious writing of the same sequence of instruction during source program construction.

Any legitimate sequence of instructions may be incorporated into a macro. This process is called "macro definition." Once defined, a single macro call may be used at any point in the source program to insert the sequence of instructions that was defined by the macro definition. The insertion of the sequence of instruction is referred to as "macro expansion."

Advantages of Using Macros

A macro definition provides a means of producing, at program assembly time, a commonly used sequence of assembler statements as many times as needed. The sequence of statements is specified just once; then, at any point in the program where these statements are to be produced, a single macro call will cause the sequence to be generated. Using macros wisely will serve to:

a. Simplify the coding of programs.

b. Significantly reduce the number of programming errors caused by rewriting similar instructions throughout the program.

c. Ensure that common functions are performed by standard routines.

d. Improve program readability.

e. Reduce duplication of effort among the several programmers assigned to the project.

Disadvantages of Using Macros

One problem with macros is that variables used in a macro are only known within it - they are local rather than global. This can create confusion without any benefits in return. Other disadvantages of macros are:

a. Repetition of the same macro may create many instructions.

b. Possible effects on registers and flags that may not be clearly stated.

Macros vs Subroutines

In some situations, a subroutine, rather than multiple in-line macro statements, can reduce overall program size. However, subroutines require branching, then returning, from another part of the program. This usually increases the program execution time. In addition, the variables in a subroutine are evaluated only during program execution while macro parameters are evaluated at assembly time.

Macro Format

A macro definition consists of three parts that must appear in the order given below:

a. Header statement

b. Source statement body

c. Trailer statement

The header statement specifies both the name of the new macro instruction and the formal arguments (parameters) that will be used in the macro instruction. The general macro header syntax is as follows:

Name **MACRO** [optional parameters]

The name of the macro definition is written in the label field of the source statement and must not be terminated by a colon(:). To avoid multiple-label conflicts, the assembler treats labels within macros as local labels, applying only to that particular macro. MACRO is written in the operation field of the source statement. The optional parameters follow in the operand field of the source statement.

The body of a macro definition defines the action of the macro instruction. There is no limit to the number of instructions that may appear. The fields within the macro body are the same as those of an assembler instruction, and the rules for forming a macro statement are about the same as the rules for forming an assembler instruction.

The trailer statement consists of a single line. The operation field of the line contains the word MEND (macro end).

An example of a macro instruction is as follows:

Label	**Operation**	**Operand**	**Comment**
SAVE	MACRO		
	OPC	CHARLEY	
	OPC	SAVEA	
	OPC	SAM	
	OPC	SAVEB	
	MEND		

NOTE

The opcode symbol (OPC) listed in the operation field above will take the form of a mnemonic instruction for the specific microprocessor being programmed.

To call the SAVE macro, insert the macro name in the operation field of the source statement and the code in the body of the macro will be generated in the program as if it had been typed there. The generated instructions will be printed in the listing of the program (only if the **expand** list option is specified).

Example:

SAVE	
OPC	CHARLEY
OPC	SAVEA
OPC	SAM
OPC	SAVEB

Optional Parameters

The formal parameters of a macro definition are often referred to as symbolic variables. Macro symbolic parameters (as distinguished from ordinary labels or symbols) are those symbols that may be assigned different values by the programmer. When assembler instructions are generated according to the macro definition, the dummy parameters are replaced by the values that have been assigned to them. The three simple rules that must be followed when forming dummy parameters are:

a. The first character of the parameter must be an ampersand (&).

b. The second character of the parameter must be an alphabetic letter. All remaining characters, if any, can be either letters of the alphabet or numbers.

c. Any number or length of parameter may be entered in the operand field of a macro definition as long as the entire line does not exceed 110 characters (not including a carriage return). In addition, after arguments are substituted for parameters in a macro call, the lines resulting from the macro expansion must not exceed 110 characters. Otherwise, an error message is issued.

Symbolic parameters are used in the macro definition and are assigned values by the programmer in each macro call which references that particular macro. An example of the general syntax for optional parameters is as follows:

```
Label     Operation     Operand
ADDS      MACRO         &SUBNAM,&PARAM
          JMP           &SUBNAM
          DEF           &PARAM
          MEND
```

The programmer assigns parameters to his ADDS macro to develop:

```
          ADDS          ADD,SUM+27
          JMP           ADD
          DEF           SUM+27
```

A macro instruction may also be used for text replacement and concatenation of a parameter to generate a new word. For example consider the following macro instruction:

```
Label     Operation     Operand
SAVE      MACRO         &REG,&PARM1,&PARM2
          LD&REG        &PARM1
          ST&REG        &PARM2
          MEND
```

You may now call this simple macro instruction, assign your own parameters, and produce the following insert into your program:

```
SAVE        A,SAM,FRED
LDA         SAM
STA         FRED
```

Note the substitution of the actual parameters of the call A, SAM, FRED - for the dummy parameters in the macro heading (&REG, &PARM1, and &PARM2). Note further that the sequence of the call parameters interchange directly with the sequence of the dummy parameters.

It is important to remember that a macro does not necessarily produce the same source code each time it is called. Changing the parameters in a macro call will change the source code that the macro generates.

Unique Label Generation

The macro assembler generates unique local labels each time a macro is called by using four ampersand characters in a label (&&&&). When a macro is called, &&&& is replaced by four decimal digits. Note, this four-digit constant is incremented every time a macro is called, even if the ampersand characters are not in the macro label. With this labeling, a macro can be called more than once in a program (no duplication of label).

Example:

```
 1  "8080"
 2
 3  TEXT MACRO &STRING
 4
 5  L1_&&&& DB L2_&&&&-L1_&&&&-1                ;Length of string.
 6      ASC &STRING
 7  L2_&&&&
 8
 9      MEND
10
11      TEXT "STRING # 1"
 +
 +  L1_0001 DB L2_0001-L1_0001-1                ;Length of string.
 +      ASC "STRING # 1"
 +  L2_0001
 +
12
13      TEXT "STRING # 2"
 +
 +  L1_0002 DB L2_0002-L1_0002-1                ;Length of string.
 +      ASC "STRING # 2"
 +  L2_0002
 +
```

Conditional Assembly

There are four conditional assembly instructions available for use with the HP Model 64000 Assembler. When inserted among the statements in the body of a macro definition, they provide the means for instructing the assembler to branch and loop among the statements of the executable program. These conditional assembly instructions will not be printed in the listing of the program (unless they contain an error). Only their effects can be seen in the generated object code. The four conditional instructions are:

.SET
.IF
.GOTO
.NOP

.SET Instruction

The .SET instruction provides a way to assign or modify an expression value of a macro local. The instruction assigns the value of the operand field to the name specified in the label field. When the label is encountered subsequently in the macro program, the assembler substitutes its new value. This value remains unchanged until altered by a subsequent .SET instruction. The general format of a .SET instruction is as follows:

Label	**Operation**	**Operand**
name	.SET	expression

An example of a .SET instruction is as follows:

```
GENTABLE      MACRO       &COUNT
LOOP_COUNT    .SET        &COUNT
LOOP_TOP      .NOP
              DEF         1
              DEF         2
              DEF         3
LOOP_COUNT    .SET        LOOP_COUNT-1
              .IF         LOOP_COUNT .GT. 0 LOOP_TOP
              MEND
```

Call expansion:

```
GENTABLE    3
DEF         1
DEF         2
DEF         3
DEF         1
DEF         2
DEF         3
DEF         1
DEF         2
DEF         3
```

.IF Instruction

The .IF instruction is the conditional-branch instruction and uses six relational operators. These operators are:

.EQ. ===>	equal
.NE. ===>	not equal
.LT. ===>	less than
.GT. ===>	greater than
.LE. ===>	less than or equal
.GE. ===>	greater than or equal

NOTE

All comparisons are 32 bits unsigned.

An .IF instruction has the following format:

Operation	**Operand**
.IF	Exp .(Relational Operator). Exp Label

The .IF instruction directs the assembler to relationally compare two expressions. If the value of this comparison is true, a branch is taken to the statement named by the label symbol in the operand field. Otherwise, the statement immediately following the .IF instruction is processed by the assembler.

.GOTO Instruction

The .GOTO statement is the unconditional-branch instruction. It has the following format:

Operation	Operand
.GOTO	Label

The .GOTO instruction directs the assembler to branch, unconditionally, to the statement named by the label symbol in the operand field.

.NOP Instruction

A .NOP instruction is a no-operation instruction. This instruction is useful with .IF and .GOTO instructions when branching is required to sections of the program that are not labelled. The .NOP instruction format is as follows:

Label	Operation
LABEL	.NOP

When a branch is taken to a .NOP instruction, the effect is the same as if a branch were taken to the statement immediately following it.

NOTE

It is important to remember that conditional assembly instructions generate no source code and the sole function of the .SET, .IF, .GOTO, and .NOP instructions are to conditionally alter the sequence in which the assembler processes the source program or macro definition instructions.

An example using the .IF, .GOTO, and .NOP instructions is as follows:

```
CONDITION       MACRO       &P1,&P2,&P3
                .IF         &P1 .EQ. 1 LOAD
                .IF         &P1 .EQ. 2 STORE
                .GOTO       DONE
LOAD            .NOP
                OPC         &P2
                OPC         &P3
                .GOTO       DONE
STORE           .NOP
                OPC         &P3
                OPC         &P2
DONE            .NOP
                MEND
```


Some call expansion examples are as follows:

```
a.              CONDITION   1,SAM,BLUE
                OPC         SAM
                OPC         BLUE

b.              CONDITION   2,SAM,BLUE
                OPC         BLUE
                OPC         SAM

c.              CONDITION   Ø

                            <NO CODE>
```

Checking Parameters

When using macro calls, you may want to omit specific parameters defined in the macro definition. This is accomplished by using the null symbol ("") or a comma (,). For example:

Macro definition:

```
SAM         MACRO       &P1,&P2,&P3,&P4
```

Macro call:

```
            SAM         ,FRED,"",ØFCH
```

In the above example, &P2 is assigned a value of FRED and &P4 is assigned a value of FCH. The &P1 and &P3 parameters are omitted.

4

An example of a macro expansion is as follows:

```
CALLSUB     MACRO       &SUB,&P1,&P2,&P3
            JMP         &SUB
            .IF         &P1 .EQ. "" DONE
            DEF         &P1
            .IF         &P2 .EQ. "" DONE
            DEF         &P2
            .IF         &P3 .EQ. "" DONE
            DEF         &P3
DONE        .NOP
            MEND
```

Some expansion call examples are as follows:

a.
```
            CALLSUB     ADD,PARAM
            JMP         ADD
            DEF         PARAM
```

b.
```
            CALLSUB     ADD
            JMP         ADD
```

```
c.      CALLSUB     ADD,IN,OUT,RESULT
        JMP         ADD
        DEF         IN
        DEF         OUT
        DEF         RESULT
```

Indexing Parameters

The assembler has the ability, when instructed, to index through a parameter list to determine if all or certain parameters are present. This is accomplished by using a macro local symbol prefaced with two ampersands (&&). The following macro directive is presented as an example:

```
Label          Operation     Operand                  Comment

 1. CALLSUB    MACRO         &P1,&P2,&P3,&P4
 2.            JMP           &P1
 3. PARAM      .SET          2
 4. PARAM_LOOP. .NOP
 5.            .IF           &&PARAM .EQ. '' JUMP-OUT
 6.            DEF           &&PARAM
 7. PARAM      .SET          PARAM+1
 8.            .GOTO         PARAM_LOOP
 9. JUMP_OUT   .NOP
10.            MEND
```

A line-by-line explanation of the above macro definition is as follows:

Line 1. Defines the macro directive named CALLSUB with its dummy parameters &P1,&P2,&P3,&P4.

Line 2. A subroutine designated by parameter &P1 is accomplished.

Line 3. Name PARAM is set to a value of 2.

Line 4. A .NOP statement is assigned the name PARAM_LOOP.

Line 5. Since the PARAM label has been assigned the value 2 (see line 3), the .IF statement checks to see if the second parameter of the macro call statement has been omitted. If it has, the .IF statement causes the program to branch to the JUMP_OUT statement.

NOTE

During each iteration of the PARAM_LOOP, the value of PARAM is increased by 1 (see line 7). The iterations continue until the .IF statement is satisfied.

Line 6. Updates the value of PARAM to the current value assigned.

Line 7. Adds 1 to the current value of PARAM.

Line 8. Loops to PARAM_LOOP.

Line 9. A .NOP statement used to exit the PARAM_LOOP iteration.

Line 10. Macro end.

4

Some macro expansions of the previous macro example are as follows:

a.
```
CALLSUB     ADD
JMP         ADD
```

b.
```
CALLSUB     ADD,LOC1,LOC2
JMP         ADD
DEF         LOC1
DEF         LOC2
```

c.
```
CALLSUB     ADD,P1,P2,P3
JMP         ADD
DEF         P1
DEF         P2
DEF         P3
```

Chapter 5
Linker Instructions

Introduction

A system application program, referred to as the linker (link), combines relocatable object modules into one file, producing an absolute image that is stored by the Model 64000 for execution in an emulation system or for programming PROMS. Interaction between the user and the linker remains basically the same regardless of which microprocessor assembler is being supported.

To prepare object code modules for the Model 64000 load program, the linker performs two functions:

a. Relocation: allocates memory space for each relocatable module of the program and relocates operand addresses to correspond to the relocated code.

b. Linking: symbolically links relocatable modules.

The user may optionally select an output listing of the program load map and a cross-reference (xref) table. The linker also generates a listing that contains all errors that were noted. These error messages will contain a description of the error along with the file name and relocation/address information when applicable.

In addition to the above output listings, the linker constructs a global symbol file (link_sym type) and stores this file under the same file name assigned the absolute image/command file. This global file may be used for symbolic referencing during emulation. The link_sym file also contains the relocation addresses for all programs. This information is used to relocate asm_sym types during emulation. The assembler translates source program inputs into relocatable object modules that may be linked and loaded into the system. Absolute addresses are assigned by the linker.

Linker Requirements

The following information is required by the linker:

a. File names of all object files to be loaded.

b. File names of libraries to be searched to resolve any unsatisfied externals.

c. Relocation information (load addresses for all relocatable areas).

d. Listing and debugging options as follows:

   1) List (Load Map): file/program name, relocatable load addresses, and absolute load addresses.

   2) Xref: symbols, value, relocation, and defining and referencing modules.

e. File name for command/absolute image file.

Since the linking operation will usually be required each time there is a software change and the information in items a through e remain constant for any given application, the linking control information is automatically saved in a command file with the same name as the absolute image file. The command file is distinguished from the absolute image file by file type.

Using the Linker

The command line in which Model 64000 commands are entered is accessed by way of the development station keyboard. Each system application function (edit, compile, assemble, link, emulate, or prom_prog) can be called using keyboard soft keys. A syntax description follows.

SYNTAX

```
link [<FILE>] [listfile <list destination>]
     [options [edit][nolist][xref][no_overlap_check][comp_db]]
```

Default Values

<FILE>	If no linker command file is specified, the default allows creation of a new file of type link__com.
<list destination>	Defaults to user specified listfile default. See userid command.
options	If "options" is not entered, listing defaults to options specified in the linker command file. If options is specified, followed by nothing, a load map listing with no cross-reference is performed.

Examples:

link	Requests the linker to create a new linker command file. Listing output will go to the listfile default.
link PROGABS	Links absolute file PROGABS containing files in linker command file PROGABS. Listing output will go to listfile default and options in PROGABS type link__com are in effect
link PROGABS options edit	Request the linker for purpose of viewing or modifying PROGABS:link__com. Listing output will go to listfile default.

FUNCTION

The linker combines and relocates specified relocatable files creating an absolute file with the same name as that of the link__com file which can be used to program a PROM (with prom programmer option) or to load emulation RAM to be executed and analyzed with the emulator.

Parameters:

<FILE>	A file of type link__com to be used to direct the linker as to relocatable and relocation addresses.
<list destination>	File or device to which listing output is sent.
options	Allows user to override options specified in the linker command file.
nolist	Overrides the list option specified in the linker command file and suppresses output of a load map.
xref	Overrides no xref option specified in the linker command file and forces output of a global symbol cross-reference table.
edit	Allows user to edit existing link__com file specified.
no__overlap__check	Overrides overlap__check option specified in the linker command file and suppresses errors caused by memory overlaps. Default condition for linker overlap__check is ON.
comp__db	This file is created by the linker when requested and is a data base containing information from all of the comp__sym files associated with relocatables in an absolute file.

DESCRIPTION

The linker may be called by one of two methods: simple calling or interactive calling.

The simple calling method is used when interaction with an established command file is not required. That is, the current information in the command file is valid and no changes are required.

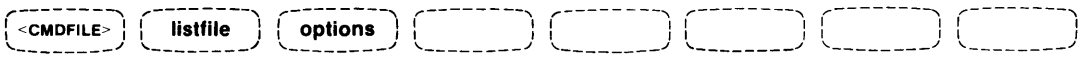
The interactive calling method is used when building a new linker command file or when the information in the current command file needs revision.

How to Use the Linker

Simple Calling Method

a. Ensure that the following soft key prompts are displayed on the system CRT:

b. Press the link soft key. The soft key configuration will be:



c. The next prompt is CMDFILE. Type in the name of the established command file to be linked. The soft key configuration will change to:

d. If it is necessary to change the output listing destination, press the listfile soft key. The soft key configuration will change to:

e. Route the linker output listing to the desired location by selecting the FILE option, or by pressing the display soft key, the printer soft key, or the null soft key.

NOTE

Pressing the null soft key results in no output listing. Error messages will be displayed on the system CRT.

f. If the FILE option is desired in step e, type in the file name under which the listing is to be stored. You can then review your output listing on the system CRT using the edit function and your assigned file name.

g. The soft key configuration will change to:

h. Refer to the "options" default description in the LINK SYNTAX definition block.

i. If the [options] soft key is not used, the linker defaults to the list options specified in the command file and to noedit. To override the command file list options (for this link only), press the [options] soft key. The soft key configuration will change to:

[edit] [nolist] [xref] [ ] [ ] [ ] [ ] [ ]

If only the [options] soft key is used, the linker defaults to list, noxref, and noedit. Any of these defaults may be changed by pressing the appropriate soft key.

j. After accomplishing step i, press the [RETURN] key.

The linker will link the relocatable modules and produce the desired output listing.

Interactive Calling Method

The interactive calling method allows the user to create a new linker command file or edit an existing linker command file.

a. Ensure that the following soft key prompts are displayed on the system CRT:

[edit] [compile] [assemble] [link] [emulate] [prom_prog] [<run>] [---ETC---]

b. Press the [link] soft key. The soft key configuration will change to:

[<CMDFILE>] [listfile] [options] [ ] [ ] [ ] [ ] [ ]

c. The user may start creating a new linker command file by not specifing any command file. An existing command file may be modified by specifying the command file name and the edit option.

NOTE

In the following paragraphs, the procedures are written for establishing a new command file. If an existing command file is being edited, just type in the changes required after each query. If no changes are required for a particular query, proceed to the next query. In all instances, to proceed to the next query, press the [RETURN] key.

d. The command query displayed in the command line on the system CRT is:

Object files? file1,file2,.....,filen

This query asks for the names of the files to be linked and relocated. Type in the names of the files and then proceed to the next query.

NOTE

The soft key configuration 'prompts' will change with each query from the linker. The soft key 'prompts' indicate the type of information that is required.

Object files that are listed after the "Object files?" query may contain relocatable object modules, no-load files, and previously linked linker symbol files (for global symbol references).

No-load files are differentiated from normal relocatable files by enclosing the no-load files in parentheses. Linker symbol files are specified by including the file type ':link-sym' in the file name.

Example:

FILE1,(FILE2,FILE3),FILE4:link-sym

NOTE

Refer to the paragraphs in this chapter that discuss no-load and link-sym files for additional information.

---

e. The next command query displayed in the command line on the system CRT is:

Library files? lib1,lib2,.....,lib3

Interrogation for library files is the same as for object files. After all object files have been linked, the linker determines if any external symbols remain undefined. The linker then searches the library files for object modules that define these symbols. The linker relocates and links only those relocatable modules that satisfy external references. Since a library file may contain more than one object module, all of its relocatable modules may not be linked. Refer to the paragraph in this chapter that discusses libraries and their construction.

NOTE

No-load files or linker symbol files, used for global referencing, must not be listed after this query. The no-load and linksym files can only be referenced during the "Object files?" query.

After typing in the list of reference library files (or if library files are not referenced in the program), proceed to the next query.

f. The next command query displayed in the command line on the system CRT is:

Load addresses:PROG,DATA,COMN= addr,addr,addr

This query allows selection of separate, relocatable memory areas for the different modules of the program. For example, if you type in the following addresses:

Load addresses:PROG,DATA,COMN= 1000H,2000H,3000H

the linker will relocate the PROG file module to memory location starting at address 1000H, the DATA module will be relocated to memory location starting at address 2000H, and the COMN module will be relocated to memory location starting at address 3000H.

NOTE

Load addresses may be entered using any number base (binary, octal, decimal, or hexadecimal); however, the addresses listed in the load map are given in hexadecimal only.

The default addresses are zeros. After entering the load addresses or if the default addresses are acceptable, proceed to the next query.

g. The next command query displayed in the command line on the system CRT is:

More files? no

The linker asks if more files are to be linked. If the response is yes, the linker begins interrogation again, allowing additional object and library files to be specified with new load addresses. When specifying new relocatable areas, the user may continue with the previously relocatable area by typing "CONT" in the appropriate field (or using the CONT soft key). The relocatable area is treated as if no new address was assigned.

Example:

Load addresses:PROG,DATA,COMN=0BCCH,CONT,3FFCH

The default condition to the "more files?" query is no. Proceed to the next query.

h. The next command query displayed in the command line on the system CRT concerns output listing options. It has the following syntax:

List,xref= on off

The linker asks you to specify what output listings are required. Using the [on] or [off] soft key, select, in the sequence indicated in the syntax statement (list,xref), the desired output listings. After inserting the requirements, proceed to the next query.

NOTE

The output listings indicated after the list,xref=query are the command file values that will be used during this and future operations. They can be overridden by using the [options] soft key during the linker call.

The default condition for this query is on, off.

i. The next command query displayed in the command line on the system CRT is:

Absolute file name=name

This final query from the linker allows you to assign a name to the new command/absolute image file that you are about to link. The absolute image file that is created by the linker is always associated with a link command file of the same name. A global symbol file is also established under the name of the command/absolute image file name. The global symbol file contains all global symbols and their relocation values.

After entering the absolute file name, press the [RETURN] key.

The linker will link, relocate the files, and save the linking information in the command file.

Linker Output

The linker listings may be output to the system display, line printer, or any file. The following information may be included in the linker output listing:

a. List (Load Map)

b. Cross-reference table

c. Error messages

NOTE

Certain error messages contain more than 80 characters and will not be completely displayed on the system CRT. However, complete error messages will be printed when using the line printer or a list file for listings.

List (Load Map)

A load map is a listing of the memory areas allocated to each relocatable file. The listing begins with the first file linked and proceeds to list all other linked files with their allocated memory locations. An example of a load map listing that will be printed on the system printer is as follows:

FILE/PROG NAME	PROGRAM	DATA	COMMON	ABSOLUTE	DATE	TIME	COMMENTS
KYBD:SAVE	0000				Thu, 5 Jun 1980	11:37	
EXCT:SAVE				0B00-0B34	Thu, 5 Jun 1980	10:38	
DSPL:SAVE		A100			Thu, 5 Jun 1980	11:38	
next address	0021	A121					
REG1:SAVE	B000				Thu, 5 Jun 1980	11:52	
REG2:SAVE	B103				Thu,5 Jun 1980	11:53	
REG3:SAVE	B206				Thr, 5 Jun 1980	11:58	
next address	B30C						
Libraries							
PARAMETER:SAVE	0021				Thu, 5 Jun 1980	11:43	
MULTEQUAT:SAVE	0221				Thu, 5 Jun 1980	11:45	
next address	0421	A121					

XFER address=0B00Defined by EXCT
No. of passes through libraries=1
absolute & link_com file name=SETAG1:SAVE
Total# of bytes loaded=0782

A brief description of each column in the listing is as follows:

a. FILE/PROG NAME - this column will contain the name of the files that are linked. In the event library files are referenced, not only will the master library file be listed, but its subsections that are referenced will also be listed beneath the library file name. The subsections will be indented to indicate that they are part of the main library file. No-load files will be displayed in parentheses (...).

b. PROGRAM - this column will indicate the first address (hexadecimal) of a memory block that contains the PROG relocatable code in the file listed in the FILE/PROG NAME column.

c. DATA - this column will indicate the first address (hexadecimal) of a memory block that contains the DATA relocatable code in the file listed in the FILE/PROG NAME column.

d. COMMON - this column will indicate the first address (hexadecimal) of a memory block that contains the COMN relocatable code in the file listed in the FILE/PROG NAME column.

e. ABSOLUTE - this column will indicate the hexadecimal addresses of a memory block that contains the absolute code assigned by the file listed in the FILE/PROG NAME column.

NOTE

The "next address" statement in the load map listing indicates the next available hexadecimal address in the PROG, DATA, or COMN memory areas. It may also be used to determine the number of bytes (words for 16-bit processors) that are contained in each area (next address–starting address=total bytes).

f. DATE - this column will indicate the date that the file listed in the FILE/PROG NAME column was assembled (assuming the system date/time clock was current).

g. TIME - this column will indicate the time that the file listed in the FILE/PROG NAME column was assembled (assuming the system date/time clock was current).

h. COMMENTS - this column will contain user comments entered during assembly by the assembler pseudo NAME instruction.

Cross-reference Table

The cross-reference table lists all global symbols, the relocatable object modules that define them, and the relocatable modules that reference them. An example of a cross-reference listing that will be listed on the system printer is as follows:

SYMBOL	R	VALUE	DEF BY	REFERENCES
DSPL6	P	ØØ34	PGM68D	PGM68E
KYBD6	P	ØØØ1	PGM68K	PGM68E

A brief description of each column in the cross-reference listing is as follows:

a. SYMBOL - all global symbols will be listed in this column.

b. R (Relocation) - in this column a letter will identify the type of program module. The letters that are available and their definitions are:

A = Absolute
C = Common (COMN)
D = Data (DATA)
P = Program (PROG)
U = Undefined

c. VALUE - relocated address of the symbol.

d. DEF BY - this column will contain the file name that defines the global symbol.

e. REFERENCES - this column will list the file names that reference the global symbol.

5

"No-Load" Files

Files that are enclosed in parentheses in the "Object files?" query indicates to the linker that no code is to be generated for the file. Relocation and linking occurs in the same manner as if the file was a load file; however, the absolute image file generated by the linker does not contain the object code for the no-load file. No-load files may be useful in linking to existing ROM code or in the design of software systems requiring memory overlays.

Linker Symbol File

The linker creates a global symbol file for every link operation. The global file name is the same as the assigned command/absolute image file name assigned to the link. The user may find that linking to a common piece of code (global) is simplified by referring to that code by its linker symbol file. This is accomplished by referencing the correct linker symbol file name during the "Object files?" query by the linker. The linker symbol file name referenced at the time of the query must be specified by type ':link-sym'.

Object files? PGM68K,PGM68D:link-sym

Library Files

Libraries are a collection of relocatable modules that are stored on the system disc and may be referenced by the linker.

If a library file name is given as a response to the "Object files?" query, all the relocatable modules in the library file will be relocated and linked. If a library file name is given as a response to the "library files?" query, only those relocatable modules that define the unsatisfied externals will be relocated and linked. The remaining relocatable modules in the library file are ignored.

It is possible to combine relocatables into a library by using the system library command. Refer to the System Overview Manual for a detailed description of the library command.

Error Messages

When an error is detected during the link process, the linker will determine if the error is fatal or nonfatal. If the error is classified as fatal, the linker will abort the linking process. If the error is nonfatal the linker will continue the linking process, but will generate error messages that will be listed in the output listing. A description of each error message is given in the following paragraphs.

Fatal Error Messages

Upon encountering a fatal error the linker will display one of the following messages on the system CRT STATUS line. The linker will abort the link process and return control of the system to the monitor.

a. **Out of Memory in Pass 1.**

   The linker will issue this message to indicate that there is insufficient memory to accommodate the current operation. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

NOTE

As a general rule, the available memory space can handle programs containing approximately 3000 symbols. However, if cross-reference symbol tables are required, the symbol handling capability is reduced to approximately 1500 symbols.

---

b. **Out of Memory in Pass 2.**

The linker will issue this message to indicate that there is insufficient memory to accommodate the current operation. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

c. **Out of Memory in Xref.**

The linker will issue this message to indicate that there is insufficient memory to accommodate the building of a cross-reference table. This error does not affect the absolute file since it is created and stored prior to the linker attempting to build the cross-reference file. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

d. **Target Processors Disagree.**

The linker will issue this message if the relocatable modules to be linked are designed for different processors. Ensure that all relocatable modules assigned for linking are written for the same type microprocessor.

e. **Checksum Error.**

The linker will issue this message if it is unable to read a relocatable file due to a checksum error or other irregularities in the file. To correct this situation, reassemble the relocatable file; then, relink.

f. **Linker System Error.**

The linker will issue this message if it detects a hardware or software failure in the Model 64000. To correct this situation relink the relocatable modules or run the hardware performance verification program.

g. **File Manager Errors.**

The linker will issue certain messages if the system file manager is unable to perform the specified file operation as requested by the linker. Refer to the System Overview Manual for a list of File Manager Errors.

Nonfatal Error Messages

Upon encountering nonfatal errors, the linker will continue the link operation and print the error messages (except initialization errors) in the output listing. An error message that is listed will contain a description of the error and the name of the file where the error occurred. If the null list file is in effect, the linker will direct the error messages to the data area on the system CRT.

a. **Illegal entry: re-enter.**

During initialization the linker will indicate in the STATUS line on the system CRT that the user has made an illegal response to an interrogation. To correct this situation, re-enter the proper response.

b. **Duplicate symbol.**

During pass 1 of the link process, the linker detects that the same symbol has been declared global by more than one relocatable module. The first definition holds true. The relocatable module that first defines the symbol may be found in the cross-reference table. To correct this error, remove the extra global declarations.

c. **Load address out of range.**

The linker has tried to relocate code beyond the addressing range of the specified microprocessor. To correct this situation, reassign the relocatable addresses.

d. **Multiple transfer address.**

During pass 1, the linker finds that the transfer address has been defined by more than one relocatable module. The first definition holds true. The relocatable module that first defined the transfer address will be given at the conclusion of the linking. To correct this situation, remove the extra transfer address. Reassemble the amended relocatable module; then, relink. If a xfer address is defined by both a noload program and a load program, no error will be given. The load program xfer address takes precedence.

e. **Undefined symbol.**

During pass 2, the linker finds that a symbol has been declared external but not defined by a global definition. To correct this situation, define the symbol.

f. **Out of memory in xref.**

Unlike the fatal error (Out of Memory in Xref), this error occurs when memory space is available for a complete symbol table but only a portion of the cross-reference table. The linker will complete the xref operation, listing only that portion of the cross-reference table for which memory space was available. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

g. **Memory overlap.**

Relocatable program areas have been overlapped in memory. The error message will list the program names and the overlapping areas.

h. **Address out of range.**

The operand address is not within a valid addressing range for the specific microprocessor involved.

Chapter 6
Introduction to Assemblers

General

The information in this chapter is designed for those who are not familiar with assemblers or their operation. The topics are of a general nature and do not go into great detail. Some basic computer terminology is defined in Appendix A.

Assembly Language

Since a computer recognizes only strings of "1"s and "Ø"s (referred to as machine language), a second-level language (assembly language) was developed for programming ease. Assembly-language programming is the most fundamental form of program development. It consists of learning a particular microprocessor's mnemonics and composing a program in accordance with the operations that they perform. The assembler converts these mnemonic codes into binary format that the computer recognizes on a one-to-one basis.

Assemblers

An assembler is a stored program (stored in memory) that translates the data from a source program into relocatable object codes. The assembler allows you to represent numeric machine instructions by character strings called mnemonics. These mnemonic operation codes (opcodes) are easy to remember (for example, MOV for a move instruction, SHL for a shift-left instruction) and represent a valid machine instruction.

An assembler provides you with three programming tools: it allows you to specify instructions by name, to specify addresses by name, and to specify data in several forms other than binary. Instead of writing down a list of binary numbers for instructions, addresses, and data words, you list mnemonics for instructions words, symbols for the address words, and you specify data constants in more convenient decimal, octal, or hexadecimal data formats. The assembler then processes this list to create a corresponding list of binary codes.

In addition to freeing you from remembering all the machine codes, the assembler also keeps track of storage locations. Labels can be used for symbolic addressing and the assembler will assign a memory location to each label when that label is defined. Each time the program is changed, the assembler reassigns all the address labels and symbols. Symbols can also be used to define data constants.

While an assembler produces one machine code for every mnemonic in the source program, a macro-assembler expands that capability so that a single symbol can represent a group of machine instructions. For example, you may find that you are generating routines repeatedly, routines that are identical except for certain parameters. By symbolically identifying these routines at the beginning of the program, you can insert them anywhere in the program, along with the specific parameters needed, just by referring to their symbolic names. Such routines are called macros. Macro definitions are discussed in detail in Chapter 4.

NOTE

It should be noted that when you develop a macro definition you assign a "symbolic" name to identify that macro. Once assigned, the assembler treats the symbolic name as a mnemonic.

Assembler Operation

Assemblers normally make two passes through a source program to develop the machine coding required by the microprocessor. On pass-1 the assembler looks for user-created symbols and stores them in memory in a label table. On pass-2 the assembler recognizes the microprocessor instruction mnemonics and looks up their machine-code equivalents. In addition, it converts the operand fields in the program to machine code equivalents, using the label table to translate the user created symbols. (See figure 6-1.)

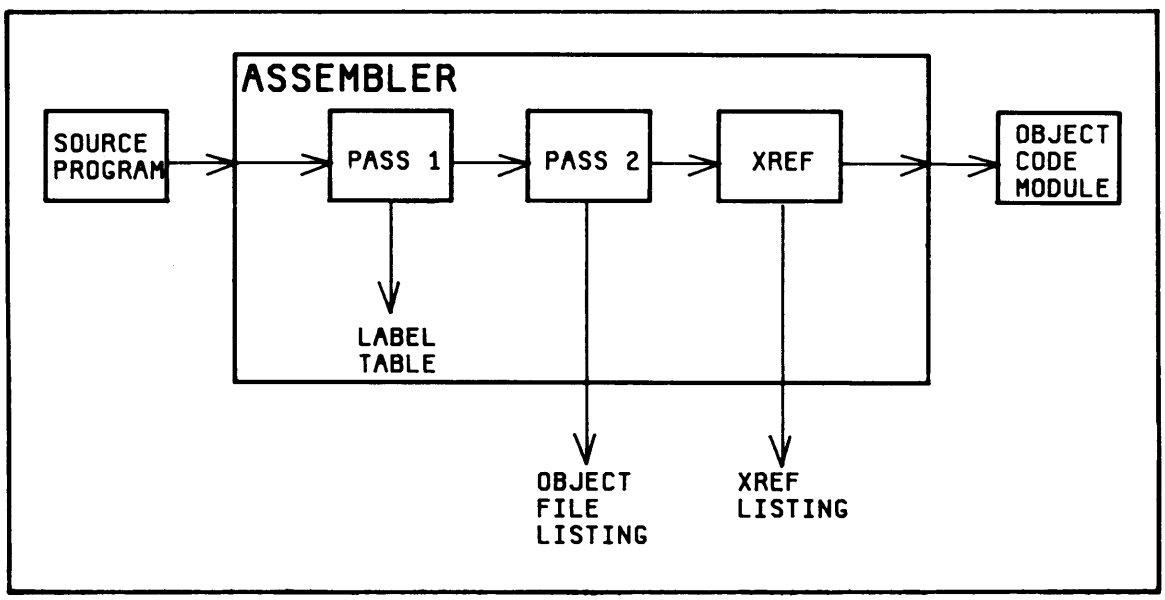


Figure 6-1. Assembly Flow Diagram

Source Program Format

As a rule, a single statement generates a single command. Each statement in the program contains "fields" which are designated as follows:

a. Label/Name (optional except for macro definitions and EQU statements)

b. Operation (Opcode)

c. Operand

d. Comments (optional)

The comment and label fields are optional. It should be remembered, however, that a symbol is required in the label field for a macro definition. Although the assembler will accept the macro that has no symbol assigned, you will never be able to call it. Refer to table 6-1 for a sample assembler listing. The table shows only two lines of instruction; however, the purpose of the table is to identify each field and its content only.

Table 6-1. Typical Assembler Listing

Machine Language		Assembly Language			English Language
1	2	3	4	5	6
Location	Object Code	Label (user-assg)	Opcode	Operand	Comments
0000			ORG	1000H	
1000	E7	BUMP	ADD	A,CHAR	;ADD CHAR ;TO ACCUM

Column: 1. Location: Memory location

2. Object Code: Basic machine language or object code that the microprocessor can understand

3. Label: User-created name for instruction

4. Opcode: Microprocessor operational code

5. Operand: Identifies register/data to be operated on

6. Comments: User comments for reference

NOTE

Columns 1 and 2 are created by the assembler. Columns 3, 4, 5, and 6 are generated by your source program.

In the label column of the program worksheet, you assign names or symbols for the various parts of the program. These symbols are defined when they appear in the label field or in the name field of an EQU statement or a macro definition.

In the opcode column, you must use the mnemonic instruction (codes assigned by the manufacturer of the microprocessor). You cannot assign your own names because the instructions for the microprocessor opcodes have been permanently written in the assembler program. Also, in this column, you will use macro instruction mnemonics that you developed for your program.

NOTE

Throughout this manual the term "symbol" refers to a user-assigned label that occupies the label field in a source statement. The term "mnemonic" refers to manufacturer assigned codes and are used only in the opcode field of the source statement.

In the operand column, you provide the data required by the opcode instruction. The values assigned to the data may be expressed numerically or symbolically. The symbols assigned to operands can be selected so as to suggest their purpose, making them as mnemonic as the opcodes.

In the comment field, you can write anything you want. The only restriction is that you must be sure that the proper delimiter character is entered before the comments so that the assembler will recognize the statements as comments and ignore them.

Normally, you will develop your source program using a program worksheet and pen or pencil. If a program is extremely long, you may sectionalize your program, breaking the long program into modules and writing each module separately. While developing your program you must follow certain rules and conventions that apply to the Model 64000 assembler. These rules and conventions are discussed in Chapter 2.

HP Model 64000 Assembler

The HP Model 64000 Assembler, using a specific program, converts a user's source program into executable machine language. The source program must be written using manufacturer's mnemonic codes. The program can be maintained by the Editor program (refer to HP Model 64000 Text Editor manual for further information).

The source program applied to the assembler will usually include assembler instructions (pseudo-codes) and control instructions. However, only the source program instructions are converted into executable object codes. The pseudo-codes and control instructions initiate various functions that direct and assist the assembler in its translation operation. The assembler outputs consist of the object file, program listings, and other information. The object file contains binary instructions and data constants that were coded from the source program. The entire object file must be linked and then loaded into program memory so that it can be executed on the Emulator Processor.

Program listings provide a permanent record of both the source program and the object codes developed. These listings, produced by the assembler, are composed of line numbers, the developed object codes, and the source codes. The assembler also provides error messages whenever errors are detected.

Following the source code listing, a symbol cross-reference table (optional) is produced. This table lists all program symbols alphabetically with their line numbers defined, plus the line number where the symbol was referenced. Following the cross-reference table (if generated) will be a statement indicating the number of errors noted, plus a reference to the last error. Following the error statement will be a listing of the error codes noted during the running of the program, plus their description.

Numbering Systems

In normal everyday use, a number means a decimal number. However, in digital electronics, all data in a computer are stored in binary form. A decimal (base 10) number will appear in the computer as a binary (base 2) number because the computer will convert the decimal number to its binary equivalent. Other numbering systems, such as octal (base 8) or hexadecimal (base 16) are also used in computers depending on the byte and word structure of the particular computer. The following paragraphs describe the several computer numbering systems.

Binary Numbering System

The binary numbering system is based on two states, Ø and 1. Where the decimal system uses ten digits (Ø through 9), thus having a base of 1Ø, the binary numbering system has two digits, Ø and 1, and has a base of 2. In the binary system, to represent numbers greater than 1 we must use more than one digit. Each digit in a binary string is weighted and its value depends on its location in the binary string. The sum of the weighted values of the digits produce the decimal equivalent.

Example:

Weight:	64	32	16	8	4	2	1
Binary String:	1	Ø	Ø	1	Ø	Ø	1

>Decimal Equivalent = 73

Octal Numbering System

The octal numbering system is a system with a base of 8. Its numbers are commonly expressed either with decimal or binary digits. The use of the octal numbering system is common in computer systems because it allows the conversion of large binary numbers to a simpler form. Every octal digit represents exactly three binary digits. Converting binary numbers into their octal equivalent is very straight forward; you simply partition the binary string into groups of three digits and replace each group of digits with its octal digit equivalent. This can be illustrated with the binary string used in the previous binary example:

Binary String:	1 Ø Ø 1 Ø Ø 1		
Partitioned Elements:	Ø Ø 1	Ø Ø 1	Ø Ø 1
Octal Equivalent:	1	1	1

Therefore, the octal equivalent 111 equates with the decimal number 73 and the binary string 1ØØ1ØØ1.

Hexadecimal Numbering System

The hexadecimal numbering system is a system with a base of 16. Its numbers are expressed with decimal digits and characters of the alphabet (A through F). Hexadecimal numbers can be converted to decimal numbers in the same manner that octal numbers are converted. Instead of breaking the binary string into groups of three bits as you did to convert to the octal equivalent, you simply partition the binary string into groups of four binary digits and replace each group of digits with its hexadecimal equivalent. Again using the binary string from the previous example:

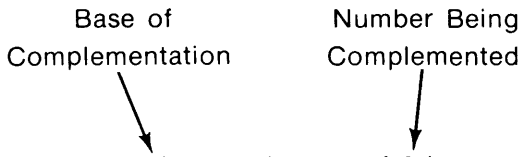
```
Binary String:          1 Ø Ø 1 Ø Ø 1

Partitioned Elements:   Ø 1 Ø Ø   1 Ø Ø 1

Hexadecimal Equivalent:     4         9
```

Therefore, the hexadecimal equivalent 49 equates with the octal number 111, decimal number 73, and binary string 1ØØ1ØØ1. To illustrate the letter values of a hexadecimal number the following examples are given:

```
Binary              Hexadecimal

1 Ø 1 Ø <---------- A

1 1 Ø Ø <---------- C

Ø 1 Ø Ø 1 1 1 1 <---------- 4F

1 Ø Ø 1 1 1 1 Ø <---------- 9E

1 Ø Ø 1 1 Ø Ø Ø <---------- 98
```

Complement of Numbers

The complement of a number is the difference between the base of the complementation and the number being complemented. For example:



The 1Ø's complement of 2 is 8 (base 1Ø-2)

The 9's complement of 3 is 6 (base 9-3)

The 2's complement of 1 is 1 (base 2-1)

The 1's complement of 1 is Ø (base 1-1)

From the examples given it can be seen that a complement of any arbitrary base and any number can be obtained. However, the complements most useful in the binary number systems are the 1's and 2's complements listed above. There are two main reasons for using complements:

1. They can be used to represent negative numbers.

2. They can be used to perform subtraction operations by means of an "addition" operation.

1's Complement

Because the binary system has only two states, the 1's complement of a binary number can be obtained by writing every bit in its opposite state, that is, every Ø bit is changed to a 1, and every 1 bit is changed to a Ø. For example:

```
Binary Number: Ø 1 Ø 1 Ø 1

   Complement: 1 Ø 1 Ø 1 Ø
```

Optionally, complementing can be performed by subtracting from all 1 bits as follows:

```
  1 1 1 1 1 1
- Ø 1 Ø 1 Ø 1  <----- Binary Number
-------------
  1 Ø 1 Ø 1 Ø  <----- Complement
```

2's Complement

The easiest method of finding a 2's complement of any binary number is to first obtain its 1's complement, then add 1 as follows:

```
  Ø 1 Ø 1 Ø 1  <----Number

  1 Ø 1 Ø 1 Ø  <----1's Complement
      + 1      <----Add 1
-------------
  1 Ø 1 Ø 1 1  <----2's Complement
```

Optionally, a 2's complement can be found by subtracting from a 1 followed by Ø's as follows:

```
1 Ø Ø Ø Ø Ø Ø
  Ø 1 Ø 1 Ø 1  <----Number
---------------
  1 Ø 1 Ø 1 1  <----2's Complement
```

Appendix A
Glossary

a

ASCII	American Standard Code for Information Interchange. A seven bit character code.
Absolute Address	A number used to refer directly to a specific memory location.
Accumulator	A register used to accumulate the results of operations.
Address	To specify a memory location, or the specific location of data in memory.
Addressing Modes	The various means of accessing memory (see direct, indirect, relative, indexed, etc.).
Analysis	As applied to microprocessor development systems, analysis consists of breaking down the operation of a microprocessor system into time and state sequence of bus transactions. The system can then be studied by investigating the state sequences.
Array	An indexed set of variables. In mathematics, arrays are often operated upon as units by applying special arithmetic rules.
Assembler	The program which performs the transformation from assembly language to object code.
Assembly Language	The language defined for a particular processor; composed of mnemonic opcodes and operands which allow use of the processor's machine instructions.
Asynchronous	Not describable in terms of fixed units of time; occurring at various time intervals.

b

BCD	Binary coded decimal. A system of representing decimal numbers.
Batch (processing)	Processing a number of commands or programs without user interaction. (See Command Files)
Baud	A measure of data flow. The number of signal elements per second based on duration of the shortest element.
Benchmark	A frequently used routine or program selected for the purpose of comparison.
Bidirectional	A term applied to a port or bus line that can be used to transfer data in either direction.
Block (PASCAL)	In a programming language, a bracketed segment of program text containing declarations of variables and a sequence of statements.
Branch	To depart from the normal sequence of executing instructions.
Breakpoint	A hardware or software condition (bit pattern) that stops program execution - e.g., specific addresses or control signals.
Byte	Eight binary digits (bits).

c

CPU (Central Processing Unit)	Computer unit which controls the processing routines and performs arithmetic functions.
Call	The branching or transfer of control to a specified subroutine.
Command (key word)	A reserved word for the operating system which cannot be used for data names, file names, or program names. Key words can only be used as defined by the subsystem.

Command File	A source file which contains system commands intended to be executed in sequential order optionally with parameter text substitution.
Comment	Annotation within the text of a program, that is not interpreted by the computer as part of the program.
Compiler	A program which translates source text in a high level language (e.g., PASCAL, FORTRAN) into low level object code for some processor.
Control Bus	A group of parallel signal paths that transfer electrical signals to regulate computer system operations. In particular, the control bus drives system functions such as timing, data transfer, and initialization/termination of program execution.
Cursor	The blinking underline prompt function that usually corresponds to many assembly language instructions.

d

DMA (Direct Memory Access)	Control of address and data bus without CPU.
Data Bus	A bi-directional signal path that transfers data to and from the CPU, memory storage, and peripheral devices.
Debug	To locate and correct any errors in a computer program or in hardware.
Declaration (PASCAL)	One or more instructions which specify the type, characteristics, or amount of data associated with identifiers.
Default Value	The value assumed by a parameter when no other value is assigned to it.
Delimiter	A character used to separate fields in a command.
Direct Addressing	An addressing mode characterized by the ability to reach memory storage directly.

e

Edit	To alter a source file in any fashion (including creation).
Editor	The program which allows editing of a source file.
Emulation	A hardware model of the target microprocessor used by the MDS to check-out the target system. This can be either the same microprocessor model as used in a target system, or bit-slice architecture that mimics the function of the target microprocessor. Using the target microprocessor is called substitutional emulation, or in-circuit emulation.
Event	Describes a system at a given point in time in terms of the current address, data and status information available.
Execution Time	The time necessary for a CPU to carry out a process.
Expression	In the text of a program, a string of identifiers and operator symbols describing an evaluation.
Extended Addressing	An addressing mode that allows access to any place in memory. (This is machine dependent, e.g., 6800.)
External Reference	A reference to a location (Variable Label) not defined by the program making the reference but by another program linked to it which declared the name to be global.

f

Fetch	That portion of a computer cycle during which the location of the data or the next instruction is determined, the instruction is taken from memory and processed.
Fields	Describes items or areas within an instruction which are expected to contain specific information.
File	A data record referenced by a name, a user identification, a disc number, and a file type.

File Manager

A memory management system that controls the organization and access of files in the memory system.

Firmware

Software instructions stored in ROM.

Floating Point

Floating point is a notation convention used to represent a wide range of real numbers in the computer. Each number is considered to consist of a mantissa, M, and an exponent, E. (The exponent is sometimes called the characteristic.) Any number N in radix (base) R can be represented as the produce of the mantissa multiplied by the radix taken the exponent power:

$N = M * R^E$

The number 123456 in base 10 can be represented in foating point notation as any of the following:

$123456 * 10^0$

$12.3456 * 10^4$

$.123456 * 10^6$

Flow Chart (Diagram)

A graphical representation of a sequence of operations using symbols to represent the operations (e.g., compute, substitute, compare, jump, copy, read, write).

g

Global Reference

1) A reference to a location given a name by a program declaring it GLB (global), which may be accessed by other programs linked to it by their declaring it EXT (external).

2) (PASCAL) A reference to a location given a name by a main program, which may be accessed by the main program and all of its subroutines.

h

HP-IB — Hewlett-Packard implementation of the IEEE-488 bus specification.

Handshaking — A synchronization process by which communication is established between receiving and transmitting circuits. Handshaking refers to interaction between the CPU and peripheral devices. For instance, the CPU outputs a word to a printer. The printer will then tell the CPU when it has finished printing and is ready for a new character. In more sophisticated systems, the CPU can determine (and act upon) several status conditions of both input and output devices.

Hexadecimal — A base 16 number system.

High-level Language — A procedure (or problem) oriented language which allows you to describe tasks that are problem oriented rather than computer oriented. Each statement in a high-level language performs a recognizable function that usually corresponds to many assembly language instructions.

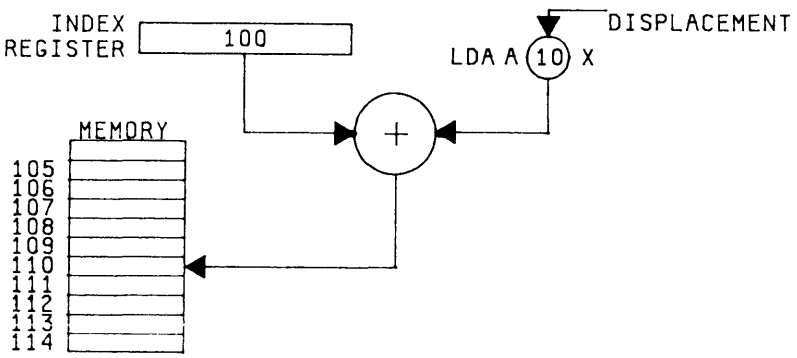
i

Immediate Addressing — An addressing mode in which the operand contains the value to be operated on with no further address reference required.

Indexed Addressing — A computer instruction which uses indexed addressing refers to the contents of a memory location whose address is computed by adding a displacement included with the instruction to the contents of an index register. For example, in the Motorola 6800, the instruction sequence:

```
LDX #100
LDA A 10,X
```

will load the A accumulator with the contents of the location specified by adding the displacement of 1Ø to the 1ØØ in the index register. Indexed addressing is a very convenient way to handle manipulations of data in tables. The index register is initialized to the start of the area containing the data. The data can then be sequentially accessed by modifying the index register contents.



Index Register — CPU register whose contents can be used to form an indexed address. In most computers the index registers can also be used for temporary data storage and other program operations.

Indirect Addressing — An addressing mode which addresses a memory location that contains the address of the data rather than the data itself.

Instruction Set — The group of instructions which can be executed by a given microprocessor.

Interface — A common boundary between adjacent components, circuits, or systems that enables devices to yield and/or acquire information from one another.

Internal Variable (also Local Variable) — A variable that pertains only to the procedure of a program in which it is declared.

Interrupt — The suspension of the normal programming routine of a microprocessor in order to handle a request for service.

Inverse Video — A display enhancement mode in which normal white on black background characters appear as black and white background.

I/O

Abbreviation for Input-Output. This refers to data transfers to and from the CPU. These designations should be used relative to the processor to avoid confusion. For instance, output data from the computer is input data for a TTY and output data from a keyboard is input data to the CPU. In this case, the TTY is an output device and the keyboard is an input device.

k

Key Words

A set of reserved words for an operating system which cannot be used for data names, file names, or program names. Key words can only be used as defined by the subsystem.

l

Label

An identifier which corresponds to a memory location in the programmable system. Once labeled, the specific address may be referred to by the label name.

Library

One or more relocatable files appended one behind another in a single file. (See Linker Manual for use.)

Link

The process of combining one or more object code files into a single absolute file.

Linker

The program which takes assembler or compiler generated object code files and locates them in specified addresses generating a single absolute file. The linker modifies program object modules so they can properly reference each other.

Logic Analyzer

Equipment that displays program timing and response signals. The trigger is normally a match with a specified bit pattern, or a signal that fills the logic analyzer's buffer. Individual probes that can be attached to any desired signal line greatly enhance the power of the analyzer.

m

MDS	Microprocessor Development System.
Machine Code	Collective term for machine instructions, represented by a hex code or octal code.
Machine Instruction	A single command to a microprocessor directing it to take some action it is capable of performing.
Machine Language	A system of binary digits for a computer by which information or data can be read directly, and used without further processing.
Macro Command	A program entity formed by a string of commands which are put into effect by means of a single command.
Mapping (Memory)	A mode of operation in a computer that provides dynamically relocatability for programs.
Memory	An organization of storage units, primarily for retrieval of information and data. Memory types include disk, semiconductor, magnetic tape, etc.
Microprocessor	A central processing unit fabricated on one or several ICs.
Mnemonic Code	Identifiers assigned to machine language instructions that suggest the definition of the instructions.
Modular Programming	A block oriented program structuring.
Monitor	The specific program to schedule and control the operation of a system (Executive).

n

Nesting	Including a routine or block of data within a larger routine or block of data. A series of looping instructions may be nested within each other.
Nibble	Four binary digits.

o

Object Code	The code output by the assembler or compiler which is comprehensive to the linker (see relocatable).
Operand	A part of a computer instruction which may be an argument, a result, a parameter, or an indication of the location of the next instruction.
Operation Code (Opcode)	A combination of bits specifying an absolute machine-language operator, or the symbolic representation of the machine-language operator.
Overlay	A memory management technique of repeatedly using the same blocks of storage during different stages of a program; e.g., when one routine is no longer needed in storage, another routine can replace all or part.

p

PDS	Processor Development System.
PROM	Programmable read-only memory.
Packed (PASCAL)	An array that is stored in the minimum amount of memory possible.
Port	The point of a computer at which the I/O is in contact with the outside world and allows the CPU to perform I/O.
Post Trigger	An analysis technique that acquires data after a delay from a trigger point.
Pre Trigger	An analysis technique in which data is acquired before a trigger point.
Procedure (PASCAL)	Program statements which are combined to form named paragraphs. Paragraphs may be combined to form sections. Paragraph and section names are assigned by the programmer so control may be transferred from one section or paragraph to another.

Program Loop (PASCAL) — A sequence of instructions that is repeated until a terminal condition prevails.

Pseudo Op — In assembly language, a Pseudo Opcode is an opcode which generates no machine instructions but instructs the assembler.

r

RAM — Random-Access Memory.

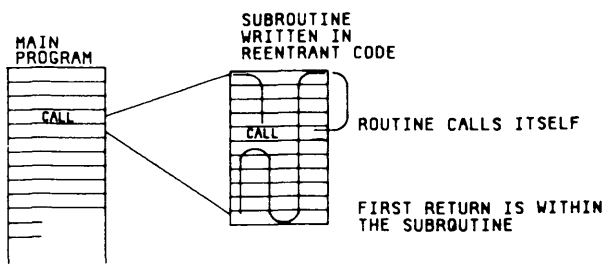
Real-time Emulation — This term indicates that the emulator (see definition) works at the speed of the target system - otherwise timing problems may not show. This is especially important when the clock is linked with I/O decoding or when used for noise reduction.

Real Time Operation — Operation at full speed with no artificial interruption of execution.

Real Time Trace — Program monitoring at the full operating speed of the system.

Reentrant — The property of a program that enables it to be interrupted at any point; reentered and executed under interrupt; and then resumed from the point of interruption without loss of integrity.

Reentrant Code — This is a program or portion of a program which can be used simultaneously by different routines. It may call itself repeatedly or may call a routine which in turn calls the reentrant coded program again. This type of code cannot store data in absolute addresses, store data in temporary CPU registers, or modify any portion of itself. Storage information must be done through stack operations or in other orderly sequential storage so that the program can return from the sequential call statements properly. An example of a simple flow between a main program and a reentrant coded subroutine is shown below.



Recursive (Pascal)	A procedure that calls itself directly or indirectly.
Refresh	The process of systematically and periodically accessing dynamic memory for the purpose of recharging capacity storage elements.
Register	A memory unit usually one word long, contained within the CPU for use in performing operations.
Relative Addressing	An addressing mode in which the desired address is described relative to another base address.
Relocatable	Descriptive of object files which can be modified to be executed in any memory location.
ROM	Read-Only Memory.
RS-232C	An IEEE standard for serial data communication.

S

Scalar (PASCAL)	A data type that is an ordered set of identifiers.
Simulator	A special program that simulates logical operation of a microprocessor or I/O or procedure.
Soft Key	A key whose label appears on the CRT display and thus may be relabeled to perform different functions over time.
Software	Written programs or data for computer applications.
Source	Information coded in other than machine language that must be translated into machine language before use.
Statement	An imperative sentence in a programming language interpreted as one or a sequence of instructions by a computer.
Status Word Register	A register which holds binary data representing the state of various parameters in the microprocessor. Typical parameters may include zero flag, carry flag, parity flag, accumulator content, and sign bit.

String	A sequence of characters delimited by ' ' or ∧.
Subroutine	Part of a master program or routine which may be used at will in a variety of master routines. The object of a branch or call command.
Symbolic Addressing	Referencing addresses given symbolic names by a program (see LOCAL, GLOBAL, EXTERNAL references).
Synchronous	Clocked operation.
Syntax	A set of rules for specifying the sentence structure and statement structure of a language.

t

Target System	The microprocessor based system under development; the prototype.
Trace	A software diagnostic technique used to follow program execution step by step to determine where an error is occurring. A running trace usually displays the contents of CPU registers as each instruction is executed, thereby enabling the user to determine where values are not changing as predicted.
Transparent	A property of hardware or software which need not be understood by a user in order to operate a system. Generally, this hardware or software is never seen by the user and is therefore "transparent."
Trigger Point	A defined event that initiates or precipitates a reaction (e.g., a specific data transfer which initiates data acquisition).
Two's Complement Numbers	A representation of positive and negative binary numbers which is distinguished by having one representation for the value of zero (Ø). Positive numbers are identically represented in one's complement and two's complement numbers. Negative numbers are not identical. The two's complement of a positive binary number is formed by complementing the magnitude of the positive number and adding the last significant digit.

Example: $42.5_{10} = 0\ 101010.1$

Complement magnitude (42.5) = 0 010101.0
add 1 to LSB 1
———————
2's complement of $42.5_{10} = -42.5_{10}$ 1 010101.1

Check $42.5_{10} + (-42.5_{10})$ should equal zero

42.5_{10}	0 101010.1
-42.5_{10}	1 010101.1
———	———
Total = 0 =	1 0000000.0

carry ignored

u

User ID — An identifier used in the System 64000 to differentiate between users of the system. A user id may be up to six alphanumeric characters but must begin with an upper case alphabetic character.

v

Variable — A named object that can hold a value from a designated data type, and can receive new values by assignment.

Vectored Interrupt — An interrupt system which employs a table of pointers (vectors) indicating the location of interrupt service routines.

w

Word — A sequence of binary digits treated as a unit.

Appendix B

ASCII Conversion Table

U.S Standard Code for Information Interchange (ASCII)

ROW (HEX)	BITS 7,6,5-> 4,3,2,1 ↓	000	001	010	011	100	101	110	111
	COLUMN (HEX)	0	1	2	3	4	5	6	7
0	0000	NUL	DLE	SP	0	@	P	'	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

bits

7 1

Example: Code for B = 100 0010 (Hex = 42)
Code for Z = 101 1010 (Hex = 5A)
Code for n = 110 1110 (Hex = 6E)

Appendix C

Assembler Pseudo Instructions Summary

General

Assembler instructions can be specified during the assembly operation, or can be embedded in the source program.

Macro definitions and calls are not listed in this summary. Refer to Chapter 4 for macro information.

Summary

Operation Code	Function
ASC	Stores data in memory in ASCII format.
BIN	Stores data in memory in binary format.
COMN	Assigns common block of data or code to a specific location in memory.
DATA	Assigns data to a specific location in memory.
DECIMAL	Stores data in memory in decimal format.
END	Terminates the logical end of a program module. Operand field can be used to indicate starting address in memory for program execution.
EQU	Defines label field symbol with operand field value. Symbol cannot be redefined.
EXPAND	Causes an output listing of all source and macro generated codes.
EXT	Indicates symbol defined in another program module.

Operation Code	Function
GLB	Defines symbol that is used globally (referenced by other program modules).
HEX	Stores data in memory in hexadecimal format.
IF	Allows sections of code to be conditionally assembled.
INCLUDE	Allows a secondary file to be included in the source input stream.
LIST	Used to modify output listing of program.
MASK	Performs AND/OR logical operations on designated ASCII string.
NAME	Permits user to add comments for reference in the load map.
NOLIST	Suppresses output listings (except error messages).
OCT	Allows user to store data in octal format.
ORG	Sets program counter to specific memory address for absolute programming.
PROG	Assigns source statements to a specific location in memory. Assembler default condition is PROG storage area.
REAL	Converts real decimal numbers to IEEE binary floating point constants.
REPT	Enables user to repeat a source statement any given number of times.
SET	Defines label field symbol with operand field value. Symbol can be redefined.
SKIP	Enables user to skip to a new page to continue program listing.
SPC	Enables user to generate blank lines within program listing.
TITLE	Enables user to create a text line at the top of each page listing for the source program.
WARN NOWARN	Turn warning message in source listing on and off.

Appendix D

List of Assembler Error Messages

Detection and Listing

The assembler detects and lists all errors noted in a source program module. The program errors are indicated in the source program listing by a two-letter code following each source statement that contains an error.

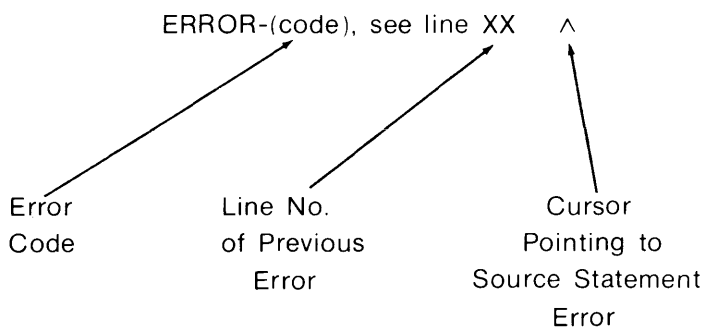
NOTE

If multiple errors occur in the same source statement, only the first error noted will be reported (in most cases).

Each error message contains an error code, a cursor (^) that points to the error location in the source statement, and a statement that indicates the line number of the previous source statement that was in error (facilitates error tracing).

A summary of the number of errors within the program, along with a brief description of all error codes noted, is given at the end of the program listing.

The error message format is as follows:



Error Codes

The list of error codes (in alphabetical order) along with a description of their meaning is as follows:

Code	Error Definition
AS	ASCII STRING — The length of ASCII string was not valid or the string was not terminated properly.
CL	CONDITIONAL LABEL — Syntax of a conditional macro source statement requires a conditional label that is missing.
DE	DEFINITION ERROR — Indicated symbol must be defined prior to it being referenced. Symbol may be defined later in program sequence.
DS	DUPLICATE SYMBOL — Indicates that the defined symbol noted has been previously defined in the program assembly sequence. This occurs when the same symbol is equated to two values (using EQU directive) or when the same symbol labels two instructions.
DZ	DIVISION BY ZERO — Invalid mathematical operation resulting in the assembler trying to divide by zero.
EG	EXTERNAL GLOBAL — Externals cannot be defined as globals.
EO	EXTERNAL OVERFLOW — Program module has too many external declarations (512 externals maximum).
ES	EXPANDED SOURCE — Indicates insufficient input buffer area to perform macro expansion. It could be the result of too many arguments being specified for a parameter substitution, or too many symbols being entered in the macro definition.
ET	EXPRESSION TYPE — The resulting type of expression is invalid. Absolute expression was expected and not found or expression contains an illegal combination of relocatable types (refer to Chapter 2 for rules and conventions).

Code	Error Definition
IC	ILLEGAL CONSTANT — Indicates that the assembler encountered a constant that is not valid. **For Example:** 1Ø9B (9 is invalid) 97Q (9 is invalid)
IE	ILLEGAL EXPRESSION — Specified expression is either incomplete or an invalid term was found within the expression.
IO	INVALID OPERAND — Specified operand is either incomplete or inaccurately used for this operation. This occurs when an unexpected operand is encountered or the operand is missing. If the required operand is an expression, the error indicates that the first item in the operand field is illegal.
IP	ILLEGAL PARAMETER — Illegal parameters in macro header.
IS	ILLEGAL SYMBOL — Syntax expected an identifier and encountered an illegal character or token.
LR	LEGAL RANGE — Address or displacement causes the location counter to exceed the maximum memory location of the instruction's addressing capability.
MC	MACRO CONDITION — Relational (conditional) operator in macro is invalid.
MD	MACRO DEFINITION — Macro is called before being defined in the source file. Macro definition must precede call.
ML	MACRO LABEL — Label not found within the macro body.
MM	MISSING MEND — Indicates that a macro definition with a missing MEND directive was included in the program.
MO	MISSING OPERATOR — An arithmetic operator was expected but was not found.
MP	MISMATCHED PARENTHESIS — Missing right or left parenthesis.

Code	Error Definition
MS	MACRO SYMBOL — A local symbol within a macro body was required but was not found.
PC	PARAMETER CALL — Invalid parameter in macro header.
PE	PARAMETER ERROR — An error has been detected in the macro parameter listed in the source statement.
RC	REPEAT CALL — Repeat cannot precede a macro call.
RM	REPEAT MACRO — The repeat pseudo-operation code cannot precede a macro definition.
SE	STACK ERROR — Indicates that a statement or expression does not conform to the required syntax.
TR	TEXT REPLACEMENT — Indicates that the specified text replacement string is invalid.
UC	UNDEFINED CONDITIONAL — Conditional operation code invalid.
UO	UNDEFINED OPERATION CODE — Operation code encountered is not defined for the microprocessor, or the assembler disallows the operation to be processed in its current context. This occurs when the operation code is misspelled or an invalid delimiter follows the label field.
UP	UNDEFINED PARAMETER — The parameter found in a macro body was not included in the macro header.
US	UNDEFINED SYMBOL — The indicated symbol is not defined as a label or declared an external.

Subject Index

I

m

n

o

p

r

s

t

u

w

x

OPERATING MANUAL CHANGES

MANUAL IDENTIFICATION

Model Number: 64100A

Manual Title: Assembler/Linker Ref. Manual

Part Number: 64980-90997

Date Printed: July 1983

This supplement contains important information for correcting manual errors and for adapting the manual to instruments containing improvements made after the printing of the manual.

To use this supplement:

Make all ERRATA corrections.

Make all appropriate software or hardware related changes indicated in the tables below.

Software Rev. and Date	Make Manual Changes
SSS Nov. 1, 1983	Change 1
SSS Feb. 1, 1984	Change 1, 2

Software Rev. and Date	Make Manual Changes

▲ NEW ITEM

CHANGE 1

Chapter 5, Page 5-6, subparagraph i:
Change softkey display to the following:

[edit] [nolist] [xref] [no_overlap] [ ] [ ] [ ] [ ]

Chapter 5, Page 5-9, subparagraph h:
Change to read as follows:

h. The next command query displayed in the command line on the system CRT concerns output listing options and memory overlap check. It has the following syntax:

List,xref,overlap__check = on, off, on

The linker asks you to specify what output listings are required and if memory overlap should be checked. Using the [on] or [off] softkey select, in the sequence indicated in the syntax statement (list, xref, overlap__check), the desired output listing and memory check condition. After inserting the requirements, proceed to the next query.

NOTE

Manual change supplements are revised as often as necessary to keep manuals as current and accurate as possible. Hewlett-Packard recommends that you periodically request the latest edition of this supplement. When requesting copies quote the manual identification information from your supplement, or the model number and print date from the title page of the manual.

HEWLETT PACKARD

Printed in U.S.A.

MANUAL IDENTIFICATION

Model Number: 64100A

Manual Title: Assembler/Linker Ref. Manual

Part Number: 64980-90997

Date Printed: July 1983

CHANGE 1 (CONT'D)

Chapter 5, Page 5-9, subparagaph h: (Cont'd)

NOTE

The output listings and memory check indicated after the list, xref, overlap__check = query are the command file values that will be used during this and future operations. They may be overridden by using the (options) softkey during the linker call.

The default condition for this query is: on, off, on.

▲ CHANGE 2

Chapter 5, Pages 5-3/5-4:
Delete.
Add attached pages 5-3/5-4.

Printed in U.S.A.

64980-90997, JULY 1983
Replaces: 64980-90997, July 1982

PRINTED IN U.S.A.