DIGITAL RESEARCH™

Programmer's Utilities Guide

For the
CP/M® Family of
Operating Systems

DIGITAL RESEARCH™

Programmer's Utilities Guide

For the
CP/M® Family of
Operating Systems

COPYRIGHT

DISCLAIMER

TRADEMARKS

The Programmer's Utilities Guide for the CP/M Family of Operating Systems was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

First Edition: September 1982

Programmer's Utilities Guide for the CP/M® Family of Operating Systems

Release Notes

Addendum to the First Printing--1982

The following LINK-80™ option switches are not documented in Section 15.4 of The Programmer's Utilities Guide for the CP/M® Family of Operating Systems.

The BIOS Link (B) Switch

The B switch is used to link a BIOS in a banked CP/M 3 system. LINK-80 aligns the data segment on a page boundary, puts the length of the code segment in the header, and defaults to the SPR filetype.

The Output RSP File (OR) Switch

The OR switch outputs RSP (Resident System Process) files for execution under MP/M™ .

The Output SPR File (OS) Switch

The OS switch outputs SPR (System Page Relocatable) files for excution under MP/M.

Foreword

This manual describes several utility programs that aid the programmer and system designer in the software development process. Collectively, these utilities allow you to assemble 8080 assembly language modules, link them together to form an executable program, and generate a cross-reference listing of the variables used in a program. With these utilities, you can also create and manage your own libraries of object modules, as well as create large programs by breaking them into separate overlays.

The Programmer's Utilities Guide assumes you are familiar with the CP/M® or MP/M II™ Operating System environment. It also assumes you are familiar with the basic elements of assembly language programming as described in the 8080 *Assembly Language Programming Manual,* published by Intel®.

MAC™, the CP/M macro assembler, translates 8080 assembly language statements and produces a hex format object file suitable for processing in the CP/M environment. MAC is upward compatible with the standard CP/M nonmacro assembler, ASM™. (See the CP/M documentation published by Digital Research.)

MAC facilities include assembly of Intel 8080 microcomputer mnemonics, along with assembly-time expressions, conditional assembly, page formatting features, and a powerful macro processor compatible with the standard Intel definition. MAC also accepts most programs prepared for the Processor Technology Software #1 assembler, requiring only minor modifications. This revision is not compatible with previous versions.

MAC is supplied on a standard disk, along with a number of library files. MAC requires about 12K of machine code and table space, along with an additional 2.5K of I/O buffer space. Because the BDOS portion of CP/M is coresident with MAC, the minimum usable memory size for MAC is about 20K. Any additional memory adds to the available Symbol Table area, allowing larger programs to be assembled.

Sections 1 through 5 describe the simple assembler facilities of MAC: 8080 mnemonic forms, expressions, and conditional assembly. These facilities are similar to those of the CP/M assembler (ASM). If you are familiar with ASM, you might want to skip Sections 1 through 5 and begin with Section 6.

Sections 6 through 8 describe MAC macro facilities in detail. Section 7 describes inline macros, and Section 8 explains the definition and evaluation of stored macros. If you are familiar with macros, briefly skim these sections, referring primarily to the examples. Section 9 explains macro applications, common macro forms, and programming practices. Skim the examples and refer back to the explanations for a detailed discussion of each program.

Sections 10 through 13 describe other features of macro assembler operation. Section 10 details assembly parameters. Section 11 introduces iterative improvement, a common debugging practice used in developing macros and macro libraries. Section 12 defines MAC's symbol storage requirements.

Section 13 explains the differences between MAC and RMAC™, the CP/M Relocating Macro Assembler.

Section 14 details XREF, an assembly language cross-reference program used with MAC and RMAC.

Section 16 describes LINK-80™, the linkage editor that combines relocatable object modules into an absolute file ready to run under CP/M or MP/M II. Section 17 describes how to use LINK-80, in conjunction with the PL/I-80™ compiler, to produce overlays. Section 18 explains how to use LIB-80™, the software librarian for creating and manipulating library files containing object modules.

The appendixes contain a complete list of error messages output by each of the utility programs.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Appendixes

Table of Contents (continued)

List of Tables

List of Figures

List of Listings

Table of Contents (continued)

Table of Contents (continued)

Section 1
Macro Assembler Operation

Start MAC with a command of the form:

MAC filename

where filename corresponds to the assembly language file with an assumed filetype ASM. During the translation process, MAC creates a file called filename.HEX containing the machine code in the Intel hexadecimal format. You can subsequently load or test this HEX file. (See the LOAD command and the Dynamic Debugging Tool, DDT™, in the CP/M documentation.) MAC also creates a file named filename.PRN containing an annotated source listing, along with a file called filename.SYM containing a sorted list of symbols defined in the program.

Listing 1-1 provides an example of MAC output for a sample assembly language program stored on the disk under the name SAMPLE.ASM. Type MAC SAMPLE followed by a carriage return to execute the macro assembler. The PRN, SYM, and HEX files then appear as shown in the listing. The assembler listing file (PRN) includes a 16-column annotation at the left showing the values of literals, machine code addresses, and generated machine code. Note that an equal sign (=) is used to denote literal values to avoid confusion with machine code addresses. (See Section 4.3.) Output files contain tab characters (ASCII CTRL-I) whenever possible to conserve disk space.

Source Program (SAMPLE.ASM)

```
        org     100h    ;transient program area
bdos    equ     0005h   ;bdos entry point
wchar   equ     2       ;write character function
;       enter with ccp's return address in the stack
;       write a single character (?) and return
        mvi     c,wchar ;write character function
        mvi     e,'?'   ;character to write
        call    bdos    ;write the character
        ret             ;return to the ccp
        end     100h    ;start address is 100h
```

Listing 1-1. Sample ASM, PRN, SYM, and HEX files from MAC

Assembler Listing File (SAMPLE.PRN)

```
0100            ORG     100H            ;TRANSIENT PROGRAM AREA
0005 =          BDOS    EQU     0005H   ;BDOS ENTRY POINT
0002 =          WCHAR   EQU     2       ;WRITE CHARACTER FUNCTION
                ;       ENTER WITH CCP'S RETURN ADDRESS IN THE STACK
                ;       WRITE A SINGLE CHARACTER (?) AND RETURN
0100 0E02               MVI     C,WCHAR ;WRITE CHARACTER FUNCTION
0102 1E3F               MVI     E,'?'   ;CHARACTER TO WRITE
0104 CD0500             CALL    BDOS    ;WRITE THE CHARACTER
0107 C9                 RET             ;RETURN TO THE CCP
0108                    END     100H    ;START ADDRESS IS 100H
```

Assembler Sorted Symbol File (SAMPLE.SYM)

```
0005 BDOS        0002 WCHAR
```

Assembler Hex Output File (SAMPLE.HEX)

```
:080100000E021E3FCD0500C9EF
:00010000FF
```

Listing 1-1. (continued)

End of Section 1

Section 2
Program Format

A program acceptable as input to the macro assembler consists of a sequence of statements of the form

line# label operation operand comment

where any or all of the elements can be present in a particular statement. Each assembly language statement terminates with a carriage return and line-feed. Note that the ED program automatically inserts the line-feed when you enter a carriage return. You can also terminate an assembly language statement by typing the exclamation point (!) character. MAC treats this character as an end-of-line. You can write multiple assembly language statements on the same physical line if you separate them with exclamation points.

A sequence of one or more blank or tab characters delimits statement elements. Tab characters are preferred because they conserve source file space and reduce the listing file size. The tab characters are not expanded until the file is printed or typed at the console.

The line# is an optional decimal integer value representing the source program line number. It is allowed on any source line. The assembler ignores the optional line#.

The label field takes the form:

identifier

or

identifier:

The label field is optional, except where noted in particular statement types.

The identifier is a sequence of alphanumeric characters: alphabetics, question marks, commercial at-signs, and numbers, the first character of which is not numeric. You can use identifiers freely to label elements such as program steps and assembler directives, but identifiers cannot exceed 16 characters in length.

All characters are significant in an identifier, except for the embedded dollar sign ($) that you can use to improve name readability. Further, MAC treats all lower-case alphabetics in an identifier as though they were upper-case. Note that the colon (:) following the identifier in a label is optional. The following examples are all valid labels:

```
x               xy              long$name
x?              xy1:            longer$named$data
x1x2            @123:           ??@@abcDEF
Gamma           @GAMMA          ?ARE$WE$HERE?
x234$5678$9012$3456:
```

The operation field contains an assembler directive (pseudo operation), 8080 machine operation code, or a macro invocation with optional parameters. The pseudo operations and machine operation codes are described in Section 5. Macro calls are discussed in Section 6.

The operand field of the statement contains an expression formed from constant and label operands, with arithmetic, logical, and relational operations on these operands. Properly formed expressions are detailed in Section 3.

A leading semicolon character denotes the comment field, which contains arbitrary characters until the next carriage return or exclamation point character. MAC reads, lists, and otherwise ignores comment fields. To maintain compatibility with other assemblers, MAC also treats statements that begin with an asterisk (*) in column one as comment lines.

The assembly language program is thus a sequence of statements of the form described above, terminated optionally by an END statement. The assembler ignores all statements following the END.

End of Section 2

Section 3
Forming the Operand

Expressions in the operand field consist of simple operands—labels, constants, and reserved words—combined into properly formed subexpressions by arithmetic and logical operators. MAC carries out expression computation as the assembly proceeds. Each expression produces a 16-bit value during the assembly. The number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate (see the MVI instruction), the absolute value of the operand must fit within an 8-bit field. Instructions for each expression give the restrictions on expression significance.

3.1 Labels

A label is an identifier of a statement. The label's value is determined by the type of statement it precedes. If the label occurs on a statement that generates machine code or reserves memory space, such as a MOV instruction or a DS pseudo operation, then the label is given the value of the program address it labels. If the label precedes an EQU or SET, then the label is given the value that results from evaluating the operand field. In a macro definition, the label is given a text value, a sequence of ASCII characters, that is the body of the macro definition. With the exception of the SET and MACRO pseudo operations, an identifier can label only one statement.

When a nonmacro label appears in the operand field, the assembler substitutes its 16-bit value. This value can then be combined with other operands and operators to form the operand field for an instruction. When a macro identifier appears in the operation field of the statement, the text stored as the value of the macro name is substituted for the name. In this case, the operand field of the statement contains actual parameters. These are substituted for dummy parameters in the body of the macro definition. Later sections give the exact mechanisms for defining, calling, and substituting macro text.

3.2 Numeric Constants

A numeric constant is a 16-bit value in a number base. A trailing radix indicator denotes the base, called the radix of the constant. The radix indicators are

B	binary constant (base 2)
O	octal constant (base 8)
Q	octal constant (base 8)
D	decimal constant (base 10)
H	hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers because the letter O is easily confused with the digit 0. Any numeric constant that does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is composed of a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. Binary constants must be composed of 0 and 1 digits. Octal constants can contain digits in the range 0-7. Decimal constants contain decimal digits. Hexadecimal constants contain decimal digits and hexadecimal digits A through F, corresponding to the decimal numbers 10 through 15.

Note that the leading digit of a hexadecimal constant must be a decimal digit to avoid confusing a hexadecimal constant with an identifier. A leading 0 prevents ambiguity. A constant composed in this manner produces a binary number that can be contained within a 16-bit counter, truncated on the right by the assembler. Like identifiers, embedded $ symbols are allowed within constants to improve readability.

Finally, the radix indicator translates to upper-case if a lower-case letter is encountered. The following examples are valid numeric constants:

```
1234        1234D       1100B         1111$0000$1111$0000B
1234H       0FFFEH      33770         33$77$22Q
3377o       0fe3h       1234d         0ffffh
```

3.3 Reserved Words

Several reserved character sequences have predefined meanings in the operand field of a statement. The names of 8080 registers and their values are given in Table 3-1.

Table 3-1. 8080 Registers and Values

symbol	*value*	*symbol*	*value*
A	7	B	0
C	1	D	2
E	3	H	4
L	5	M	6
SP	6	PSW	6

Lower-case names have the same values as their upper-case equivalents. Machine instructions can also be used in the operand field, resulting in their internal codes. For instructions that require operands, where the operand is a part of the binary bit pattern of the instruction (e.g., MOV A,B), the value of the instruction is the bit pattern of the instruction, with zeros in the optional fields. For example, the statement

```
LXI H,MOV
```

assembles an LXI H instruction with an operand equal to 40H, the value of the MOV instruction with zeros as operands.

When the $ symbol appears in the operand field—not embedded within identifiers and numbers—its value is the address of the beginning of the current instruction. For example, the two statements

```
X:   JMP X
```

and

```
JMP $
```

produce a jump instruction to the current location. As an exception, the $ symbol at the beginning of a logical line can introduce assembly formatting instructions. (See Section 10.)

3.4 String Constants

String constants represent sequences of graphic ASCII characters, enclosed in apostrophes ('). All strings must be fully contained within the current physical line, with the exclamation point (!) character within strings treated as an ordinary string character. Each individual string must not exceed 64 characters in length, or MAC reports an error. The apostrophe character can be included in a string by typing two apostrophes (''). The assembler reads the two apostrophes as a single apostrophe.

Note that particular operation codes can require the string length to be no longer than one or two characters. The LXI instruction, for example, accepts a character string operand of one or two characters. The CPI instruction accepts only a one-character string. The DB instruction, however, allows strings zero through 64 characters long in its list of operands. In the case of single-character strings, the value is the 8-bit ASCII code for the character, without case translation. Two-character strings produce a 16-bit value with the second character as the low-order byte and the first character as the high-order byte. For example, the string constant 'A' is equivalent to 41H. The two-character string 'AB' produces the 16-bit value 4142H. The following are valid strings in MAC statements:

```
'A'   'AB'   'ab'   'c'   '' ''   'she said "hello"'
```

Note: You can use the ampersand (&) character to cause evaluation of dummy arguments within macro expansions inside string quotes. Section 8 details the substitution process.

3.5 Arithmetic, Logical, and Relational Operators

MAC can combine the operands described above in algebraic notation using properly formed operands, operators, and parenthesized expressions. The operators MAC recognizes in the operand field are listed below.

- a+b produces the arithmetic sum of a and b; +b is b.
- a−b produces the arithmetic difference between a and b; −b is 0−b.
- a*b is the unsigned multiplication of a by b.
- a/b is the unsigned division of a by b.
- a MOD b is the remainder after division of a by b.
- a SHL b produces a shifted left by b, with zero right fill.

- a SHR b produces a shifted right by b, with zero left fill.
- NOT b is the bit-by-bit logical inverse of b.
- a EQ b produces true if a equals b, false otherwise.
- a LT b produces true if a is less than b, false otherwise.
- a LE b produces true if a is less than or equal to b, false otherwise.
- a GT b produces true if a is greater than b, false otherwise.
- a GE b produces true if a is greater than or equal to b, false otherwise.
- a AND b produces the bitwise logical AND of a and b.
- a OR b produces the bitwise logical OR of a and b.
- a XOR b produces the logical exclusive OR of a and b.
- HIGH b is identical to b SHR 8 (high-order byte of b).
- LOW b is identical to b AND 0FFH (low-order byte of b).

The letters a and b represent operands that are treated as 16-bit unsigned quantities in the range 0-65535. All arithmetic operators produce a 16-bit unsigned arithmetic result. Relational operators produce a true (0FFFFH) or false (0000H) 16-bit result. Logical operators operate bit-by-bit on their operands producing a 16-bit result of 16 individual bit operations. The HIGH and LOW functions always produce a 16-bit result with a high-order byte of zero. Table 3-2 lists arithmetic, logical, and relational operators.

Table 3-2. Operators

arithmetic	*relational*	*logical*
+	EQ	NOT
–	LT	AND
*	LE	OR
/	GT	XOR
MOD	GE	
SHL	NE	
SHR		

MAC performs all computations during the assembly process as 16-bit unsigned operations, as described above. The resulting expression must fit the operation code in which it is used. For example, the expression used in an ADI (add immediate) instruction must fit into an 8-bit field. Thus, the high-order byte must be zero. If the computed value does not fit the field, the assembler produces a value error for that statement.

As an exception to this rule, negative 8-bit values are allowed in 8-bit fields under the following conditions: if the program attempts to fill an 8-bit field with a 16-bit value that has all ls in the high-order byte, and the sign bit is set, then the high order byte is truncated, and no error is reported. This condition arises when a negative sign is placed in front of a constant. For example, the value -2 is defined and computed as 0-2, producing the 16-bit value 0FFFEH, where the high-order byte (0FFH) contains extended sign bits that are all 1s, and the low-order byte (0FEH) has the sign bit set. The following instructions do not produce value errors in MAC:

```
ADI -1   ADI -15   ADI -127   ADI -128   ADI 0FF80H
```

The following instructions produce value errors:

```
ADI 256   ADI 32768   ADI -129   ADI 0FF7FH
```

The special operator NUL is used in conjunction with macro definition and expansion operations. The NUL operator takes a single operand. NUL must be the last operator in the operand field.

Expressions can be formed from simple operands such as labels, numeric constants, string constants, and machine operation codes, or from fully enclosed parenthesized expressions such as

```
10+20,
10H+37Q,
L1/3,
(L2 + 4) SHR 3,
('a' and 5fh) + '0',
('BB' + B) OR (PSW + M),
(1+ (2+C)) shr (A-(B +1)),
(HIGH A) SHR 3
```

where blanks and tabs are ignored between the operators and operands of the expression.

3.6 Precedence of Operators

MAC assumes operators have a relative precedence of application allowing expressions to be written without nested parentheses. The resulting expression has assumed parentheses that are defined by this relative precedence. The order of application of operators in unparenthesized expressions is listed below. Operators listed first have highest precedence. These are applied first in an unparenthesized expression. Operators listed last have lowest precedence and are applied last. Operators listed on the same line have equal precedence and are applied from left to right as they are encountered in an expression:

```
 *   /   MOD  SHL  SHR
          +  -
EQ  LT   LE   GT   GE   NE
           NOT
           AND
         OR  XOR
        HIGH  LO
```

The following expressions are equivalent:

```
a * b + c produces (a * b) + c
a + b * c produces a + (b * c)
a MOD b * c SHL d produces ((a MOD b) * c) SHL D
a OR b AND NOT c + d SHL e produces
a OR (b AND (NOT (c + (d SHL e))))
```

Balanced parenthesized subexpressions can always override the assumed parentheses. The last expression above can be rewritten to force application of operators in a different order, as shown below:

```
(a OR b) AND (NOT c) + d SHL e
```

resulting in the assumed parentheses

```
(a OR b) AND ((NOT c) + (d SHL e))
```

Note that an unparenthesized expression is well formed only if the expression that results from inserting the assumed parentheses is well formed.

Relational operators can be expressed in either of two forms, as shown in Table 3-3.

Table 3-3. Equivalent Forms of Relational Operators

<	LT
<=	LE
=	EQ
<>	NE
>=	GE
>	GT

End of Section 3

Section 4
Assembler Directives

Assembler directives set labels to specific values during assembly, perform conditional assembly, define storage areas, and specify starting addresses in the program. Each assembler directive is denoted by a pseudo operation that appears in the operation field of the statement. Table 4-1 lists the acceptable pseudo operations.

Table 4-1. Pseudo Operations

Directive	*Meaning*
ORG	sets the program or data origin.
END	terminates the physical program.
EQU	performs a numeric equate.
SET	performs a numeric set or assignment.
IF	begins a conditional assembly.
ELSE	is an alternate to a previous IF.
ENDIF	marks the end of conditional assembly.
DB	defines data bytes or strings of data.
DW	defines words of storage (double bytes).
DS	reserves uninitialized storage areas.
PAGE	defines the listing page size for output.
TITLE	enables page titles and options.

In addition to those listed above, several pseudo operations are used in conjunction with the macro processing facilities. MACRO, EXITM, ENDM, REPT, IRPC, IRP, LOCAL, and MACLIB are reserved words. They are fully described in Sections 7 and 8. The nonmacro pseudo operations are detailed below.

4.1 The ORG Directive

The ORG statement takes the form

```
label ORG expression
```

where label is an optional program label—an identifier followed by an optional colon (:)—and expression is a 16-bit expression consisting of operands defined before the ORG statement. The assembler begins machine code generation at the location specified in the expression. There can be any number of ORG statements within a program. There are no checks to ensure that you are not redefining overlapping memory areas. Note that most programs written for CP/M begin with an ORG 100H statement that causes machine code generation to begin at the base of the CP/M Transient Program Area. Programs assembled with RMAC and linked with LINK-80 do not need an ORG 100H statement. (See Sections 13 and 15.)

If the ORG statement has a label, then the label takes on the value given by the expression. The expression is the next machine code address to assemble. This label can then be used in the operand field of other statements to represent this expression.

4.2 The END Directive

The END statement is optional in an assembly language program; if present, it must be the last statement. All statements following the END are ignored. The two forms of the END statement are

```
label  END
label  END     expression
```

where the label is optional. If the first form is used, the assembly process stops, and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated and becomes the program starting address. This starting address is included in the last record of the Intel format machine code hex file resulting from the assembly. Most CP/M assembly language programs end with the statement

```
END     100H
```

resulting in the default starting address of 100H, the beginning of the Transient Program Area.

4.3 The EQU Directive

The EQU (equate) statement names synonyms for particular numeric values. The directive takes the form:

label EQU expression

The label must be present, and it must not label any other statement. The assembler evaluates the expression and assigns this value to the identifier given in the label field. The identifier is usually a name describing the value in a more human-oriented manner. You can use this name throughout the program as a parameter for certain functions. Suppose, for example, that data received from a teletype appears on an input port, and data is sent to the teletype through the next output port in sequence. The series of equate statements that can define these ports for a particular hardware environment is shown below.

```
TTYBASE    EQU  10H          ;BASE TTY PORT
TTYIN      EQU  TTYBASE      ;TTY DATA IN
TTYOUT     EQU  TTYBASE+1    ;TTY DATA OUT
```

At a later point in the program, the statements that access the teletype could appear as

```
IN     TTYIN       ;READ TTY DATA TO A
OUT    TTYOUT      ;WRITE DATA FROM A
```

making the program more readable than the absolute I/O port addresses. If the hardware environment is later redefined to start the teletype communications ports at 7FH instead of 10H, the first statement need only be changed to

```
TTYBASE    EQU   7FH        ;BASE PORT NUMBER FOR TTY
```

and the program can be reassembled without changing any other statements.

4.4 The SET Directive

The SET statement is similar to the EQU, taking the form

```
label SET expression
```

except that the label, taken as a variable name, can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, unlike the EQU statement, where a label takes on a single value throughout the program, the SET statement can assign different values to a name at different parts of the program. In particular, the SET statement gives the label a value that is valid from the current SET statement to the point where the label occurs on the next SET statement. The use of SET is similar to the EQU, except that SET is used more often to control conditional assembly within macros.

4.5 The IF, ELSE, and ENDIF Directives

The IF, ELSE, and ENDIF directives define a range of assembly language statements to be included or excluded during the assembly process. The IF and ENDIF statements alone can bound a group of statements to be conditionally assembled, as shown in the following example:

```
IF    expression
statement#1
statement#2
        ...
statement#n
ENDIF
```

Upon encountering the IF statement, the assembler evaluates the expression following the IF. All operands in the expression must be defined ahead of the IF statement. If the expression evaluates to a nonzero value, then statement#1 through statement#n are assembled. If the expression evaluates to zero, then the statements are listed but not assembled.

Conditional assembly is often used to write a single generic program that includes a number of possible alternative subroutines or program segments, where only a few of the possible alternatives are to be included in any given assembly. Listings 4-1 and 4-2 give an example of such a program.

Assume that a console device, either a teletype or a CRT, is connected to an 8080 microcomputer through I/O ports. Due to the electronic environment, the current loop teletype is connected through ports 10H and 11H, while the RS-232 CRT is connected through ports 20H and 21H. The program continually loops, reading and writing console characters. The program shown below operates either with a teletype or a CRT, depending on the value of the symbol TTY.

Listing 4-1 shows an assembly for the teletype environment. Listing 4-2 shows the assembly for a CRT-based system. Note that the assembler leaves the leftmost 16 columns blank when statements are skipped due to a false condition.

```
CP/M MACRO ASSEM 2.0      #001    Teletype Echo Program

 FFFF =          TRUE     EQU     OFFFFH   ;DEFINE TRUE
 0000 =          FALSE    EQU     NOT TRUE ;DEFINE FALSE
 FFFF =          TTY      EQU     TRUE     ;SET TTY ON
 0010 =          TTYBASE  EQU     10H      ;BASE OF TTY PORTS
 0020 =          CRTBASE  EQU     20H      ;BASE OF CRT PORTS
                          IF      TTY      ;ASSEMBLE TTY PORTS
                          TITLE   'Teletype Echo Program'
 0010 =          CONIN    EQU     TTYBASE         ;CONSOLE INPUT
 0011 =          CONOUT   EQU     TTYBASE+1       ;CONSOLE OUT
                          ENDIF
                          IF      NOT TTY ;ASSEMBLE CRT PORTS
                          TITLE   'CRT Echo Program'
                 CONIN    EQU     CRTBASE         ;CONSOLE IN
                 CONOUT   EQU     CRTBASE+1       ;CONSOLE OUT
                          ENDIF
                 ;
 0000 DB10       ECHO:    IN      CONIN           ;READ CONSOLE
                                                   CHARACTER
 0002 D311                OUT     CONOUT          ;WRITE CONSOLE
                                                   CHARACTER
 0004 C30000              JMP     ECHO
 0007                     END
```

Listing 4-1. Conditional Assembly with TTY True

```
CP/M MACRO ASSEM 2.0      #001     CRT Echo Program

 FFFF =           TRUE     EQU      OFFFFH     ;DEFINE TRUE
 0000 =           FALSE    EQU      NOT TRUE   ;DEFINE FALSE
 0000 =           TTY      EQU      FALSE      ;SET CRT ON
 0010 =           TTYBASE  EQU      10H        ;BASE OF TTY PORTS
 0020 =           CRTBASE  EQU      20H        ;BASE OF CRT PORTS
                           IF       TTY        ;ASSEMBLE TTY PORTS
                           TITLE    'Teletype Echo Program'
                  CONIN    EQU      TTYBASE         ;CONSOLE INPUT
                  CONOUT   EQU      TTYBASE+1       ;CONSOLE OUT
                           ENDIF
                           IF       NOT TTY    ;ASSEMBLE CRT PORTS
                           TITLE    'CRT Echo Program'
 0020 =           CONIN    EQU      CRTBASE         ;CONSOLE IN
 0021 =           CONOUT   EQU      CRTBASE+1       ;CONSOLE OUT
                           ENDIF
                  ;
 0000 DB20        ECHO:    IN       CONIN      ;READ CONSOLE
                                                CHARACTER
 0002 D321                 OUT      CONOUT     ;WRITE CONSOLE
                                                CHARACTER
 0004 C30000               JMP      ECHO
 0007                      END
```

Listing 4-2. Conditional Assembly with TTY False

The ELSE statement can be used as an alternative to an IF statement. The ELSE statement must occur between the IF and ENDIF statements. The form is

```
IF    expression
statement#1
statement#2
      ...
statement#n
ELSE
statement#n+1
statement#n+2
      ...
statement#m
ENDIF
```

If the expression produces a nonzero (true) value, then statements 1 through n are assembled as before. However, the assembly process skips statements n+1 through m. When the expression produces a zero value (false), MAC skips statements 1 through n and assembles statements n+1 through m. For example, the conditional assembly shown in Listings 4-1 and 4-2 can be rewritten as shown in Listing 4-3.

```
CP/M MACRO ASSEM 2.0      #001     CRT Echo Program

 FFFF =          TRUE     EQU      0FFFFH   ;DEFINE TRUE
 0000 =          FALSE    EQU      NOT TRUE;DEFINE FALSE
 0000 =          TTY      EQU      FALSE    ;SET CRT ON
 0010 =          TTYBASE  EQU      10H      ;BASE OF TTY PORTS
 0020 =          CRTBASE  EQU      20H      ;BASE OF CRT PORTS
                          IF       TTY      ;ASSEMBLE TTY PORTS
                          TITLE    'Teletype Echo Program'
                 CONIN    EQU      TTYBASE          ;CONSOLE INPUT
                 CONOUT   EQU      TTYBASE+1        ;CONSOLE OUT
                          ELSE              ;ASSEMBLE CRT PORTS
                          TITLE    'CRT Echo Program'
 0020 =          CONIN    EQU      CRTBASE          ;CONSOLE IN
 0021 =          CONOUT   EQU      CRTBASE+1        ;CONSOLE OUT
                          ENDIF
                 ;
 0000 DB20       ECHO:    IN       CONIN    ;READ CONSOLE CHARACTER
 0002 D321                OUT      CONOUT   ;WRITE CONSOLE CHARACTER
 0004 C30000              JMP      ECHO
 0007                     END
```

Listing 4-3. Conditional Assembly Using ELSE for Alternate

Properly balanced IF, ELSE, and ENDIF statements can be completely contained within the boundaries of outer encompassing conditional assembly groups. The structure outlined below shows properly nested IF, ELSE, and ENDIF statements:

```
IF      exp#1
group#1
IF      exp#2
group#2
ELSE
group#3
ENDIF
group#4
ELSE
group#5
IF      exp#3
group#6
ENDIF
group#7
ENDIF
```

Groups 1 through 7 are sequences of statements to be conditionally assembled, and exp#1 through exp#3 are expressions that control the conditional assembly. If exp#1 is true, then group#1 and group#4 are always assembled, and groups 5, 6, and 7 are skipped. Further, if exp#1 and exp#2 are both true, then group#2 is also included in the assembly. Otherwise, group#3 is included. If exp#1 produces a false value, groups 1, 2, 3, and 4 are skipped, and groups 5 and 7 are always assembled. If exp#3 is true under these circumstances, then group#6 is also included with 5 and 7. Otherwise, it is skipped in the assembly. A structure similar to this is shown in Listing 4-4, where literal true/false values show conditional assembly selection.

There can be up to eight pending IFs or ELSEs with unresolved ENDIFs at any point in the assembly, but the assembly usually becomes unreadable after two or three levels or nesting. The nesting level restriction also holds, however, for pending IFs and ELSEs during macro evaluation. Nesting level overflow produces an error during assembly.

```
FFFF =          TRUE   EQU    OFFFFH        ;DEFINE TRUE
0000 =          FALSE  EQU    NOT TRUE      ;DEFINE FALSE
                       IF     FALSE
                       MVI    A,1
                       IF     TRUE
                       MVI    A,2
                       ELSE
                       MVI    A,3
                       ENDIF
                       MVI    A,4
                       ELSE
0000 3E05              MVI    A,5
                       IF     TRUE
0002 3E06              MVI    A,6
                       ELSE
                       MVI    A,7
                       ENDIF
0004 3E08              MVI    A,8
                       ENDIF
                       END
```

Listing 4-4. Sample Program Using Nested IF, ELSE, and ENDIF

4.6 The DB Directive

The DB directive defines initialized storage areas in single-precision (byte) format.

The statement form is

label DB e#l, e#2, . . . , e#n

where the label is optional, and e#1 through e#n are either expressions that produce 8-bit values (the high-order eight bits are zeros, or the high-order nine bits are ones), or are ASCII strings no longer than 64 characters each. There is no practical restriction on the number of expressions included on a single source line. The assembler evaluates expressions and places them into the machine code sequentially following the last program address generated.

String characters are similarly placed into memory, starting with the first character and ending with the last character. Strings longer than two characters cannot be used as operands in more complicated expressions. They must stand alone between the commas. Note that ASCII characters are always placed in memory with the high-order (parity) bit reset to zero. Further, recall that there is no translation from lower to upper-case within strings. The optional label can be used to reference the data area throughout the program. The following are examples of valid DB statements:

```
data:       DB   0,1,2,3,4,5,6
            DB   data and 0ffh,5,377Q,1+2+3+4
signon:     DB   'please type your name:',cr,lf,0
            DB   'AB' SHR 8, 'C', 'DE' AND 7FH
            DB   HIGH data, LOW (signon GT data)
```

4.7 The DW Directive

The DW statement is similar to the DB statement except double-precision (two-byte) words of storage are initialized. The form of the DW statement is

label DW e#1, e#2, . . . , e#n

where the label is optional, and e#1 through e#n are expressions that produce 16-bit values. Note that ASCII strings one or two characters long are allowed, but strings longer that two characters are disallowed. In all cases, the data storage is consistent with the 8080 processor; the least significant byte of the expression is stored first in memory, followed by the most significant byte. The following are examples of properly formed DW statements:

```
doub:     DW   0ffefh, doub+4, signon-$,255+255
          DW   'a', 5, 'AB', 'CD', doub LT signon
```

4.8 The DS Directive

The DS statement reserves an area of uninitialized memory and takes the form

```
label   DS   expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the statement sequences:

```
label:     EQU   $                ;CURRENT CODE LOC
           ORG   $+expression     ;MOVE PAST AREA
```

4.9 The PAGE and TITLE Directives

The PAGE and TITLE pseudo operations give you control over the output formatting that is sent to the PRN file or directly to the printer device. The forms for the PAGE statement are

```
PAGE
PAGE  expression
```

If the PAGE statement stands alone, an ASCII CTRL-L (form-feed) is sent to the output file after the PAGE statement has been printed. The PAGE command is often issued directly ahead of major sections of an assembly language program, such as a group of subroutines, to cause the next statement to appear at the top of the following page.

The second form of the PAGE command specifies the output page size. In this case, the expression following the PAGE pseudo operation determines the number of output lines to be printed on each page. If the expression is zero, there are no page breaks. The print file is simply a continuous sequence of annotated output lines. If the expression is nonzero, then the page size is set to the value of the expression. Form-feeds are issued to cause page ejects when this count is reached for each page.

The assembler initially assumes that

```
PAGE 56
```

is in effect, producing a page eject at the beginning of the listing and at each 56-line increment.

The TITLE directive takes the form

```
TITLE string-constant
```

where the string-constant is an ASCII string enclosed in apostrophes, not exceeding 64 characters in length. If a TITLE pseudo operation is given during the assembly, each page of the listing file is prefixed with the title line, preceded by a standard MAC header. The title line thus appears as

```
CP/M MACRO ASSEM n.n   #ppp     string-constant
```

where n.n is the MAC version number, #ppp is the page number in the listing, and string-constant is the string given in the TITLE pseudo operation. MAC initially assumes that the TITLE operation is not in effect. When specified, the title line and the blank line following the title are not included in the line count for the page. No more than one TITLE statement is included in a program. Similarly, only one PAGE statement with the expression option is included.

If a TITLE statement is included, and the Symbol Table is being appended to the PRN file (see Section 10), then the SYM file also contains the title at the beginning of the symbol listing with page breaks given by either the default or specified value of the PAGE statement.

4.10 A Sample Program Using Pseudo Operations

The program in Listing 4-5 demonstrates the pseudo operations available in MAC. The sample program, called TYPER, operates in the CP/M environment by selecting one of three messages for output at the console. This program is created using the ED program, assembled using MAC, and then placed into COM file format using the CP/M LOAD function. After these steps have been accomplished, TYPER executes at the Console Command Processor level of CP/M by typing one of the commands:

```
TYPER A
TYPER B
TYPER C
```

to select message A, B, or C for printing. The TYPER program loads under the CCP and jumps to the label START where the 8080 stack is initialized. The TYPER program then prints its sign-on message:

```
'typer' version 1.0
```

The program then retrieves the first character typed at the console following the command TYPER. This character should be A, B, or C. If one of these letters is not specified, then TYPER reboots the CP/M system to give control back to the CCP. If a valid letter is provided, TYPER selects one of the three messages (MESS@A, MESS@B, or MESS@C) and prints it at the console before returning to CP/M.

The TITLE and PAGE statements produce a title at the beginning of each page; page size is 33 lines, excluding the title lines. Form-feeds are suppressed. A number of EQU statements at the beginning improve program readability. Note that throughout the program the exclamation point allows several simple assembly language statements on the same line. Although multiple statements make the program more compact, they often decrease the overall readability of the source program. Note also that the program terminates without the END statement. The END statement is necessary only if a starting address is specified. The END statement is often included, however, to maintain compatibility with other assemblers.

The DB statements labeled by SIGNON contain simple strings of characters and expressions that produce single-byte values. The DW statement following TABLE defines the base address of each string, corresponding to A, B, and C. Finally, the DS statement at the end of the program reserves space for the stack defined within the TYPER program.

```
      CP/M MACRO ASSEM 2.0 #001 Typer Program

                          TITLE   'Typer Program'
                          PAGE    33
                  ;       PRINT THE MESSAGE SELECTED BY THE INPUT  COMMAND A,B, OR C
000A =           VERS     EQU     10       ;VERSION NUMBER N.N
0000 =           BOOT     EQU     0000H    ;REBOOT ENTRY POINT
0005 =           BDOS     EQU     0005H    ;BDOS ENTRY POINT
005C =           TFCB     EQU     005CH    ;DEFAULT FILE CONTROL BLOCK (GET A,B, OR C)
0002 =           WCHAR    EQU     2        ;WRITE CHARACTER FUNCTION
000D =           CR       EQU     0DH      ;CARRIAGE RETURN CHARACTER
000A =           LF       EQU     0AH      ;LINE FEED CHARACTER
0010 =           STKSIZ   EQU     16       ;SIZE OF LOCAL STACK (IN DOUBLE BYTES)
                  ;
0100                      ORG     100H     ;ORIGIN AT BASE OF TPA
0100 C31201               JMP     START    ;JUMP PAST THE MESSAGE SUBROUTINE
                  ;
                  WMESSAGE:
                          ;WRITE THE STRING AT THE ADDRESS GIVEN BY HL 'TIL 00
0103 7EB7C8                       MOV A,M! ORA A! RZ ;RETURN IF AT 00
0106 5F0E02E5                     MOV E,A! MVI C,WCHAR! PUSH H ;READY TO PRINT
010A CD0500E1                     CALL BDOS! POP H ;CHARACTER PRINTED, GET NEXT
010E 23C30301                     INX H! JMP WMESSAGE
                  ;
                  START:  ;ENTER HERE FROM THE CCP, RESET TO LOCAL STACK
0112 31C101               LXI     SP,STACK          ;SET TO LOCAL STACK
0115 213701               LXI     H,SIGNON          ;WRITE THE MESSAGE
0118 CD0301               CALL    WMESSAGE          ;'TYPER' VERSION N.N
                  ;
011B 3A5D00               LDA     TFCB+1           ;GET FIRST CHAR TYPED AFTER NAME
011E D641                 SUI     'A'              ;NORMALIZE TO 0,1,2
0120 FE03                 CPI     TABLEN           ;COMPARE WITH THE TABLE LENGTH
0122 D20000               JNC     BOOT             ;REBOOT IF NOT VALID
                  ;
                  ;       COMPUTE INDEX INTO ADDRESS TABLE BASED ON A'S VALUE
```

Listing 4-5. TYPER Program Listing

```
      CP/M MACRO ASSEM 2.0   #002   Typer Program

0125 5F                 MOV     E,A            ;LOW ORDER INDEX
0126 1600               MVI     D,0            ;EXTENDED TO DOUBLE PRECISION
0128 214D01             LXI     H,TABLE        ;BASE OF THE TABLE TO INDEX
012B 19                 DAD     D              ;SINGLE PRECISION INDEX
012C 19                 DAD     D              ;DOUBLE PRECISION INDEX
012D 5E                 MOV     E,M            ;LOW ORDER BYTE TO E
012E 23                 INX     H
012F 56                 MOV     D,M            ;HIGH ORDER MESSAGE ADDRESS TO DE
0130 EB                 XCHG                   ;READY FOR PRINTOUT
0131 CD0301             CALL    WMESSAGE       ;MESSAGE WRITTEN TO CONSOLE
0134 C30000             JMP     BOOT           ;REBOOT, GO BACK TO CCP LEVEL
               ;
               ;        DATA AREAS
               SIGNON:
0137 2774797065         DB       '''typer'' version '
0147 312E30             DB       VERS/10+'0', '.', VERS MOD 10 +'0'
014A 0D0A00             DB       CR,LF,0 ;END OF MESSAGE
               ;
               TABLE:   ;OF MESSAGE BASE ADDRESSES
014D 5301670182         DW       MESS@A,MESS@B,MESS@C
0003 =         TABLEN   EQU      ($-TABLE)/2     ;LENGTH OF TABLE
               ;
0153 7468697320MESS@A:  DB       'this is message a',CR,LF,0
0167 796F752073MESS@B:  DB       'you selected b this time',CR,LF,0
0182 7468697320MESS@C:  DB       'this message comes out for c',CR,LF,0
               ;
 01A1                   DS       STKSIZ*2        ;RESERVES AREA FOR STACK
               STACK:
```

Listing 4-5. (continued)

End of Section 4

Section 5
Operation Codes

Operation codes, found in the operation field of the statement, form the principal components of assembly language programs. MAC accepts all the standard mnemonics for the Intel 8080 microcomputer. These standard mnemonics are detailed in the *8080 Assembly Language Programming Manual*, published by Intel. Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued by the assembler. The individual operators are listed briefly in the following sections. See the Intel documentation for exact operator details. In this section, operation codes are categorized for discussion; a sample assembly shows the hexadecimal codes produced for each operation. The following notation is used throughout:

e3	represents a 3-bit value in the range 0-7 that usually takes one of the predefined register values A, B, C, D, H, L, M, SP, or PSW
e8	represents an 8-bit value in the range 0-255; signed 8-bit values are also allowed in the range −128 through +127
e16	represents a 16-bit value in the range 0-65535

where e3, e8, and e16 can be formed from an arbitrary combination of operands and operators in a well-formed expression. In some cases, the operands are restricted to particular values within the range, such as the PUSH instruction.

5.1 Jumps, Calls, and Returns

In some cases, the condition flags are tested to determine whether or not to take the jump, call, or return. The forms are shown below. The jump instructions are

```
JMP e16        JNZ e16        JZ e16
JNC e16        JC e16         JPO e16
JPE e16        JP e16         JM e16
```

The call instructions are

```
CALL e16       CNZ e16        CZ e16
CNC e16        CC e16         CPO e16
CPE e16        CP e16         CM e16
```

The return instructions are

```
RET            RNZ            RZ
RNC            RC             RPO
RPE            RP             RM
```

The restart instruction takes the form:

```
RST e3
```

and performs exactly the same function as the instruction CALL e3*8 except that RST e3 requires only one byte of memory.

Listing 5-1 shows the hexadecimal codes for each instruction, along with a short comment on each line describing the function of the instruction.

```
CP/M MACRO ASSEM 2.0    #001   8080 JUMPS, CALLS, AND RETURNS

                               TITLE   '8080 JUMPS, CALLS, AND RETURNS'
                        ;
                        ;      JUMPS ALL REQUIRE A 16-BIT OPERAND
0000 C31B00                    JMP     L1       ;JUMP UNCONDITIONALLY TO LABEL
0003 C25C00                    JNZ     L1+'A'   ;JUMP ON NON ZERO TO LABEL
0006 CA0001                    JZ      100H     ;JUMP ON ZERO CONDITION TO LABEL
0009 D21F00                    JNC     L1+4     ;JUMP ON NO CARRY TO LABEL
000C DA4142                    JC      'AB'     ;JUMP ON CARRY TO LABEL
000F E21700                    JPO     $+8      ;JUMP ON PARITY ODD TO LABEL
0012 EA0D00                    JPE     L1/2     ;JUMP ON EVEN PARITY TO LABEL
0015 F24100                    JP      GAMMA    ;JUMP ON POSITIVE RESULT TO LABEL
0018 FA1B00                    JM      LOW L1   ;JUMP ON MINUS TO LABEL
                        L1:
                        ;
                        ;      CALL OPERATIONS ALL REQUIRE A 16-BIT OPERAND
001B CD3600                    CALL    S1       ;CALL SUBROUTINE UNCONDITIONALLY
001E C43800                    CNZ     S1+X     ;CALL SUBROUTINE IF NON ZERO FLAG
0021 CC0001                    CZ      100H     ;CALL SUBROUTINE IF ZERO FLAG
0024 D43A00                    CNC     S1+4     ;CALL SUBROUTINE IF NO CARRY FLAG
0027 DC0000                    CC      S1 MOD 3;CALL SUBROUTINE IF CARRY FLAG
002A E43200                    CPO     $+8      ;CALL SUBROUTINE IF PARITY ODD
002D EC0900                    CPE     S1-$     ;CALL SUBROUTINE IF PARITY EVEN
0030 F44100                    CP      GAMMA    ;CALL SUBROUTINE IF POSITIVE
0033 FC4100                    CM      GAM$MA   ;CALL SUBROUTINE IF MINUS FLAG
                        S1:
                        ;
                        ;      PROGRAMMED RESTART (RST) REQUIRES 3-BIT OPERAND
                        ;      (RST X IS EQUIVALENT TO CALL X*8)
0036 C7                        RST     0        ;RESTART TO LOCATION 0
0037 DF                        RST     X+1
                        ;
                        ;      RETURN INSTRUCTIONS HAVE NO OPERAND
0038 C9                        RET              ;RETURN FROM SUBROUTINE
0039 C0                        RNZ              ;RETURN IF NON ZERO
003A C8                        RZ               ;RETURN IF ZERO FLAG SET
003B D0                        RNC              ;RETURN IF NO CARRY FLAG
003C D8                        RC               ;RETURN IF CARRY FLAG SET
003D E0                        RPO              ;RETURN IF PARITY IS ODD
003E E8                        RPE              ;RETURN IF PARITY IS EVEN
003F F0                        RP               ;RETURN IF POSITIVE RESULT
0040 F8                        RM               ;RETURN IF MINUS FLAG SET
                       ;
0002 =                 X       EQU     2
                       GAMMA:
0041                           END
```

Listing 5-1. Assembly Showing Jumps, Calls, Returns, and Restarts

5.2 Immediate Operand Instructions

Several instructions load single- or double-precision registers or single-precision memory locations with constant values. Other instructions perform immediate arithmetic or logical operations on the accumulator (register A). The move immediate instruction takes the form:

```
MVI e3,e8
```

where e3 is the register to receive the data given by the value e8. The expression e3 must produce a value corresponding to one of the registers A, B, C, D, E, H, L, or the memory location M, which is addressed by the HL register pair.

The accumulator immediate operations take the form:

```
ADI e8    ACI e8    SUI e8    SBI e8
ANI e8    XRI e8    ORI e8    CPI e8
```

where the operation is always performed on the accumulator using the immediate data value given by the expression e8.

The load extended immediate instructions take the form:

```
LXI e3,e16
```

where e3 designates the register pair to receive the double-precision value given by e16. The expression e3 must produce a value corresponding to one of the double-precision register pairs B, D, H, or SP.

Listing 5-2 shows the accumulator immediate operations in an assembly language program and briefly describes each instruction.

```
     CP/M MACRO ASSEM 2.0    #001    IMMEDIATE OPERAND INSTRUCTIONS

                               TITLE   'IMMEDIATE OPERAND INSTRUCTIONS'
                     ;
                     ;         MVI USES A REGISTER (3-BIT) OPERAND AND 8-BIT DATA
0000 06FF                      MVI     B,255    ;MOVE IMMEDIATE A,B,C,D,E,H,L,M
                     ;
                     ;         ALL REMAINING IMMEDIATE OPERATIONS USE A REGISTER
0002 C601                      ADI     1        ;ADD IMMEDIATE TO A W/O CARRY
0004 CEFF                      ACI     0FFH     ;ADD IMMEDIATE TO A WITH CARRY
0006 D613                      SUI     L1+3     ;SUBTRACT FROM A W/O BORROW (CARRY)
0008 DE10                      SBI     LOW L1   ;SUBTRACT FROM A WITH BORROW (CARRY)
000A E602                      ANI     $ AND 7  ;LOGICAL AND WITH IMMEDIATE DATA
000C EE3C                      XRI     1111$00B ;LOGICAL XOR WITH IMMEDIATE DATA
000E F6FD                      ORI     -3       ;LOGICAL OR WITH IMMEDIATE DATA
                     L1:
0010                           END
```

Listing 5-2. Assembly Using Immediate Operand Instructions

5.3 Increment and Decrement Instructions

The 8080 set includes instructions for incrementing or decrementing single- and double-precision registers. The instruction forms for single-precision registers are

INR e3   DCR e3

where e3 produces a value corresponding to register A, B, C, D, H, L, or M. These registers correspond to the byte value at the memory location addressed by HL. The double-precision instructions are

INX e3   DCX e3

where e3 must be equivalent to one of the double-precision register pairs B, D, H, or SP.

Listing 5-3 shows a sample assembly language program using both single- and double-precision increment and decrement operations.

```
        CP/M MACRO ASSEM 2.0    #001    INCREMENT AND DECREMENT INSTRUCTIONS

                              TITLE  'INCREMENT AND DECREMENT INSTRUCTIONS'
                      ;
                      ;         INSTRUCTIONS REQUIRE REGISTER (3-BIT) OPERAND
 0000 1C                        INR     E       ;BYTE INCREMENT A,B,C,D,E,H,L,M
 0001 3D                        DCR     A       ;BYTE DECREMENT A,B,C,D,E,H,L,M
 0002 33                        INX     SP      ;16-BIT INCREMENT B,D,H,SP
 0003 0B                        DCX     B       ;16-BIT DECREMENT B,D,H,SP
 0004                           END
```

Listing 5-3. Assembly Containing Increment and Decrement Instructions

5.4 Data Movement Instructions

A number of 8080 instructions move data from memory to the CPU and from the CPU to memory. Data movement instructions also include a number of register-to-register move operations. The single-precision move register instruction takes the form:

MOV e3, e3′

where the e3 and e3′ expressions each produce a single-precision register A, B, C, D, E, H, L, or M, where the M register corresponds to the memory location addressed by HL. The register named by e3 always receives the 8-bit value given by the register expression e3′. The instruction is often read as move to register e3 from register e3′. The instruction MOV B,H would thus be read as move to register B from register H. Note that the instruction MOV M,M is not allowed.

The single-precision load and store extended operations take the form:

LDAX e3   STAX e3

where e3 is a register expression that must produce one of the double-precision register pairs B or D. The 8-bit value in register A is either loaded from (LDAX) or stored to (STAX) the memory location addressed by the specified register pair.

The load and store direct instructions operate on either the A register for single-precision operations, or on the HL register pair for double-precision operations. Load and store direct instructions take the form:

```
LHLD e16     SHLD e16     LDA e16     STA e16
```

where e16 is an expression that produces the memory address to obtain (LHLD, LDA) or store (SHLD, STA) the data value.

The stack pop and push instructions perform double-precision load and store operations, with the 8080 stack as the implied memory address. The forms are

```
POP e3     PUSH e3
```

where e3 must evaluate to one of the double-precision register pairs PSW, B, D, or H.

The input and output instructions are also in this category, even though they receive and send their data to the electronic environment external to the 8080 processor. The input instruction reads data to the A register; the output instruction sends data from the A register. In both cases, the data port is given by the data value that follows the instruction. The forms are

```
IN e8       OUT e8
```

A set of instructions transfers double-precision values between registers and the stack. These instructions are

```
XTHL     PCHL     SPHL     XCHG
```

Listing 5-4 lists these instructions in an assembly language program and briefly describes them.

```
    CP/M MACRO ASSEM 2.0    #001    DATA/MEMORY/REGISTER MOVE OPERATIONS

                              TITLE   'DATA/MEMORY/REGISTER MOVE OPERATIONS'
                       ;
                       ;      THE MOV INSTRUCTION REQUIRES TWO REGISTER OPERANDS
                       ;      (3-BITS) SELECTED FROM A,B,C,D,E,H, OR M (M,M INVALID)
0000 78                       MOV     A,B      ;MOVE DATA TO FIRST REGISTER FROM
                                               ;SECOND
                       ;
                       ;      LOAD/STORE EXTENDED REQUIRE REGISTER PAIR B OR D
0001 0A                       LDAX    B        ;LOAD ACCUM FROM ADDRESS GIVEN BY BC
0002 12                       STAX    D        ;STORE ACCUM TO ADDRESS GIVEN BY DE
                       ;
                       ;      LOAD/STORE DIRECT REQUIRE MEMORY ADDRESS
0003 2A1900                   LHLD    D1       ;LOAD HL DIRECTLY FROM ADDRESS D1
0006 221B00                   SHLD    D1+2     ;STORE HL DIRECTLY TO ADDRESS D1+2
0009 3A1900                   LDA     D1       ;LOAD THE ACCUMULATOR FROM D1
000C 326400                   STA     D1 SHL 2;STORE THE ACCUMULATOR TO D1 SHL 2
                       ;
                       ;      PUSH AND POP REQUIRE PSW OR REGISTER PAIR FROM B,D,H
000F F1                       POP     PSW      ;LOAD REGISTER PAIR FROM STACK
0010 C5                       PUSH    B        ;STORE REGISTER PAIR TO THE STACK
                       ;
                       ;      INPUT/OUTPUT INSTRUCTIONS REQUIRE 8-BIT PORT NUMBER
0011 DB06                     IN      X+2      ;READ DATA FROM PORT NUMBER TO A
0013 D3FE                     OUT     0FEH     ;WRITE DATA TO THE SPECIFIED PORT
                       ;
                       ;      MISCELLANEOUS REGISTER MOVE OPERATIONS
0015 E3                       XTHL             ;EXCHANGE TOP OF STACK WITH HL
0016 E9                       PCHL             ;PC RECEIVES THE HL VALUE
0017 F9                       SPHL             ;SP RECEIVES THE HL VALUE
0018 EB                       XCHG             ;EXCHANGE DE AND HL
                       ;
                       ;      END OF INSTRUCTION LIST
0019                   D1:    DS      2        ;DOUBLE WORD TEMPORARY
001B                          DS      2        ;ANOTHER TEMPORARY
0004 =                 X      EQU     4        ;LITERAL VALUE
001D                          END
```

Listing 5-4. Assembly Using Various Register/Memory Moves

5.5 Arithmetic Logic Unit Operations

The 8080 set includes instructions that operate between the accumulator and single-precision registers, including operations on the A register and carry flag. The accumulator/register instructions are

ADD e3	ADC e3	SUB e3	SBB e3
ANA e3	XRA e3	ORA e3	CMP e3

where e3 produces a value corresponding to one of the single-precision registers A, B, C, D, E, H, L, or M, where the M register is the memory location addressed by the HL register pair.

The accumulator/carry operations given below operate upon the A register, or carry bit, or both.

DAA	CMA	STC	CMC
RLC	RRC	RAL	RAR

The function of each instruction is listed in the comment line shown in Listing 5-5.

```
      CP/M MACRO ASSEM 2.0    #001    ARITHMETIC LOGIC UNIT OPERATIONS

                              TITLE   'ARITHMETIC LOGIC UNIT OPERATIONS'
                       ;
                       ;      ASSUME OPERATION WITH ACCUMULATOR AND REGISTER,
                       ;      WHICH MUST PRODUCE A, B, C, D, E, H, L, OR M
                       ;
0000 80                       ADD     B       ;ADD REGISTER TO A W/O CARRY
0001 8D                       ADC     L       ;ADD TO A WITH CARRY INCLUDED
0002 94                       SUB     H       ;SUBTRACT FROM A W/O BORROW
0003 99                       SBB     B+1     ;SUBTRACT FROM A WITH BORROW
0004 A1                       ANA     C       ;LOGICAL AND WITH REGISTER
0005 AF                       XRA     A       ;LOGICAL XOR WITH REGISTER
0006 B0                       ORA     B       ;LOGICAL OR WITH REGISTER
0007 BC                       CMP     H       ;COMPARE REGISTER, SETS FLAGS
                       ;
                       ;      DOUBLE ADD CHANGES HL PAIR ONLY
0008 09                       DAD     B       ;DOUBLE ADD B,D,H,SP TO HL
                       ;
                       ;      REMAINING OPERATIONS HAVE NO OPERANDS
0009 27                       DAA     ;DECIMAL ADJUST REGISTER A USING LAST OP
000A 2F                       CMA     ;COMPLEMENT THE BITS OF THE A REGISTER
000B 37                       STC     ;SET THE CARRY FLAG TO 1
000C 3F                       CMC     ;COMPLEMENT THE CARRY FLAG
000D 07                       RLC     ;8-BIT ACCUM ROTATE LEFT, AFFECTS CY
000E 0F                       RRC     ;8-BIT ACCUM ROTATE RIGHT, AFFECTS CY
000F 17                       RAL     ;9-BIT CY/ACCUM ROTATE LEFT
0010 1F                       RAR     ;9-BIT CY/ACCUM ROTATE RIGHT
                       ;
0011                          END
```

Listing 5-5. Assembly Showing ALU Operations

The double-precision add instruction performs a 16-bit addition of a register pair (B, D, H, or SP) into the 16-bit value in the HL register pair. This addition produces the 16-bit (unsigned) sum of the two values. The sum is placed into the HL register pair. The form is

DAD e3

5.6 Control Instructions

The four remaining instructions in the 8080 set are control instructions. These take the forms:

```
HLT
DI
EI
NOP
```

They stop the processor (HLT), enable the interrupt system (EI), disable the interrupt system (DI), or perform a no-operation (NOP).

End of Section 5

Section 6
An Introduction to Macro Facilities

The fundamental difference between the Digital Research ASM and MAC assemblers is that ASM provides only the facilities for assembling 8080 operation codes, and MAC includes a powerful macro processing facility. MAC implements the industry standard Intel macro definition, which includes the following pseudo operations.

Macro definitions allow groups of instructions to be stored and substituted in the source program as the macro names are encountered. Definitions and macro calls can be nested; symbols can be constructed through concatenation using the special & operator, and locally defined symbols can be created using the LOCAL pseudo operation. Macro parameters can be formed to pass arbitrary strings of text to a specific macro for substitution during expansion.

The MACLIB (macro library) feature allows the programmer to define a set of macros, equates, and sets and automatically includes them in a program. A macro library can contain an instruction set for another central processor that is not directly supported by the MAC built-in mnemonics. The macro library can also include general purpose input/output macros used in programs that operate in the CP/M environment to perform peripheral or disk I/O functions.

IRPC, IRP, and REPT pseudo operations repeat source statements under control of a count or list of characters or items to be substituted each time the assembler rereads the statements. This feature is particularly useful in generating groups of assembly language statements with similar structure, such as a set of File Control Blocks where only the filetype is changed in each statement.

To illustrate the power of macro facility, consider the macro library shown in Listing 6-1, which resides in a disk file called MSGLIB.LIB. This macro library contains macro definitions that have standard instruction sequences for program startup, message typeout, and program termination. The program shown in Listing 6-2 provides an example of the use of this macro library. The assembly shown in Listing 6-2 lists both the macro calls and the statements in macro expansions that generate machine code. The statements marked by + in Listing 6-2 are generated from the macro calls. The remaining statements are a part of the calling program.

The macro call

```
ENTCCP 10
```

in Listing 6-2 shows a specific expansion of ENTCCP (enter from CCP). ENTCCP is defined in Listing 6-1. The macro call causes MAC to retrieve the definition—the text between MACRO and ENDM in Listing 6-1—and substitute this text following the macro call in Listing 6-2. Upon entry to the program from CCP, this macro saves the stack pointer (SP) into a variable called @ENTSP for later retrieval. The stack pointer is then reset to a local area for the remainder of the program execution.

The size of the local stack is defined by the macro parameter named in the macro definition as SSIZE (see Listing 6-1), and filled in at the call with the value 10. The ENTCCP macro reserves space for a local stack of SSIZE=10 double bytes (2*10 bytes) and, after setting up the stack, branches around this reserved area to continue the program execution.

```
;       SIMPLE MACRO LIBRARY FOR MESSAGE TYPEOUT
REBOOT  EQU     0000H   ;WARM START ENTRY POINT
TPA     EQU     0100H   ;TRANSIENT PROGRAM AREA
BDOS    EQU     0005H   ;SYSTEM ENTRY POINT
TYPE    EQU     2       ;WRITE CONSOLE CHARACTER FUNCTION
CR      EQU     0DH     ;CARRIAGE RETURN
LF      EQU     0AH     ;LINE FEED
;
;MACRO  DEFINITIONS
;
CHROUT MACRO            ;WRITE A CONSOLE CHARACTER FROM REGISTER A
       MVI      C,TYPE  ;;TYPE FUNCTION
       CALL     BDOS    ;;ENTER THE BDOS TO WRITE THE CHARACTER
       ENDM
;
TYPEOUT         MACRO   ?MESSAGE        ;TYPE LITERAL MESSAGE AT CONSOLE
       LOCAL    PASTSUB ;;JUMP PAST SUBROUTINE INITIALLY
       JMP      PASTSUB
MSGOUT:         ;;THIS SUBROUTINE PRINTS THE MESSAGE STARTING AT HL 'TIL 00
       MOV      E,M     ;;NEXT CHARACTER TO E
       MOV      A,E     ;;TO ACCUM TO TEST FOR 00
       ORA      A       ;;=00?
       RZ               ;;RETURN IF END OF MESSAGE
       INX      H       ;;OTHERWISE MOVE TO NEXT CHARACTER AND PRINT
       PUSH     H       ;;SAVE MESSAGE ADDRESS
       CHROUT
       POP      H       ;;RECALL MESSAGE ADDRESS
       JMP      MSGOUT  ;;FOR ANOTHER CHARACTER
PASTSUB:
;
;;     REDEFINE THE TYPEOUT MACRO AFTER THE FIRST INVOCATION
TYPEOUT        MACRO    ??MESSAGE
       LOCAL   TYMSG    ;;LABEL THE LOCAL MESSAGE
       LOCAL   PASTM
       LXI     H,TYMSG  ;;ADDRESS THE LITERAL MESSAGE
       CALL    MSGOUT   ;;CALL THE PREVIOUSLY DEFINED SUBROUTINE
       JMP     PASTM
;;     INCLUDE THE LITERAL MESSAGE AT THIS POINT
TYMSG: DB      'FROM CONSOLE:  &??MESSAGE',CR,LF,0
;;     ARRIVE HERE TO CONTINUE THE MAINLINE CODE
PASTM: ENDM
       TYPEOUT <?MESSAGE>
       ENDM
;
```

Listing 6-1. A Sample Macro Library

```
ENTCCP MACRO   SSIZE    ;ENTER PROGRAM FROM CCP, RESERVE 2*SSIZE STACK LOCS
       LOCAL   START    ;;AROUND THE STACK
       LXI     H,0
       DAD     SP       ;;SP VALUE IN HL
       SHLD    @ENTSP   ;;ENTRY SP
       LXI     SP,@STACK;;SET TO LOCAL STACK
       JMP     START
       IF      NUL SSIZE
       DS      32       ;;DEFAULT 16 LEVEL STACK
       ELSE
       DS      2*SSIZE
       ENDIF
@STACK:        ;;LOW END OF STACK
@ENTSP:        DS      2        ;;ENTRY SP
START: ENDM
;
RETCCP MACRO   ;RETURN TO CONSOLE PROCESSOR
       LHLD    @ENTSP   ;;RELOAD CCP STACK
       SPHL
       RET              ;;BACK TO THE CCP
       ENDM
;
ABORT  MACRO   ;ABORT THE PROGRAM
       JMP     REBOOT
       ENDM
;
;      END OF MACRO LIBRARY
```

Listing 6-1. (continued)

```
CP/M MACRO ASSEM 2.0    001   SAMPLE MESSAGE OUTPUT MACRO

                        TITLE  'SAMPLE MESSAGE OUTPUT MACRO'
                 ;
                 ;
                        MACLIB MSGLIB ;INCLUDE THE MACRO LIBRARY
0100                    ORG    TPA    ;ORIGIN AT THE TRANSIENT AREA
                 ;      USE THE MACRO LIBRARY TO TYPE TWO MESSAGES
                        ENTCCP 10     ;ENTER PROGRAM, RESERVE 10 LEVEL STACK
0100+210000             LXI    H,0
0103+39                 DAD    SP
0104+222101             SHLD   @ENTSP
0107+312101             LXI    SP,@STACK
010A+C32301             JMP    ??0001
010D+                   DS     2*10
0121+            @ENTSP:       DS     2
                        TYPEOUT <THIS IS THE FIRST MESSAGE>
0123+C33401             JMP     ??0002
0126+5E                 MOV     E,M
0127+B7                 ORA     A
0128+C8                 RZ
0129+23                 INX     H
012A+E5                 PUSH    H
012B+0E02               MVI     C,TYPE
012D+CD0500             CALL    BDOS
0130+E1                 POP     H
0131+C32601             JMP     MSGOUT
0134+213D01             LXI     H,??0003
0137+CD2601             CALL    MSGOUT
013A+C36701             JMP     ??0004
013D+46524F4D20??0003:          DB    'FROM CONSOLE: THIS IS THE FIRST MESSAGE',CR,LF,0
                        TYPEOUT <THIS IS THE SECOND MESSAGE>
0167+217001             LXI     H,??0005
016A+CD2601             CALL    MSGOUT
016D+C39B01             JMP     ??0006
0170+46524F4D20??0005:          DB    'FROM CONSOLE: THIS IS THE SECOND MESSAGE',CR,LF,0
                        TYPEOUT <THIS IS THE THIRD MESSAGE>
019B+21A401             LXI     H,??0007
019E+CD2601             CALL    MSGOUT
01A1+C3CE01             JMP     ??0008
01A4+46524F4D20??0007:          DB    'FROM CONSOLE: THIS IS THE THIRD MESSAGE',CR,LF,0
                        RETCCP  ;RETURN TO THE CONSOLE COMMAND PROCESSOR
01CE+2A2101             LHLD    @ENTSP
01D1+F9                 SPHL
01D2+C9                 RET
01D3                    END
```

Listing 6-2. A Sample Assembly Using the MACLIB Facility

Consider also the special macro statements used in Listing 6-1 within the body of the ENTCCP macro. The LOCAL statement defines the label START within the macro body. Each LOCAL statement causes the macro assembler to construct a unique symbol starting with ?? each time it is encountered. Thus, multiple macro calls reference unique labels that do not interfere with one another. ENTCCP also contains a conditional assembly statement that uses the NUL operator; this tests whether a macro parameter has been supplied or not. In this case, the ENTCCP macro can be started by

```
ENTCCP
```

with no actual parameter, resulting in a default stack size of 32 bytes. The following sections give exact details and examples.

The TYPEOUT macro is a more complicated example of macro use. Note that this macro contains a redefinition of itself within the macro body. The structure of TYPEOUT is

```
TYPEOUT   MACRO   ?MESSAGE
          ...
TYPEOUT   MACRO   ??MESSAGE
          ...
          ENDM
          ...
          ENDM
```

where the outer definition of TYPEOUT completely encloses the inner definition. The outer definition is active upon the first invocation of TYPEOUT, but upon completion, the nested inner definition becomes active.

To see the use of such a nested structure, consider the TYPEOUT macro. Each time it starts, TYPEOUT prints the message sent as an actual parameter at the console device. The typeout process, however, can be easily handled with a short subroutine. Upon the first invocation, include the subroutine inline. Then simply call this subroutine on subsequent invocations of TYPEOUT. Thus, the outer definition of TYPEOUT defines the utility subroutine and then redefines itself, so that the subroutine is called, rather than including another copy of the utility subroutine.

Note that macro definitions are stored in the symbol table area of the assembler, so each macro reduces the remaining free space. MAC allows double semicolon comments to indicate that the comment itself is to be ignored and not stored with the macro. Thus, comments with a single semicolon are stored with the macro and appear in each expansion; comments with two preceding semicolons are listed only when the macro is defined.

Listing 6-2 gives three examples of TYPEOUT invocations, with three messages that are sent as actual parameters. Note that the LOCAL statement causes a unique label to be created (??0002) in the place of PASTSUB, which is used to branch around the utility subroutine included inline between addresses 0126H and 0133H. The utility subroutine is then called, followed by another jump around the console message, also included inline. However, subsequent invocations of TYPEOUT use the previously included utility subroutine to type their messages.

Although this example concentrates all macro definitions in a separate macro library, macros are often defined in the mainline (.ASM) source program. In fact, many programs that use macros do not use the external macro library facility at all.

The rest of this manual examines many applications of macros. Macro facilities can simplify the programming task by abstracting from the primitive assembly language levels. That is, you can define macros that provide more generalized functions that are allowed at the pure assembly language level, such as macro languages for a given application, improved control facilities, and general purpose operating systems interfaces. The remainder of this manual first introduces the individual macro forms, and then presents several uses of the macro facilities in realistic applications.

End of Section 6

Section 7
Inline Macros

The simplest macro facilities involve the REPT (repeat), IRPC (indefinite repeat character), and IRP (indefinite repeat) macro groups. All these forms cause the assembler to reread portions of the source program under control of a counter or list of textual substitutions. These groups are listed below in order of increasing complexity.

7.1 The REPT-ENDM Group

The REPT-ENDM group is written as a sequence of assembly language statements starting with the REPT pseudo operation and terminated by an ENDM pseudo operation. The form is

```
label: REPT expression
       statement-1
       statement-2
       . . .
       statement-n
label: ENDM
```

where the labels are optional. The expression following the REPT is evaluated as a 16-bit unsigned count of the number of times that the assembler is to read and process statements 1 through n, enclosed within the group.

Listing 7-1 shows an example of the use of the REPT group. In this case, the REPT-ENDM group generates a short table of the byte values 5, 4, 3, 2, and 1. Upon entry to the REPT, the value of NXTVAL is 5. This is taken as the repeat count, even though NXTVAL changes within the REPT. The macro lines that do not generate machine code are not listed in the repetition, while the lines that do generate code are listed with a + sign after the machine code address. Full macro tracing is optional, however, using assembly parameters. (See Section 10.)

```
      CP/M MACRO ASSEM 2.0    #001    SAMPLE REPT STATEMENT

0100                    ORG     100H    ;BASE OF TRANSIENT AREA
                        TITLE   'SAMPLE REPT STATEMENT'
                ;       THIS PROGRAM READS INPUT PORT 0 AND INDEXES
                        INTO A TABLE
                ;       BASED ON THIS VALUE. THE TABLE VALUE IS FETCHED
                        AND SENT
                ;       TO OUTPUT PORT 0
                ;
0005 =          MAXVAL  EQU     5       ;LARGEST VALUE TO PROCESS
0100 DB00       RLOOP:  IN      0       ;READ THE PORT VALUE
0102 FE05               CPI     MAXVAL  ;TOO LARGE?
0104 D20001             JNC     RLOOP   ;IGNORE INPUT IF INVALID
0107 211401             LXI     H,TABLE ;ADDRESS BASE OF TABLE
010A 5F                 MOV     E,A     ;LOW ORDER INDEX TO E
010B 1600               MVI     D,0     ;HIGH ORDER 00 FOR INDEX
010D 19                 DAD     D       ;HL HAS ADDRESS OF ELEMENT
010E 7E                 MOV     A,M     ;FETCH TABLE VALUE FOR OUTPUT
010F D300               OUT     0       ;SEND TO THE OUTPUT PORT AND LOOP
0111 C30001             JMP     RLOOP   ;FOR ANOTHER INPUT
                ;
                ;       GENERATE A TABLE OF VALUES MAXVAL,MAXVAL-1,...,1
0005 #          NXTVAL  SET     MAXVAL  ;START COUNTER AT MAXVAL
                        TABLE:  REPT    NXTVAL
                        DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
                NXTVAL  SET     NXTVAL-1;;AND DECREMENT FILL VALUE
                        ENDM
0114+05                 DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
0115+04                 DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
0116+03                 DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
0117+02                 DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
0118+01                 DB      NXTVAL  ;FILL ONE (MORE) ELEMENT
0119                    END
```

Listing 7-1. A Sample Program Using the REPT Group

If a label appears on the REPT statement, its value is the first machine code address that follows. This REPT label is not reread on each repetition of the loop. The optional label on the ENDM is reread on each iteration; thus constant labels, not generated through concatenation or with the LOCAL pseudo operation, generate phase errors if the repetition count is greater than 1.

Properly nested macros, including REPTs, can occur within the body of the REPT-ENDM group. Further, nested conditional assembly statements are also allowed, with the added feature that conditionals beginning within the repeat group automatically terminate upon reaching the end of the macro expansion. Thus, IF and ELSE pseudo operations are not required to have their corresponding ENDIF when they begin within the repeat group, although the ENDIF is allowed.

7.2 The IRPC-ENDM Group

Similar to the REPT group, the IRPC-ENDM group causes the assembler to reread a bounded set of statements, taking the form:

```
label: IRPC identifier,character-list
       statement-1
       statement-2
       . . .
       statement-n
label: ENDM
```

where the optional labels obey the same conventions as in the REPT-ENDM group. The identifier is any valid assembler name, not including embedded $ separators. Character list denotes a string of characters terminated by a delimiter (space, tab, end-of-line, or comment).

The IRPC controls the reread process as follows: the statement sequence is read once for each character in the character list. On each repetition, a character is taken from the character list and associated with the controlling identifier, starting with the first and ending with the last character in the list. Thus, an IRPC header of the form

```
IRPC ?X,ABCDE
```

rereads the statement sequence that follows (to the balancing ENDM) five times, once for each character in the list ABCDE. On the first iteration, the character A is associated with the identifier ?X. On the fifth iteration, the letter E is associated with the controlling identifier.

On each iteration, the macro assembler substitutes any occurrence of the controlling identifier by the associated character value. Using the preceding IRPC header, an occurrence of ?X in the bounds of the IRPC-ENDM group is replaced by the character A on the first iteration, and by E on the last iteration.

The programmer can use the controlling identifier to construct new text strings within the body of the IRPC by using the special concatenation operator, denoted by an ampersand (&) character. Again using the preceding IRPC header, the macro assembler replaces LAB&?X with LABA on the first iteration. LABE is produced on the final iteration. The concatenation feature is most often used to generate unique label names on each iteration of the IRPC reread process.

The controlling identifier is not usually substituted within string quotes because the controlling identifier can appear as a part of a quoted message. Thus, the macro assembler performs substitution of the controlling identifier when it is preceded or followed by the ampersand operator. Further, all alphabetics outside string quotes are translated to upper-case, but no case translation occurs within string quotes. So the controlling identifier must not only be preceded or followed by the concatenation operator within strings, but it must also be typed in upper-case.

Listings 7-2a and 7-2b illustrate the use of the IRPC-ENDM group. Listing 7-2a shows the original assembly language program, before processing by the macro assembler. The program is typed in both upper- and lower-case. Listing 7-2b shows the output from the macro assembler, with the lower-case alphabetics translated to upper-case. Three IRPC groups are shown in this example. The first IRPC uses the controlling identifier reg to generate a sequence of stack push operations that save the double-precision registers BC, DE, and HL. The lines generated by this group are marked by a + sign following the machine code address.

```
;       construct a data table
;
;       save relevant registers
enter:  irpc    reg,bdh
        push    reg     ;;save reg
        endm
;
;       initialize a partial ascii table
        irpc    c,1Ab$?@
data&c: db      '&C'
        endm
;
;       restore registers
        irpc    reg,hdb
        pop     reg     ;;recall reg
        endm
        ret
        end
```

Listing 7-2a. Original (.ASM) File with IRPC Example

```
                ;       CONSTRUCT A DATA TABLE
                ;
                ;       SAVE RELEVANT REGISTERS
                ENTER:  IRPC    REG,BDH
                        PUSH    REG     ;;SAVE REG
                        ENDM
0000+C5                 PUSH    B
0001+D5                 PUSH    D
0002+E5                 PUSH    H
                ;
                ;       INITIALIZE A PARTIAL ASCII TABLE
                        IRPC    C,1AB$?@
                DATA&C: DB      '&C'
                        ENDM
0003+31         DATA1:  DB      '1'
0004+41         DATAA:  DB      'A'
0005+42         DATAB:  DB      'B'
0006+24         DATA$:  DB      '$'
0007+3F         DATA?:  DB      '?'
0008+40         DATA@:  DB      '@'
                ;
                ;       RESTORE REGISTERS
                        IRPC    REG,HDB
                        POP     REG     ;;RECALL REG
                        ENDM
0009+E1                 POP     H
000A+D1                 POP     D
000B+C1                 POP     B
000C C9                 RET
000D                    END
```

Listing 7-2b. Resulting (.PRN) File with IRPC Example

The second IRPC shown in Listing 7-2a uses the controlling identifier C to generate a number of single-byte constants with corresponding labels. Although the controlling variable was typed in lower-case, it has been translated to upper-case during assembly. The string '&C' occurs within the group and, because the controlling variable is enclosed in string quotes, it must occur next to an ampersand operator and be typed in upper-case for the substitution to occur properly. On each iteration of the IRPC, a label is constructed through concatenation, and a DB is generated with the corresponding character from the character list.

Substitution of the controlling identifier by its associated value can cause infinite substitution if the controlling identifier is the same as the character from the character list. For this reason, the macro assembler performs the substitution and then moves along to read the next segment of the program, rather than rereading the substituted text for another possible occurrence of the controlling identifier. Thus, an IRPC of the form

```
IRPC    C,1AC$?@
```

produces

```
DATAC:   DB    'C'
```

in place of the DB statement at the label DATAA in Listing 7-2b.

The last IRPC restores the previously saved double-precision registers and performs the exact opposite function from the IPRC at the beginning of the program.

When no characters follow the identifier portion of the IRPC header, the group of statements is read once, and the controlling identifier is deleted when it is read. It is replaced by the null string.

7.3 The IRP-ENDM Group

The IRP (indefinite repeat) functions like the IRPC, except that the controlling identifier can take on a multiple character value. The form of the IRP group is

```
label: IRP   identifier,1<4c1-1,c1-2,...,c1-n1>2
       statement-1
       statement-2
       . . .
       statement-m
       label: ENDM
```

where the optional labels obey the conventions of the REPT and IRPC groups. The identifier controls the iteration, as follows. On the first iteration, the character list given by c1-1 is substituted for the identifier wherever the identifier occurs in the bounded statement group (statements 1 through m). On the second iteration, c1-2 becomes the value of the controlling identifier. Iteration continues in this manner until the last character list, denoted by c1-n, is encountered and processed. Substitution of values for the controlling identifier is subject to the same rules as in the IRPC.

Note rules for substitution within strings and concatenation of text using the ampersand & operator. Controlling identifiers are always ignored within comments.

Listing 7-3 gives several examples of IRP groups. The first occurrence of the IRP in Listing 7-3 is a typical use of this facility—to generate a jump vector at the beginning of a program or subroutine. The IRP assigns label names (INITIAL, GET, PUT, and FINIS) to the controlling identifier ?LAB and produces a jump instruction for each label by rereading the IRP group, substituting the actual label for the formal name on each iteration.

The second occurrence of the IRP group in Listing 7-3 points out substitution conventions within strings for both IRPC and IRP groups. The controlling identifier IS takes on the values A-ROSE and ? on the two iterations of the IRP group, respectively.

The controlling identifier is replaced by the character lists in the two occurrences of &IS and IS& inside the string quotes because they are both adjacent to the ampersand operator. is& is not replaced because the controlling identifier is typed in lower-case, and there is no automatic translation to upper-case within strings. The occurrences of IS within the comments are not substituted.

The last IRP group shows the effects of an empty character list. The value of the controlling identifier becomes the null string of symbols and, in the cases where ?X is replaced, produces the statement:

```
DB   ''
```

DB produces no machine code and is therefore not listed in the macro expansion. The three statements

```
DB '?x'   DB '?X'   DB '&'
```

appear in the expansions because the '?x' is typed in lower-case and thus is not replaced. The '?X' does not appear next to an ampersand in the string and is thus not replaced. In the last case, only one of the double ampersands is absorbed in the '&&?X&' string. Here, the two ampersands surrounding ?X are removed because they occur immediately next to the controlling identifier within the string.

Substitution rules outside of string quotes and comments are much less complicated; the controlling identifier is replaced by the current character-list value whenever it occurs in any of the statements within the group. The ampersand operator can be placed before or after the controlling identifier to cause the preceding or following text to be concatenated.

The actual forms for the character lists (cl-1 through cl-n) are more general than stated here. In particular, bracket nesting is allowed, and escape sequences allow delimiters to be ignored. The exact details of character list forms are discussed in the macro parameter sections.

```
                  ;       CREATE A JUMP VECTOR USING THE IRP GROUP
                          IRP     ?LAB,<INITIAL,GET,PUT,FINIS>
                          JMP     ?LAB    ;;GENERATE THE NEXT JUMP
                          ENDM
0000+C30C00               JMP     INITIAL
0003+C34300               JMP     GET
0006+C34600               JMP     PUT
0009+C34900               JMP     FINIS
                  ;
                  ;       INDIVIDUAL CASES
                  INITIAL:
000C 211200               LXI     H,CHRS
000F C35100               JMP     ENDCASE
                  CHRS:   IRP     IS,<A-ROSE,?>
                          DB      '&IS IS IS&'    ;IS IS &IS
                          DB      '&IS isn''t is&'
                          ENDM
0012+412D524F53           DB      'A-ROSE IS A-ROSE'      ;IS IS &IS
0022+412D524F53           DB      'A-ROSE isn''t is&'
0032+3F20495320           DB      '? IS ?'        ;IS IS &IS
0038+3F2069736E           DB      '? isn''t is&'
                  ;
0043 C35100       GET:    JMP     ENDCASE
                  ;
0046 C35100       PUT:    JMP     ENDCASE
                  ;
0049 C35100       FINIS:  JMP     ENDCASE
                          IRP     ?X,<>
                          DB      '?x'
                          DB      '?X'
                          DB      '&?X'
                          DB      '&?X&'
                          DB      '&&?X&'
                          ENDM
004C+3F78                 DB      '?x'
004E+3F58                 DB      '?X'
0050+26                   DB      '&'
                  ENDCASE:
0051 C9                   RET
0052                      END
```

Listing 7-3. A Sample Program Using IRP

7.4 The EXITM Statement

The EXITM pseudo operation can occur within the body of a macro. Upon encountering the EXITM statement, the macro assembler aborts expansion of the current macro level. The EXITM pseudo operation occurs in the context

```
        macro-heading
        statement-1
        . . .
 label: EXITM
        . . .
        statement-n
        ENDM
```

where the label is optional, and macro-heading denotes the REPT, IRPC, or IRP group heading as described above. The EXITM statement can also be used with the MACRO group, as discussed in later sections.

The EXITM statement usually occurs within the scope of a surrounding conditional assembly operation. If the EXITM occurs in the scope of a false conditional test, the statement is ignored, and macro expansion continues. If the EXITM occurs within the scope of a true conditional, the expansion stops where the EXITM is encountered. Assembly statement processing continues after the ENDM of the group aborted by the EXITM statement.

Two examples of the EXITM statement are shown in Listing 7-4. This listing shows two IRPCs used to generate DB statements up to eight characters long. These IRPCs might occur within the context of another macro definition, such as in the generation of CP/M File Control Block (FCB) names. In both cases, the variable LEN counts the number of filled characters. If the count reaches eight characters, the EXITM statement is assembled under a true condition, and the IRPC stops expansion.

The first IRPC generates the entire string SHORT because the length of the character list is less than eight characters. Each evaluation of LEN = 8 produces a false value, and the EXITM is skipped. This IRPC terminates by exhausting the character list through its five repetitions.

The second IRPC stops generation at the eighth character of the list LONGSTRING when the conditional LEN EQ 8 produces a true value, resulting in assembly of the EXITM statement. Note that = and EQ are equivalent operators. The EXITM causes immediate termination of the expansion process.

The second IRPC also contains a conditional assembly without the balancing ENDIF. In this case, the ENDIF is not required because the conditional assembly begins within the macro body. The ENDM serves the dual purpose of terminating unmatched IFs and marking the physical end of the macro body.

```
                ;       SAMPLE USE OF THE EXITM STATEMENT WITH THE IRPC MACRO
                ;
                ;       THE FOLLOWING IRPC FILLS AN AREA OF MEMORY WITH AT MOST
                ;       EIGHT BYTES OF DATA:
                ;
0000 #          LEN     SET     0       ;INITIALIZE LENGTH TO 0
                        IRPC    N,SHORT
                        DB      '&N'
                LEN     SET     LEN+1
                        IF      LEN = 8
                        EXITM           ;STOP MACRO IF AREA IS FULL
                        ENDIF
                        ENDM
0000+53                 DB      'S'
0001+48                 DB      'H'
0002+4F                 DB      'O'
0003+52                 DB      'R'
0004+54                 DB      'T'
                ;
                ;
                ;       THE FOLLOWING MACRO PERFORMS EXACTLY THE SAME FUNCTIONS AS
                ;       SHOWN ABOVE, BUT ABORTS EXPANSION WHEN LENGTH EXCEEDS 8
                ;
0000 #          LEN     SET     0       ;INITIALIZE LENGTH COUNTER
                        IRPC    N,LONGSTRING
                        DB      '&N'
                LEN     SET     LEN+1
                        IF      LEN EQ 8
                        EXITM
                        ENDM
0005+4C                 DB      'L'
0006+4F                 DB      'O'
0007+4E                 DB      'N'
0008+47                 DB      'G'
0009+53                 DB      'S'
000A+54                 DB      'T'
000B+52                 DB      'R'
000C+49                 DB      'I'
                ;
000D                    END
```

Listing 7-4. Use of the EXITM Statement in Macro Processing

7.5 The LOCAL Statement

It is often useful to generate labels for jumps or data references unique on each repetition of a macro. This facility is available through the LOCAL statement. The LOCAL statement takes the form:

```
       macro-heading
label: LOCAL      id-1,id-2,. . .,id-n
       . . .
       ENDM
```

where the label is optional, macro-heading is a REPT, IRPC, or IRP heading, already discussed, or a MACRO heading as discussed in following sections, and id-1 through id-n represent one or more assembly language identifiers that do not contain embedded $ separators. The LOCAL statement must occur within the body. It should appear immediately following the macro header to be compatible with the standard Intel macro facility.

Upon encountering the LOCAL statement, the assembler creates a new frame of the form

```
??nnnn
```

for association with each identifier in the LOCAL list, where nnnn is a four-digit decimal value assigned in ascending order starting at 0001. Whenever the assembler encounters one of the identifiers in the list, the corresponding created name is substituted in its place. Substitution occurs according to the same rules as those for the controlling identifier in the IRPC and IRP groups.

Avoid the use of labels that begin with the two characters ??, so that no conflicting names accidentally occur. Symbols that begin with ?? are not usually included in the sorted symbol list at the end of assembly. (See Section 10 to override this default.) A total of 9999 LOCAL labels can be generated in any assembly. An overflow error occurs if more generations are attempted.

Listing 7-5a shows an example of a program using the LOCAL statement to generate both data references and jump addresses. This program uses the CP/M operating system to print a series of four generated messages, as shown in the output from the program in Listing 7-5b.

The program begins with equates that define the operating system primary entry point, along with names for the nongraphic ASCII characters CR (carriage return) and LF (line-feed). The REPT statement that follows contains a LOCAL statement with the identifiers X and Y. These identifiers are used throughout the body of the REPT group.

On the first iteration, X's value becomes ??0001, the first generated label; Y's value becomes ??0002. The substitution for X and Y within the generated strings follows the rules stated for controlling identifiers in previous sections.

Upon completion, four messages are generated along with four CALLs to the PRINT subroutine. At each call to PRINT, the message address is present in the DE register pair. The subroutine loads the print string function number into register C (C=9) and calls the operating system to print the string value.

```
0100                    ORG     100H     ;BASE OF THE TRANSIENT AREA
0005 =          BDOS    EQU     5        ;BDOS ENTRY POINT
000D =          CR      EQU     0DH      ;CARRIAGE RETURN (ASCII)
000A =          LF      EQU     0AH      ;LINE FEED (ASCII)
                ;
                ;       SAMPLE PROGRAM SHOWING THE USE OF 'LOCAL'
                ;
                        REPT    4        ;REPEAT GENERATION 4 TIMES
                        LOCAL   X,Y      ;;GENERATE TWO LABELS
                        JMP     Y        ;JUMP PAST THE MESSAGE
                X:      DB      'print x=&X, y=&Y',CR,LF,'$'
                Y:      LXI     D,X      ;READY PRINT STRING
                        CALL    PRINT
                        ENDM
0100+C31E01             JMP     ??0002  ;JUMP PAST THE MESSAGE
0103+7072696E74??0001: DB      'print x=??0001, y=??0002',CR,LF,'$'
011E+110301     ??0002: LXI     D,??0001        ;READY PRINT STRING
0121+CD9101             CALL    PRINT
0124+C34201             JMP     ??0004  ;JUMP PAST THE MESSAGE
0127+7072696E74??0003: DB      'print x=??0003, y=??0004',CR,LF,'$'
0142+112701     ??0004: LXI     D,??0003        ;READY PRINT STRING
0145+CD9101             CALL    PRINT
0148+C36601             JMP     ??0006  ;JUMP PAST THE MESSAGE
014B+7072696E74??0005: DB      'print x=??0005, y=??0006',CR,LF,'$'
0166+114B01     ??0006: LXI     D,??0005        ;READY PRINT STRING
0169+CD9101             CALL    PRINT
016C+C38A01             JMP     ??0008  ;JUMP PAST THE MESSAGE
016F+7072696E74??0007: DB      'print x=??0007, y=??0008',CR,LF,'$'
018A+116F01     ??0008: LXI     D,??0007        ;READY PRINT STRING
018D+CD9101             CALL    PRINT
0190 C9                 RET
                ;
0191 0E09       PRINT:  MVI     C,9
0193 CD0500             CALL    BDOS
0196 C9                 RET
0197                    END
```

Listing 7-5a. Assembly Program Using the LOCAL Statement

```
print x=??0001, y=??0002
print x=??0003, y=??0004
print x=??0005, y=??0006
print x=??0007, y=??0008
```

Listing 7-5b. Output from Program in Listing 7-5a

Upon completion of the program, control returns to the Console Command Processor (CCP) for further operations. This program uses the default stack passed by the CCP. About 16 levels are available. This example is primarily intended to show operation of the LOCAL statement. Consult the CP/M documentation for BDOS interface conventions to follow this example completely.

End of Section 7

Section 8 Definition and Evaluation of Stored Macros

The stored macro facility of MAC allows you to name a sequence of assembly language prototype statements to be included at selected places throughout the assembly process. Macro parameters can be supplied in various forms at the point of expansion which are substituted as the prototype statements are reread. These parameters tailor the macro expansion to a particular case.

Although similar in concept to subroutine definition and call, macro processing is purely textual manipulation at assembly time. That is, macro definitions cause source text to be saved in the assembler's internal tables, and any expansion involves manipulating and rereading the saved text.

You can combine macro features in various ways to greatly enhance the available facilities. Specifically, you can

- easily manipulate generalized data definitions
- define macros for generalized operating systems interface
- define simplified program control structures
- support nonstandard instruction sets, such as the Z80®

Finally, well-designed macros for an application can achieve a measure of machine independence.

8.1 The MACRO-ENDM Group

The prototype statements for a stored macro are given in the macro body enclosed by the MACRO and ENDM pseudo operations, taking the general form

```
macname     MACRO     d-1,d-2,. . .,d-n
            statement-1
            statement-2
            . . .
            statement-m
label:      ENDM
```

where the macname is any nonconflicting assembly language identifier; d-1 through d-n constitutes a (possibly empty) list of assembly identifiers without embedded $ separators, and statement-1 through statement-m are the macro prototype statements. The identifiers denoted by d-1 through d-n are called dummy parameters for this macro. Although they must be unique within the macro body, dummy parameters can be identical to any program identifiers outside the macro body without causing a conflict. The prototype statements can contain any properly balanced assembly language statements or groups, including nested REPTs, IRPCs, MACROs, and IFs.

The prototype statements are read and stored in the assembler's internal tables under the name give by macname. They are not processed until the macro is expanded. The following section gives the expansion process.

The label preceding the ENDM is optional.

8.2 Calling a Macro

The macro text stored through a MACRO-ENDM group can be brought out for processing through a statement of the form

```
label:  macname    a-1,a-2,. . .,a-n
```

where the label is optional, and macname has previously occurred as the identifier on a MACRO heading. The actual parameters a-1 through a-n are sequences of characters separated by commas and terminated by a comment or end-of-line.

Upon recognition of the macname, the assembler first pairs off each dummy parameter in the MACRO heading (d-1 through d-n) with the actual parameter text (a-1 through a-n). The assembler associates the first dummy parameter with the first actual parameter (d-1 is paired with a-1), the second dummy with the second actual, and so forth until the list is exhausted. If more actuals are provided than dummy parameters, the extras are ignored. If fewer actuals are provided, then the extra dummy parameters are associated with the empty string (a text string of zero length). The value of a dummy parameter is not a numeric value, but is instead a textual value consisting of a sequence of zero or more ASCII characters.

After each dummy parameter is assigned an actual textual value, the assembler rereads and processes the previously stored prototype statements and substitutes each occurrence of a dummy parameter by its associated actual textual value, according to the same rules as the controlling identifier in an IRPC or IRP group.

Listings 8-1 and 8-2 provide examples of macro definitions and invocations. Listing 8-1 begins with the definition of three macros, SAVE, RESTORE, and WCHAR. The SAVE macro contains prototype statements that save the principal CPU registers (PUSH PSW, B, D, and H). The RESTORE macro restores the principal registers (POP H, D, B, and PSW). The WCHAR macro contains the statements necessary to write a single character at the console using a CP/M BDOS call.

The occurrence of the SAVE macro definition between MACRO and ENDM causes the assembler to read and save the PUSHs, but does not assemble the statements into the program. Similarly, the statements between the RESTORE MACRO and the corresponding ENDM are saved, as are the statements between the WCHAR MACRO and ENDM statements. The fact that the assembler is reading the macro definition is indicated by the blank columns in the leftmost 16 columns of the output listing.

Referring to Listing 8-1, note that machine code generation starts following the SAVE macro call. The prototype statements that were previously stored are reread and assembled, with a + between the machine code address and the generated code to indicate that the statements are being recalled and assembled from a macro definition. The SAVE macro has no dummy parameters in the definition, so no actual parameters are required at the point of invocation.

The SAVE call is immediately followed by an expansion of the WCHAR macro. The WCHAR macro, however, has one dummy parameter, called CHR, which is listed in the macro definition header. This dummy parameter represents the character to pass to the BDOS for printing. In the first expansion of the WCHAR macro, the actual parameter H becomes the textual value of the dummy parameter CHR. Thus, the WCHAR macro expands with a substitution of the dummy parameter CHR by the value H. The CHR is within string quotes, so it is typed in upper-case and preceded by the ampersand operator. Following the reference to WCHAR, the prototype statements are listed with the + sign to indicate that they are generated by the macro expansion.

```
0100                    ORG     100H     ;BASE OF TRANSIENT AREA
0005 =          BDOS    EQU     5        ;BDOS ENTRY POINT
0002 =          CONOUT  EQU     2        ;CHARACTER OUT FUNCTION
                ;
                SAVE    MACRO            ;SAVE ALL CPU REGISTERS
                        PUSH    PSW
                        PUSH    B
                        PUSH    D
                        PUSH    H
                        ENDM
                ;
                RESTORE MACRO            ;RESTORE ALL REGISTERS
                        POP     H
                        POP     D
                        POP     B
                        POP     PSW
                        ENDM
                ;
                WCHAR   MACRO   CHR      ;WRITE CHR TO CONSOLE
                        MVI     C,CONOUT          ;;CHAR OUT FUNCTION
                        MVI     E,'&CHR'          ;;CHAR TO SEND
                        CALL    BDOS
                        ENDM
                ;
                ;       MAIN PROGRAM STARTS HERE
                        SAVE             ;SAVE REGISTERS UPON ENTRY
0100+F5                 PUSH    PSW
0101+C5                 PUSH    B
0102+D5                 PUSH    D
0103+E5                 PUSH    H
                        WCHAR   H        ;SEND 'H' TO CONSOLE
0104+0E02               MVI     C,CONOUT
0106+1E48               MVI     E,'H'
0108+CD0500             CALL    BDOS
                        WCHAR   I        ;SEND 'I' TO CONSOLE
010B+0E02               MVI     C,CONOUT
010D+1E49               MVI     E,'I'
010F+CD0500             CALL    BDOS
                        RESTORE          ;RESTORE CPU REGISTERS
0112+E1                 POP     H
0113+D1                 POP     D
0114+C1                 POP     B
0115+F1                 POP     PSW
0116 C9                 RET              ;RETURN TO CCP
0117                    END
```

Listing 8-1. Example of Macro Definition and Invocation

The second invocation of WCHAR is similar to the first except that the dummy parameter CHR is assigned the textual value I, causing generation of a MVI E, 'I' for this case.

After the listing of the second WCHAR expansion, the RESTORE macro starts, causing generation of the POP statements to restore the register state. The RESTORE is followed by a RET to return to the CCP following the character output.

This program saves the registers upon entry, typing the two characters HI at the console, restoring the registers, and then returning to the Console Command Processor. The SAVE and RESTORE macros are used here for illustration and are not required for interface to the CCP, since all registers are assumed to be invalid upon return from a user program. Further, this program uses the CCP stack throughout. This stack is only eight levels deep.

Listing 8-2 shows another macro for printing at the console. In this case, the PRINT macro uses the operating system call that prints the entire message starting at a particular address until the $ symbol is encountered. The PRINT macro has a slightly more complicated structure: two dummy parameters must be supplied in the invocation. The first parameter, called N, is a count of the number of carriage return line-feeds to send after the message is printed. The second parameter, called MESSAGE, is the ASCII string to print that must be passed as a quoted string in the invocation.

The LOCAL statement within the macro generates two labels denoted by PASTM and MSG. When the macro expands, substitutions occur for the two dummy parameters by their associated actual textual values, and for PASTM and MSG by their sequentially generated label values. The macro definition contains prototype statements that branch past the message (to PASTM) that is included inline following the label MSG. The message is padded with N pairs of carriage return line-feed sequences, followed by the $ that marks the end of the message. The string address is then sent to the BDOS for printing at the console.

Listing 8-2 includes two invocations of the PRINT macro. The invocation sends two actual parameters: the textual value 2 is associated with the dummy N, followed by a quoted string associated with the dummy parameter MSG. The second actual parameter includes the string quotes as a part of the textual value. The generated message is preceded by a jump instruction and followed by N = 2 carriage return line-feed pairs.

The second invocation of the PRINT macro is similar to the first, except that the REPT group is executed N = 0 times, resulting in no carriage return line-feed pairs.

Similar to Listing 8-1, the program of Listing 8-2 uses the Console Command Processor's eight-level stack for the BDOS calls. When the program executes, it types the two messages, separated by two lines, and returns to the CCP.

```
0100                    ORG     100H    ;BASE OF THE TPA
                ;
0005 =          BDOS    EQU     5       ;BDOS ENTRY POINT
0009 =          PMSG    EQU     9       ;PRINT 'TIL $ FUNCTION
000D =          CR      EQU     0DH     ;CARRIAGE RETURN
000A =          LF      EQU     0AH     ;LINE FEED
                ;
                PRINT   MACRO   N,MESSAGE
                ;;      PRINT MESSAGE, FOLLOWED BY N CRLF'S
                LOCAL   PASTM,MSG
                JMP     PASTM   ;;JUMP PAST MSG
                MSG:    DB      MESSAGE ;;INCLUDE TEXT TO WRITE
                REPT    N       ;;REPEAT CR LF SEQUENCE
                DB      CR,LF
                        ENDM
                DB      '$'     ;;MESSAGE TERMINATOR
                PASTM:  LXI     D,MSG   ;;MESSAGE ADDRESS
                        MVI     C,PMSG  ;;PRINT FUNCTION
                        CALL    BDOS
                        ENDM
                ;
                        PRINT   2,'The rain in Spain goes'
0100+C31E01             JMP     ??0001
0103+5468652072??0002:  DB      'The rain in Spain goes'
0119+0D0A               DB      CR,LF
011B+0D0A               DB      CR,LF
011D+24                 DB      '$'
011E+110301     ??0001: LXI     D,??0002
0121+0E09               MVI     C,PMSG
0123+CD0500             CALL    BDOS
                        PRINT   0,'mainly down the drain.'
0126+C34001             JMP     ??0003
0129+6D61696E6C??0004:  DB      'mainly down the drain.'
013F+24                 DB      '$'
0140+112901     ??0003: LXI     D,??0004
0143+0E09               MVI     C,PMSG
0145+CD0500             CALL    BDOS
0148 C9                 RET
```

Listing 8-2. Sample Message Printout Macro

8.3 Testing Empty Parameters

The NUL operator is specifically designed to allow testing of null parameters. Null parameters are actual parameters of length zero. NUL is used as a unary operator. NUL produces a true value if its argument is of length zero and a false value if the argument has a length greater than zero. Thus the operator appears in the context of an arithmetic expression as:

... NUL argument

where the ellipses (...) represent an optional prefixing arithmetic expression, and argument is the operand used in the NUL test. The NUL differs from other operators because it must appear as the last operator in the expression. This is because the NUL operator absorbs all remaining characters in the expression until the following comment or end-of-line is found. Thus, the expression

```
X GT Y AND NUL XXX
```

is valid because NUL absorbs the argument XXX, producing a false value in the scan for the end-of-line. The expression

```
X GT Y AND NUL M +Z)
```

is deceiving but nevertheless valid, even though it appears to be an unbalanced expression. In this case, the argument following the NUL operator is the entire sequence of characters M + Z). This sequence is absorbed by the NUL operator in scanning for the end-of-line. The value of NUL M + Z) is false because the sequence is not empty.

Listing 8-3 gives several examples of the use of NUL in a program. In the first case, NUL returns true because there is an empty argument following the operator. Thus, the true case is assembled, as indicated by the machine code to the left, and the false case is ignored. Similarly, the second use of NUL in Listing 8-3 produces a false value because the argument is nonempty. Both uses of NUL, however, are contrived examples, because NUL is only useful within a macro group, as shown in the definition of the NULMAC macro.

NULMAC consists of a sequence of three conditional tests that demonstrate the use of NUL in checking empty parameters. In each of the tests, a DB is assembled if the argument is not empty and skipped otherwise. Seven invocations of NULMAC follow its definition, giving various combinations of empty and nonempty actual parameters.

In the first case, NULMAC has no actual parameters. Thus all dummy parameters (A, B, and C) are assigned the empty sequence. As a result, all three conditional tests produce false results because both A and B are empty; B&C concatenates two empty sequences, producing an empty sequence as a result.

The second invocation of NULMAC provides only one actual parameter, XXX, assigned to the dummy parameter A. B and C are both assigned the empty sequence. Thus only the DB for the first conditional test is assembled.

```
                              IF      NUL
 0000 7472756520              DB      'true case'
                              ELSE
                              DB      'false case'
                              ENDIF
                      ;
                              IF      NUL XXX
                              DB      'xxx is nul'
                              ELSE
 0009 7878782069              DB      'xxx is not nul'
                              ENDIF
                      ;
                      NULMAC  MACRO   A,B,C
                              IF      NOT NUL A
                              DB      'a = &A is not nul'
                              ENDIF
                              IF      NOT NUL B
                              DB      'b = &B is not nul'
                              ENDIF
                              IF      NOT NUL B&C
                              DB      'bc = &B&C  is not nul'
                              ENDM
                      ;
                              NULMAC
                              NULMAC  XXX
 0017+61203D2058              DB      'a = XXX is not nul'
                              NULMAC  ,XXX
 0029+62203D2058              DB      'b = XXX is not nul'
 003B+6263203D20              DB      'bc = XXX  is not nul'
                              NULMAC  XXX,,YYY
 004F+61203D2058              DB      'a = XXX is not nul'
 0061+6263203D20              DB      'bc = YYY  is not nul'
                              NULMAC  ,,YYY
 0075+6263203D20              DB      'bc = YYY  is not nul'
                              NULMAC  ,,,
                              NULMAC  ,'','',
 0089+6263203D20              DB      'bc = ''''  is not nul'
 009C                         END
```

Listing 8-3. Sample Program Using the NUL Operator

The third case is similar to the second, except that the actual parameters for A and C are omitted. Thus, the second and third conditionals both test NOT NUL XXX, which is true because B has the value XXX, and B&C produces the value XXX as well.

The fourth invocation of NULMAC skips the actual parameter for B but supplies values for both A and C. Thus, the first and third test result in true values; the second conditional group is skipped.

The fifth invocation provides an actual parameter only for C. As a result, only the third conditional is true because B&C produces the sequence YYY.

The sixth invocation produces exactly the same result as the first because all three actual parameters are empty.

The final expansion of NULMAC in Listing 8-3 shows a special case of the NUL operator. The expression

```
NUL ''
```

where the two apostrophes are in juxtaposition, produces the value true, even though there are two apostrophe symbols on the line following NUL and before the end-of-line. The value of A is the empty string in this case. The value assigned to both B and C consists of the two apostrophe characters side by side; this is treated as a quoted string of length zero, even though it is a sequence of two characters. In this last expansion, the first conditional, however, evaluates the form

```
NOT NUL ''
```

that is the special case of NUL applied to a length zero quoted string, but not a length zero sequence. Because of the special treatment of the length zero quoted string, this expression also produces a false result. The third conditional, however, must be considered carefully. The original expression in the macro definition takes the form

NOT NUL B&C

with B and C both associated with the sequence of length two given by two adjacent apostrophes. Thus, the macro assembler examines

```
NOT NUL ''&''
```

or, after concatenation,

```
NOT NUL ''''
```

where the four apostrophes are adjacent. Considering only the four apostrophes, the macro assembler considers this a quoted string that happens to contain a single apostrophe because double apostrophes are always reduced to a single apostrophe. As a result, the test produces a true value, and the conditional segment is assembled. Usually the NUL operator is used only to test for missing arguments, as shown in later examples. (See Listing 8-6.)

8.4 Nested Macro Definitions

The MAC assembler allows you to include nested macro definitions. These take the form

```
mac1    MACRO     mac1-list
        . . .
mac2    MACRO     mac2-list
        . . .
        ENDM
        . . .
        ENDM
```

where mac1 is the identifier corresponding to the outer macro, and mac2 is an identifier corresponding to an inner nested macro that is wholly contained within the outer macro. In this case, mac1-list and mac2-list correspond to the dummy parameter lists for mac1 and mac2, respectively. As before, labels are allowed on the ENDM statements.

The statements contained within a macro definition are prototype statements that are read and stored by the assembler but not evaluated as assembly language statements until the macro is expanded. Thus, in the preceding form, only the mac1 macro is available for expansion because the assembler has stored but not processed the body of mac1 that contains the definition of mac2. mac2 cannot be expanded until mac1 is first expanded, revealing the definition of mac2.

Properly balanced embedded macros of this form can be nested to any level, but they cannot be referenced until their encompassing macros have themselves been expanded.

Listing 8-4 gives a practical example of nested macro definition and expansion. This program writes characters either to the CP/M console device or to the currently assigned list device, according to the value of the LISTDEV flag set for the assembly. If the LISTDEV flag is true, then the assembly sends characters to the listing device. Otherwise, the console is used for output. In either case, the macro OUTPUT is produced; this sends a single character to the selected device.

The sample program in Listing 8-4 uses the macro SETIO to construct the OUTPUT macro. The OUTPUT macro is wholly contained within the SETIO macro and, as a result, remains undefined until SETIO is expanded. Upon encountering the invocation of SETIO, the macro assembler reads the prototype statements within SETIO and, in the process, constructs the definition of the OUTPUT macro. Because LISTDEV is true for this assembly, the OUTPUT macro is defined as

```
OUTPUT      MACRO       CHAR
            MVI         E,CHAR
            MVI         C,LISTOUT
            CALL        BDOS
            ENDM
```

Note that the SETIO macro itself uses this newly created OUTPUT macro in its last prototype statement to print a single + at the selected device.

Following the invocation of SETIO, the invocations of OUTPUT are recognized because its definition has been entered in the process of reading the prototype statements of SETIO. These invocations send the characters 1 and 2 to the list device.

```
0100                    ORG     100H    ;BASE OF THE TPA
0000 =          FALSE   EQU     0000H           ;VALUE OF FALSE
FFFF =          TRUE    EQU     NOT FALSE       ;VALUE OF TRUE
                ;       LISTDEV IS TRUE IF LIST DEVICE IS USED
                ;       FOR OUTPUT, AND FALSE IF CONSOLE IS USED
FFFF =          LISTDEV         EQU     TRUE
                ;
                ;
0005 =          BDOS    EQU     5       ;BDOS ENTRY POINT
0002 =          CONOUT  EQU     2       ;WRITE TO CONSOLE
0005 =          LISTOUT EQU     5       ;WRITE TO LIST DEVICE
                ;
                SETIO   MACRO   ;SETUP OUTPUT MACRO FOR LIST OR CONSOLE
                ;
                OUTPUT  MACRO   CHAR
                        MVI     E,CHAR  ;;READY THE CHARACTER FOR PRINTING
                        IF      LISTDEV
                        MVI     C,LISTOUT
                        ELSE
                        MVI     C,CONOUT
                        ENDIF
                        CALL    BDOS
                        ENDM
                        OUTPUT  '*'
                        ENDM
                ;
                        SETIO           ;SETUP THE IO SYSTEM
0100+1E2A               MVI     E,'*'
0102+0E05               MVI     C,LISTOUT
0104+CD0500             CALL    BDOS
                        OUTPUT  '1'
0107+1E31               MVI     E,'1'
0109+0E05               MVI     C,LISTOUT
010B+CD0500             CALL    BDOS
                        OUTPUT  '2'
010E+1E32               MVI     E,'2'
0110+0E05               MVI     C,LISTOUT
0112+CD0500             CALL    BDOS
0115 C9                 RET
0116                    END
```

Listing 8-4. Sample Program Showing a Nested Macro Definition

8.5 Redefinition of Macros

It is often useful to redefine the prototype statements of a macro after the initial prototype statements have been entered. Redefinition is a specific instance of the nesting described in the previous section, where the inner nested macro carries the same name as the encompassing macro definition. Macro redefinition is extremely useful if the macro contains a subroutine. In this case, the subroutine can be included on the first expansion and simply called in any remaining expansions. Thus, if the macro is never invoked, the subroutine is not included in the program.

Listing 8-5 shows an example of macro redefinition. This sample program defines the macro MOVE. MOVE is intended to move byte values from a starting source address to a target destination address for a particular number of bytes. The three dummy parameters denote these three values: SOURCE is the starting address; DEST is the destination address, and COUNT is the number of bytes to move (a constant in the range 0-65535). The actions of the MOVE macro, however, are complicated enough to be performed through a subroutine, rather than inline machine code each time MOVE is expanded.

Examining the structure of MOVE in Listing 8-5, note that it contains a properly nested redefinition of MOVE, taking the general form:

```
MOVE  MACRO    SOURCE,DEST,COUNT
      . . .
      @MOVE subroutine
MOVE  MACRO    ?S,?D,?C
      call to @MOVE
      ENDM
      invocation of MOVE
      ENDM
```

Upon encountering the first invocation of MOVE, the assembler begins reading the prototype statements. Note, however, that the first expansion of the MOVE includes the subroutine for the actual move operation, labeled by @MOVE so that there is no name conflict (with a branch around the subroutine). MOVE then redefines itself as a sequence of statements that simply call the out-of-line subroutine each time it expands. The last statement of the original MOVE macro is an invocation of the newly defined version. As indicated by this example, once a macro has started expansion, it continues to completion (or until EXITM is assembled), even if it redefines itself.

```
0100                    ORG     100H    ;BASE OF TPA
                MOVE    MACRO   SOURCE,DEST,COUNT
                ;;      MOVE DATA FROM ADDRESS GIVEN BY 'SOURCE'
                ;;      TO ADDRESS GIVEN BY 'DEST' FOR 'COUNT' BYTES
                        LOCAL   PASTSUB ;;LABEL AT END OF SUBROUTINE
                ;;
                        JMP     PASTSUB ;;JUMP AROUND INLINE SUBROUTINE
                @MOVE:  ;;INLINE SUBROUTINE TO PERFORM  MOVE OPERATION
                ;;      HL IS SOURCE, DE IS DEST, BC IS COUNT
                        MOV     A,C     ;;LOW ORDER COUNT
                        ORA     B       ;;ZERO COUNT?
                        RZ              ;;STOP MOVE IF ZERO REMAINDER
                        MOV     A,M     ;;GET NEXT SOURCE CHARACTER
                        STAX    D       ;;PUT NEXT DEST CHARACTER
                        INX     H       ;;ADDRESS FOLLOWING SOURCE
                        INX     D       ;;ADDRESS FOLLOWING DEST
                        DCX     B       ;;COUNT=COUNT-1
                        JMP     @MOVE   ;;FOR ANOTHER BYTE TO MOVE
                PASTSUB:
                ;;      ARRIVE HERE ON FIRST INVOCATION - REDEFINE MOVE
                MOVE    MACRO   ?S,?D,?C        ;;CHANGE PARM NAMES
                        LXI     H,?S    ;;ADDRESS THE SOURCE STRING
                        LXI     D,?D    ;;ADDRESS THE DEST STRING
                        LXI     B,?C    ;;PREPARE THE COUNT
                        CALL    @MOVE   ;;MOVE THE STRING
                        ENDM
                ;;      CONTINUE HERE ON THE FIRST INVOCATION TO USE
                ;;      THE REDEFINED MACRO TO PERFORM THE FIRST MOVE
                        MOVE    SOURCE,DEST,COUNT
                        ENDM
                ;
```

Listing 8-5. Sample Program Showing Macro Redefinition

```
                        MOVE    X1,X2,5 ;MOVE 5 CHARS FROM X1 TO X2
0100+C30E01             JMP     ??0001
0103+79                 MOV     A,C
0104+B0                 ORA     B
0105+C8                 RZ
0106+7E                 MOV     A,M
0107+12                 STAX    D
0108+23                 INX     H
0109+13                 INX     D
010A+0B                 DCX     B
010B+C30301             JMP     @MOVE
010E+212701             LXI     H,X1
0111+114001             LXI     D,X2
0114+010500             LXI     B,5
0117+CD0301             CALL    @MOVE
                        MOVE    3000H,1000H,1500H       ;BIG MOVER
011A+210030             LXI     H,3000H
011D+110010             LXI     D,1000H
0120+010015             LXI     B,1500H
0123+CD0301             CALL    @MOVE
0126 C9                 RET             ;RETURN TO THE CCP
0127 6865726520X1:      DB      'here is some data to move'
0140 7878787878X2:      DB      'xxxxxwe are!'
```

Listing 8-5. (continued)

It is important to note the use of ?S, ?D, and ?C in the previous example. The innermost MOVE macro uses the same sequence of three parameters for the source, destination, and count. The dummy parameter names must differ, however, because they would be substituted by their actual values if they were the same. This is because the inner MOVE macro is wholly contained within the outer macro, so parameter substitution takes place regardless of the context.

Macro storage is not reclaimed upon definition, however, because the macro assembler performs two passes through the source program and saves any preceding definitions for the second pass scan.

8.6 Recursive Macro Invocation

The prototype statements of a recursive macro x contain invocations of macros that, in turn, invoke macros that eventually lead back to an invocation of x. A direct recursion occurs when x invokes itself, as shown in the form below:

```
macname     MACRO      d-1,. . .,d-n
            . . .
            macname    a-1,. . .,a-n
            . . .
            ENDM
```

Although this form is similar to the embedded macro definition discussed in the previous section, macname is expanded within its own definition, rather than being redefined. Recursion is only useful, however, in the presence of conditional assembly where various tests are made that prevent infinite recursion. In fact, recursion is allowed only to sixteen levels before returning to complete the expansion of an earlier level.

Listing 8-6 shows a situation in which indirect recursive macro invocation is useful. The macro WCHAR writes a character to the console device using the general purpose operating system macro CBDOS (call BDOS). CBDOS acts as an interface between the program and the CP/M system by performing the system function given by FUNC, with optional information address INFO. CBDOS loads the specified function to register C, then tests to see whether the INFO argument has been supplied, using the NUL operator. If supplied, INFO is loaded to the DE register pair. After register setup, the BDOS is called, and the macro has completed its expansion.

Assume, however, that CBDOS has the additional task of inserting a carriage return line-feed before writing messages where operating system Function 9 (write buffer until $) has been specified. In this case, CBDOS uses the WCHAR macro to send the carriage return line-feed. The WCHAR macro, in turn, uses CBDOS to send the character, resulting in two activations of CBDOS at the same time. The assembler holds the initial invocation of CBDOS until the WCHAR macro has completed, then returns to complete the initial CBDOS expansion.

In recursion the values of the dummy parameters are saved at each successive level of recursion and restored when that level of recursion is reinstated. Reentry into a macro expansion through recursion does not destroy the values of dummy arguments held by previous entry levels.

```
0100                   ORG     100H    ;BASE OF TRANSIENT AREA
              ;        SAMPLE PROGRAM SHOWING RECURSIVE MACROS
0005 =        BDOS     EQU     0005H   ;ENTRY TO BDOS
0002 =        CONOUT   EQU     2       ;CONSOLE CHARACTER OUT
0009 =        MSGOUT   EQU     9       ;PRINT MESSAGE 'TIL $
000D =        CR       EQU     0DH     ;CARRIAGE RETURN
000A =        LF       EQU     0AH     ;LINE FEED
              ;
              WCHAR    MACRO   CHR
              ;;       WRITE THE CHARACTER CHR TO CONSOLE
                       CBDOS   CONOUT,CHR       ;;CALL BDOS
                       ENDM
              ;
              CBDOS    MACRO   FUNC,INFO
              ;;       GENERAL PURPOSE BDOS CALL MACRO
              ;;       FUNC IS THE FUNCTION NUMBER,
              ;;       INFO IS THE INFORMATION ADDRESS OR NUL
              ;;       CHECK FOR FUNCTION 9, SEND CRLF FIRST IF SO
                       IF      FUNC=MSGOUT
              ;;       PRINT CRLF FIRST
                       WCHAR   CR
                       WCHAR   LF
                       ENDIF
              ;;       NOW PERFORM THE FUNCTION
                       MVI     C,FUNC
              ;;       INCLUDE LXI TO DE IF INFO NOT EMPTY
                       IF      NOT NUL INFO
                       LXI     D,INFO
                       ENDIF
                       CALL    BDOS
                       ENDM
              ;
                       WCHAR   'h'       ;SEND ; "H" TO CONSOLE
0100+0E02              MVI     C,CONOUT
0102+116800            LXI     D,'h'
0105+CD0500            CALL    BDOS
```

Listing 8-6. Sample Program Showing a Recursive Macro

```
                       WCHAR   'i'        ;SEND 'I' TO CONSOLE
0108+0E02              MVI     C,CONOUT
010A+116900            LXI     D,'i'
010D+CD0500            CALL    BDOS
                       CBDOS   MSGOUT,MSGADDR  ;SEND MESSAGE
0110+0E02              MVI     C,CONOUT
0112+110D00            LXI     D,CR
0115+CD0500            CALL    BDOS
0118+0E02              MVI     C,CONOUT
011A+110A00            LXI     D,LF
011D+CD0500            CALL    BDOS
0120+0E09              MVI     C,MSGOUT
0122+112901            LXI     D,MSGADDR
0125+CD0500            CALL    BDOS
0128 C9                RET              ;TERMINATE PROGRAM
                 ;
                 MSGADDR:
 0129 616E64206C       DB      'and lois$'
 0132                  END
```

Listing 8-6. (continued)

8.7 Parameter Evaluation Conventions

You can exercise a number of options in the construction of actual parameters, and in the specification of character lists for the IRP group. Although an actual parameter is simply a sequence of characters placed between parameter delimiters, these options allow overrides where delimiter characters themselves become a part of the text. A parameter x occurs in the context:

label: macname <. . ., x ,. . .>

where macname is the name of a previously defined macro, and the preceding label is optional. The ellipses . . . represent optional surrounding actual parameters in the invocation of macname. In the case of an IRP group, the occurrence of a character list x is

label: IRP id,. . ., x ,. . .

where the label is again optional, and the ellipses represent optional surrounding character lists for substitution within the IRP group where the controlling identifier id is found. In either case, the statements can be contained within the scope of a surrounding macro expansion. Hence, dummy parameter substitution can take place for the encompassing macro while the actual parameter is being scanned.

The macro assembler follows the steps shown below in forming an actual parameter or character list:

1. Leading blanks and tabs (control-I) are removed if they occur in front of x.

2. The leading character of x is examined to determine the type of scan operation to take place.

3. If the leading character is a string quote (apostrophe), then x becomes the text up to and including the balancing string quote, using the normal string scanning rules: double apostrophes within the string are reduced to a single apostrophe, and upper-case dummy parameters adjacent to the ampersand symbol are substituted by the actual parameter values. Note that the string quotes on either end of the string are included in the actual parameter text.

4. If the first character is the left angle bracket (<), then the bracket is removed, and the value of x becomes the sequence of characters up to, but not including, the balancing right angle bracket (>). The right angle bracket does not become a part of x. In this case, left and right angle brackets can be nested to any level within x, and only the outer brackets are removed in the evaluation. Quoted strings within the brackets are allowed, and substitution within these strings follows the rules stated in 3 above. Left and right brackets within quoted strings become a part of the string; these are not counted in the bracket nesting within x. Further, the delimiter characters comma, blank, semicolon, tab, and exclamation point become a part of x when they occur within the bracket nesting.

5. If the leading character is a percent (%) character, then the sequence of characters that follows is taken as an expression that is evaluated immediately as a 16-bit value. The resulting value is converted to a decimal number and treated as an ASCII sequence of digits, with left zero suppression (0-65535).

6. If the leading character is not a quote, a left bracket, or a percent, the possibly empty sequence of characters that follows, up to the next comma, blank, tab, semicolon, or exclamation point, becomes the value of x.

There is one important exception to the preceding rules: the single-character escape, denoted by an up arrow, causes the macro assembler to read the special (nonalphabetic) character immediately following as a part of x without treating the character as significant. The character following the up arrow, however, must be a blank, tab, or visible ASCII character. The up arrow itself can be represented by two up arrows in succession. If the up arrow directly precedes a dummy parameter, then the up arrow is removed, and the dummy parameter is not replaced by its actual parameter value. Thus, the up arrow can be used to prevent evaluation of dummy parameters within the macro body. Note that the up arrow has no special significance within string quotes and is simply included as a part of the string.

Evaluation of dummy parameters in macro expansions has been presented throughout the previous sections. The macro assembler evaluates dummy parameters as follows:

- If a dummy parameter is either preceded or followed by the concatenation operator &, then the preceding or following & operator is removed, the actual parameter is substituted for the dummy parameter, and the implied delimiter is removed at the position where the ampersand occurs.
- Dummy parameters are replaced only once at each occurrence as the encompassing macro expands. This prevents the infinite substitution that occurs if a dummy parameter evaluates to itself.

In summary, parameter evaluation follows these rules:

- Leading and trailing tabs and blanks are removed.
- Quoted strings are passed with their string quotes intact.
- Nested brackets enclose arbitrary characters with delimiters.
- A leading percent symbol causes immediate numeric evaluation.
- An up arrow passes a special character as a literal value.
- An up arrow prevents evaluation of a dummy parameter.
- The & operator is removed next to a dummy parameter.
- Dummy parameters are replaced only once at each occurrence.

Listings 8-7, 8-8, and 8-9 show examples of macro definitions and invocations illustrating these points. In Listing 8-7, for example, two macros are defined, called MAC1 and MAC2. Each has several dummy parameters. In this case, the macro definitions are headed by DB statements to reveal the actual values passed in each case. There is a single mainline invocation of MAC2 with the actual parameters

```
I ,,  X+1, % X + 1, 'Kwote'
```

that associates I with E, the null sequence with F, the sequence X+1 with G, the value 16 with H, and the literal string 'kwote' with S. MAC2 expands, filling the DB and MVI instructions with the substituted values. Before leaving MAC2, MAC1 is invoked with the value of E (the sequence I), the concatenation of the dummy argument F with the sequence M (producing M since F's value is null), along with the literal value A, followed by the value of H (which is 16), and terminated by the value of S (yielding the string 'kwote'). These values are associated with MAC1's dummy parameters.

```
;       MACRO PARAMETER EVALUATION
;
MAC1    MACRO   A,B,C,D,S
;
;       ENTERING MACRO 1:
        DB      '&A &B &C &D'
        DB      S
A:      NOP
        MVI     B,1
C&1:    NOP
L&A&D:  NOP
;       LEAVING MACRO 1
;
        ENDM
;
MAC2    MACRO   E,F,G,H,S
;
;       ENTERING MACRO 2:
        DB      '&E &F &G &H'
        DB      S
        MVI     M,H
        MAC1    E,F&M,A,H,S
;       LEAVING MACRO 2
;
        ENDM
```

Listing 8-7. Macro Parameter Evaluation Example

```
                ;
000F =          X       EQU     15
                        MAC2    I  ,,  X+1,  % X + 1, 'Kwote'
    +           ;
    +           ;       ENTERING MACRO 2:
0000+4920205828         DB      'I  X+1 16'
0009+6B776F7465         DB      'Kwote'
000E+3610               MVI     M,16
    +                   MAC1    I,M,I,16,'Kwote'
    +           ;
    +           ;       ENTERING MACRO 1:
0010+49204D2049         DB      'I M I 16'
0018+6B776F7465         DB      'Kwote'
001D+00         I:      NOP
001E+3601               MVI     M,1
0020+00         I1:     NOP
0021+00         LI16:   NOP
    +           ;       LEAVING MACRO 1
    +           ;
    +                   ENDM
    +           ;       LEAVING MACRO 2
    +           ;
    +                   ENDM
0022                    END
```

Listing 8-7. (continued)

Upon expanding MAC1, the DB statements are filled out, followed by the substitution of A as a label (producing A's value I). The MVI instruction references memory because B's value is M. Note that the concatenation of C with 1 reduces to a concatenation of A with 1 because C's value is A. The replacement of C by A constitutes a substitution of a single occurrence of a dummy parameter. Thus the A that is produced is not itself replaced at this point. Finally, the literal value L is concatenated to the value of A and D to produce the label LI16.

Listing 8-8 illustrates the use of bracketed notation, using IRPs (indefinite repeats) within three macros, called IRPM1, IRPM2, and IRPM3. Note that one bracket level is removed in the first invocation of IRPM1, leaving the IRP list with one bracket level (required in the IRP heading). Similarly, the IRPM2 invocation also eliminates the outer bracket level, but these brackets are replaced at the IRP heading within IRPM2. IRPM3 has three distinct dummy parameters that are reconstructed as a single list at the IRP heading it contains. IRPM4 shows the effect of passing parameters through two macro invocation levels by accepting a single parameter X, which is immediately passed along to the IRPM1 macro. Note that the invocation requires three bracket levels: the first is removed at the nested invocation of IRPM1 inside IRPM4, and the innermost level is required at the IRP heading within IRPM1.

Listing 8-9 presents various combinations of bracketed actual parameters, quoted strings, and escape sequences. The MAC1 macro has two parts: the first portion includes a DB statement showing the value of the first parameter X, if it is not empty, and the second part produces the value of Y, if not empty. Note that the first invocation includes a properly nested bracketed sequence for X and an empty parameter for Y. The second invocation sends a properly nested bracketed expression for X that produces an empty value because no characters remain after the brackets are removed. The second parameter includes a quoted string ('string of pearls') and a hexidecimal value that becomes a part of the DB in MAC1.

The third invocation of MAC1 passes a bracketed expression, including a quoted string (the pair of adjacent apostrophes), followed immediately by a sequence of ASCII characters. Note that the pair of apostrophes are passed intact because they appear as an empty quoted string. In this case, the value of Y is empty. The remaining examples show various cases of strings and escape sequences. Take care in passing quoted strings that contain apostrophes because a pair of apostrophes is considered a single apostrophe at each evaluation level in the sequence of macro invocations. Pay particular attention to the use of the escape character to pass an unevaluated dummy parameter from MAC2 to the MAC1 invocation.

```
                IRPM1   MACRO   X
                ;;      INDEFINITE REPEAT MACRO
                        IRP     Y,X
                Y:      NOP
                        ENDM
                        ENDM
                ;
                        IRPM1   <<ONE,TWO,THREE>>
0000+00         ONE:    NOP
0001+00         TWO:    NOP
0002+00         THREE:  NOP
                ;
                IRPM2   MACRO   X
                        IRP     Y,<X>
                Y:      NOP
                        ENDM
                        ENDM
                ;
```

Listing 8-8. Parameter Evaluation Using Bracketed Notation

```
                        IRPM2   <FOUR,FIVE,SIX>
0003+00         FOUR:   NOP
0004+00         FIVE:   NOP
0005+00         SIX:    NOP
                ;
                IRPM3   MACRO   X1,X2,X3
                        IRP     Y,<X1,X2,X3>
                Y:      NOP
                        ENDM
                        ENDM
                ;
                        IRPM3   SEVEN,EIGHT,NINE
0006+00         SEVEN:  NOP
0007+00         EIGHT:  NOP
0008+00         NINE:   NOP
                ;
                IRPM4   MACRO   X
                        IRPM1   X
                        ENDM
                ;
                        IRPM4   <<<TEN,ELEVEN,TWELVE>>>
0009+00         TEN:    NOP
000A+00         ELEVEN: NOP
000B+00         TWELVE: NOP
000C                    END
```

Listing 8-8. (continued)

```
               ;      SAMPLE BRACKETED PARAMETERS, WITH ESCAPE CHARACTER
               ;
               MAC1   MACRO   X,Y
                      DB      '&X'      ;(ONE)
                      IF      NUL Y
                      EXITM
                      ENDIF
                      DB      Y       ;(TWO)
                      ENDM
               ;
                      MAC1    <<LEFT SIDE> MIDDLE <RIGHT SIDE>>
0000+3C4C454654       DB      '<LEFT SIDE> MIDDLE <RIGHT SIDE>'     ;(ONE)
               ;
                      MAC1    <>,<'string of pearls',34H>
001F+737472696E       DB      'string of pearls',34H    ;(TWO)
               ;
                      MAC1    <A QUOTE IS A '', RIGHT?>
0030+412051554F       DB      'A QUOTE IS A '', RIGHT?'   ;(ONE)
               ;
                      MAC1    <>,<'right, but also '''''>
0046+7269676874       DB      'right, but also '''        ;(TWO)
               ;
                      MAC1    ,<'is this ',''''''confusing'''''',63>
0057+6973207468       DB      'is this ',''''confusing'''',63   ;(TWO)
               ;
                      MAC1    <HERE IS A ^> AND A ^ ^ >
006B+4845524520       DB      'HERE IS A > AND A ^'    ;(ONE)
               ;
               MAC2   MACRO   APAR,BPAR
                      LOCAL   X
               X      EQU     10
                      DB      APAR
                      MAC1    ^APAR,BPAR
                      ENDM
               ;
                      MAC2    (X+5)*4,'what''''''''is going on?'
000A+=         ??0001 EQU     10
007E+3C               DB      (??0001+5)*4
007F+41504152         DB      'APAR'    ;(ONE)
0083+7768617427       DB      'what''s going on?' ;(TWO)
```

Listing 8-9. Examples of Macro Parameter Evaluation

Examine the various parameters and their evaluations in Listing 8-9 to ensure that the rules for evaluation given in this section are consistent.

8.8 The MACLIB Statement

The macro assembler allows you to create and reference macro library files that are external to the mainline program. The form of the macro library reference is

```
    MACLIB      libname
```

where libname is an identifier referencing file libname.LIB assumed to exist on the disk. Macro libraries are in source program form, so you can easily create and modify them using an editor program.

In order to speed up the assembly process, macro libraries are read only on the first assembly pass. This places some restrictions on the use of the MACLIB statement, as listed below:

- The statements included in the macro library cannot generate machine code. For example, comments, EQUs, SETs, and MACRO definitions are allowed; DB statements outside macro definitions are not allowed.
- Macro libraries are not listed with the source program, although an overriding parameter can be supplied. (See Section 10.)
- All MACLIB statements must appear before the mainline program macro definitions. The MACLIB statements are placed at the beginning of the program, followed by the mainline declarations and machine code.

The principal advantage of the MACLIB feature is that you can predefine macros that enhance the facilities of the assembly language itself. For example, the additional operations codes of the Zilog Z80 microprocessor can be defined in a macro library that is referenced in a single statement

```
MACLIB    Z80
```

causing the assembler to read the file Z80.LIB from the disk that contains the necessary macros for Z80 code generation. These macros can then be referenced within the program, intermixed with the usual 8080 mnemonics.

The libname.LIB file is assumed to exist on the currently logged disk drive. You can override this default condition using a special parameter (L) when the macro assembler is started that redirects the .LIB references to a different disk. (See Section 10.)

Listings 6-1 and 6-2 show the use of the macro library facility, as introduced in the initial macro discussion. The following sections contain additional examples of the use of MACLIB in practical applications.

End of Section 8

Section 9
Macro Applications

The MAC assembler provides a powerful tool for microcomputer systems development through its macro facilities. To demonstrate this, the following sections describe a number of macro applications that solve practical problems in four applications areas:

- implementation of special purpose languages
- emulation of nonstandard machine architectures
- implementation of additional control structures
- operating systems interface macros

9.1 Special Purpose Languages

A wide variety of microcomputer designs can be broadly classed as controller applications. Specifically, the microcomputer is used as the controlling element in sequencing and decision making as real-time events are sampled and directed.

Typical applications of this sort include assembly line sensing and control, metal machine control, data communications and terminal control functions, production instrumentation and testing, and traffic control systems.

In many cases, application programmers set up the sequence of operations that the microprocessor carries out in performing its task. To avoid unnecessary details, the application programmer is not expected to know how to program and debug microcomputer assembly language programs.

In this situation, it is useful to define a language through macros that suit the application. The application programmer uses these predefined macros as the primitive language elements. If properly defined, the application language is easily programmed, allowing considerable machine independence. That is, an application program written for a particular microprocessor can be used with another processor by changing the definitions of the individual macros that implement the primitive operations. Further, the macro bodies can incorporate debugging facilities for application development.

To illustrate language definition, consider the following situation. Hornblower Highway Systems, Inc. produces turnkey traffic control systems for cities throughout the country. Their hardware subsystems consist of various traffic lights and sensors customized for the traffic layout in a particular city. When Hornblower negotiates a contract, their engineers survey the intersections of the city and produce plans showing a configuration of their standard hardware for each intersection, along with the algorithms required for traffic flow at that point.

The standard hardware items Hornblower manufactures consist of central and corner traffic lights that display green, yellow, and red (or off completely); pushbutton switches for pedestrian cross requests; road treadles for sensing the presence of an automobile at an intersection; and a central controller box.

The central controller box contains an 8080 microcomputer connected through external logic to relays that control the lights and latches that hold the sensor input information. The controller box also contains a time of day clock that changes on an hourly basis from 0 through 23. The 8080 processor in the controller box can be configured for any particular intersection with up to 1024 bytes of programmable Read-Only Memory (PROM) in 256-byte increments. Although Random Access Memory can be included in the controller box, Hornblower uses only ROM when possible.

Thus, the Hornblower engineers examine the hardware requirements for each intersection in the city and produce hardware configuration plans that intermix the various standard components. Programs are then written and debugged that control each intersection, based on predicted traffic patterns.

The intersection of Easy Street and Maria Avenue, for example, controls minimal traffic and thus consists of a controller box with a single central light. The algorithm for this intersection simply alternates red and green lights between Easy and Maria, with a bias toward Easy Street because traffic along Easy has measured higher in the past surveys. Thus the green light along Easy lasts for 20 seconds, while the green along Maria lasts for only 15 seconds. Given this situation, the application programmer writes the following program:

```
;       HORNBLOWER HIGHWAYS SYSTEMS, INC.
;       INTERSECTION:
;           EASY STREET (N-S) / MARIA AVENUE(E-W)
;
        MACLIB     INTERSECT     ;LOAD MACROS
;
CYCLE:  SETLITE    NS,GREEN
        SETLITE    EW,RED
        TIMER      20            ;WAIT 20 SECS
;
;       CHANGE LIGHTS
        SETLITE    NS,YELLOW
        TIMER      3             ;WAIT 3 SECS
        SETLITE    NS,RED
        SETLITE    EW,GREEN
        TIMER      15            ;WAIT 15 SECS
;
;       CHANGE BACK
        SETLITE    EW,YELLOW
        TIMER      3             ;WAIT 3 SECS
        RETRY      CYCLE
```

The macro library INTERSECT.LIB contains the macro definitions that implement the primitive operations SETLITE and TIMER, setting the central traffic light and time out for the specified interval, respectively. Further, the RETRY macro causes the traffic light to recycle on each light change. The sequence of operations is easy to write and is completely machine independent.

Listing 9-1 gives an example of a macro library for intersect that assumes the following hardware with an 8080 processor: the central traffic light is controlled by the 8080 output port 0 (given by light); the time of day clock is read from port 3 (clock). Further, the north-south (nsbits) of the central light are given by the high-order 4 bits of output port 0; the east-west direction (ewbits) is specified in the low-order 4 bits of output port 0. When either of these fields is set to 0, 1, 2, or 3, the light in that direction is turned off, or set to red, yellow, or green, respectively. Thus, the SETLITE macro in Listing 9-1 accepts a direction (NS or EW) along with a color (OFF, RED, YELLOW, or GREEN) and sets the specified direction to the appropriate color.

```
;       macro library for basic intersection
;
;       input/output ports for light and clock
light   equ     00h     ;traffic light control
clock   equ     03h     ;24 hour clock (0,1,...,23)
;
;       constants for traffic light control
nsbits  equ     4       ;north south bits
ewbits  equ     0       ;east west bits
;
off     equ     0       ;turn light off
red     equ     1       ;value for red light
yellow  equ     2       ;value for yellow light
green   equ     3       ;green light
;
setlite macro   dir,color
;;      set light ;"dir" (ns,ew) to ;"color" (off,red,yellow,green)
        mvi     a,color shl dir&bits    ;;color readied
        out     light   ;;sent in proper bit position
        endm
;
timer   macro   seconds
;;      construct inline time-out loop
        local   t1,t2,t3        ;;loop entries
        mvi     d,4*seconds     ;;basic loop control
t1:     mvi     b,250   ;;250msec *4 = 1 sec
t2:     mvi     c,182   ;;182*5.5usec = 1msec
t3:     dcr     c       ;;1 cy = .5 usec
        jnz     t3      ;;+10 cy = 5.5 usec
        dcr     b       ;;count 250,249...
        jnz     t2      ;;loop on b register
        dcr     d       ;;basic loop control
        jnz     t1      ;;loop on d register
```

Listing 9-1. Macro Library for Basic Intersection

```
;;      arrive here with approximately ;"seconds" secs timeout
        endm
;
clock?  macro   low,high,iftrue
;;      jump to ;"iftrue" if clock is between low and high
        local   iffalse ;;alternate to true case
        in      clock   ;;read real-time clock
        if      not nul high    ;;check high clock
        cpi     high    ;;equal or greater?
        jnc     iffalse ;;skip to end if so
        endif
        cpi     low     ;;less than low value?
        jnc     iftrue  ;;skip to label if not
iffalse:
        endm
;
retry   macro   golabel
;;      continue execution at ;"golabel"
        jmp     golabel
        endm
```

Listing 9-1. (continued)

The TIMER macro in Listing 9-1 uses the internal cycle time of the 8080 processor to construct an inline timing loop, based on the value of SECONDS. This loop is not generated as a subroutine because Hornblower prefers not to include RAM in the controller box. (Subroutines require return addresses in RAM.)

In addition to the basic intersection macro library, Hornblower has also defined macro libraries for all of the optional hardware components. Listing 9-2a, for example, is included when the intersection contains treadles in the street to detect automobiles; Listing 9-2b shows the macro library for pedestrian pushbuttons. In the case of automotive treadles, the sensors are attached to input port 1 (trinp) of the processor. The treadles, however, require a reset operation that clears the latched value through output port 1 (trout) of the controlling 8080 processor. In any particular intersection, the treadles are numbered clockwise from true north, labeled 0, 1, through a maximum of 7 treadles. Each sensor and reset position of the treadle ports corresponds to one bit position, numbered from the least to most significant bit. Thus the treadle #0 sensor is read from bit 0 of port 1 and reset by setting bit 0 of output port 1. Similarly, treadle #1 uses bit position 1 of input and output port 1. The TREAD? macro is invoked to sense the presence of a latched value for treadle tr and, if on, the sensor is reset, with control transferring to the label given by iftrue.

Listing 9-2b shows the macro library that processes pedestrian pushbuttons. Hornblower's hardware senses the latched pedestrian switches on input port 0 (cwinp) as a sequence of 1s and 0s in the least significant positions, corresponding to the switches at the intersection. Thus, if there are four pedestrian switches, bit positions 0, 1, 2, and 3 correspond to these switches. A 1 bit in any of these positions indicates that the pushbutton has been depressed. Unlike the automotive treadles, the crosswalk switch latches are all cleared whenever input port 0 is read. Hornblower has defined several other libraries that support optional hardware manufactured by their company.

```
;       macro library for street treadles
;
trinp   equ     01h     ;treadle input port
trout   equ     01h     ;treadle output port
;
tread?  macro   tr,iftrue
;;      ;"tread?" is invoked to check if
;;      treadle given by tr has been sensed.
;;      if so, the latch is cleared and control
;;      transfers to the label ;"iftrue"
        local   iffalse         ;;in case not set
;;
        in      trinp   ;;read treadle switches
        ani     1 shl tr        ;;mask proper bit
        jz      iffalse         ;;skip reset if 0
        mvi     a,1 shl tr      ;;to reset the bit
        out     trut            ;;clear it
        jmp     iftrue          ;;go to true label
iffalse:
        endm
```

Listing 9-2a. Macro Library for Treadle Control

```
;       macro library for pedestrian pushbuttons
;
cwinp   equ     00h     ;input port for crosswalk
;
push?   macro   iftrue
;;      ;"push?" jumps to label ;"iftrue" when any one
;;      of the crosswalk switches is depressed.  The
;;      value has been latched, and reading the port
;;      clears the latched values
        in      cwinp   ;;read the crosswalk switches
        ani     (1 shl cwcnt) - 1       ;;build mask
        jnz     iftrue  ;;any switches set?
;;      continue on false condition
        endm
```

Listing 9-2b. Macro Library for Corner Pushbuttons

The intersection of Bumpenram Boulevard and Lullabye Lane presents a more complicated situation. Bumpenram carries heavy traffic in an E-W direction to and from the center of town. Lullabye, however, feeds a residential portion of the city, running perpendicular to Bumpenram in a N-S direction. The contracting city wants the traffic control biased toward Bumpenram as follows: the traffic light must remain green along Bumpenram until the treadles along Lullabye detect the presence of automobiles or until the pedestrian switches are pushed. At that time, the light must change to allow the traffic to move N-S through Lullabye, allowing all traffic to clear before returning to the major E-W flow along Bumpenram. Late night traffic along Bumpenram is not very heavy, so the city also wants the E-W light to flash yellow and the N-S direction to flash red between the hours of 2 and 5 a.m.

The application program created by Hornblower for the Bumpenram and Lullabye intersection is shown in Listings 9-3a, 9-3b, and 9-3c. Each major cycle of the traffic light enters at CYCLE where the time of day is tested. Between 2 and 5 a.m., control transfers to NIGHT where the yellow and red lights are flashed in the appropriate directions. During other hours, the switches and treadles are sampled until N-S traffic along Lullabye is sensed. If cross traffic is detected, the lights switch until all the traffic is through. Sampling also stops when the time of day reaches 2 a.m.

Listing 9-3a shows the assembly with no macro generated lines, controlled by the -M parameter. (See Section 10.) Although the machine code locations are shown to the left, no 8080 machine code is listed. Listing 9-3b shows a segment of this same program with machine code generation, but no 8080 mnemonics, controlled by *M. Listing 9-3a is the most readable to the application programmer. Listings 9-3b and 9-3c are useful for macro debugging.

Note that the resulting program requires no RAM for execution because all temporary values are maintained in the 8080 registers. Further, the program is less than 256 bytes, so it can be placed in a single programmable Read-Only memory chip for a minimum memory/processor configuration.

```
                ;       INTERSECTION: BUMPENRAM BLVD / LULLABYE LN.

0004 =          CWCNT   EQU     4       ;SET TO 4 CROSSWALK SWITCHES
0000 =          LULL0   EQU     0       ;NAME FOR TREADLE ZERO
0001 =          LULL1   EQU     1       ;NAME FOR TREADLE ONE

                        MACLIB  INTER           ;BASIC INTERSECTION
                        MACLIB  TREADLES        ;INCLUDE TREADLES
                        MACLIB  BUTTONS         ;INCLUDE PUSHBUTTONS

                CYCLE:  ;ENTER HERE ON EACH MAJOR CYCLE OF THE LIGHT
0000                    CLOCK?  2,5,NIGHT       ;SPECIAL FLASHING?
                        ;NOT BETWEEN 2 AND 5 AM
000C                    SETLITE NS,RED          ;RED LIGHT ON LULLABYE
0010                    SETLITE EW,GREEN        ;GREEN ON BUMPENRAM

                SAMPLE: ;SAMPLE THE BUTTONS AND TREADLES
0014                    PUSH?   SWITCH  ;ANYONE THERE?
001B                    TREAD?  LULL0,SWITCH    ;TREADLE 0?
0029                    TREAD?  LULL1,SWITCH    ;TREADLE 1?
0037                    CLOCK?  2,,NIGHT        ;PAST 2AM?
003E                    RETRY   SAMPLE          ;TRY AGAIN IF NOT

                SWITCH:
                        ;SOMEONE IS WAITING, CHANGE LIGHTS
0041                    SETLITE EW,YELLOW       ;SLOW 'EM DOWN
0045                    TIMER   3               ;WAIT 3 SECONDS
0057                    SETLITE EW,RED          ;STOP 'EM
005B                    SETLITE NS,GREEN        ;LET 'EM GO
005F                    TIMER   23              ;FOR AWHILE

                DONE?:  ;IS ALL THE TRAFFIC THROUGH ON LULLABYE?
0071                    TREAD?  LULL0,NOTDONE   ;TREADLE 0?
007F                    TREAD?  LULL1,NOTDONE   ;TREADLE 1?
                        ;NEITHER TREADLE IS SET, CYCLE
008D                    RETRY   CYCLE           ;FOR ANOTHER LOOP


                NOTDONE:
0090                    TIMER   5               ;WAIT 5 SECONDS
00A2                    RETRY   DONE?           ;TRY AGAIN

                NIGHT:  ;THIS IS NIGHTTIME, FLASH LIGHTS
00A5                    SETLITE EW,OFF          ;TURN OFF
00A9                    SETLITE NS,OFF          ;TURN OFF
00AD                    TIMER   1               ;WAIT WITH OFF
00BF                    SETLITE EW,YELLOW       ;TURN TO YELLOW
00C3                    SETLITE NS,RED          ;TURN TO RED
00C7                    TIMER   1               ;LEAVE ON FOR 1 SEC
00D9                    RETRY   CYCLE           ;GO AROUND AGAIN
```

Listing 9-3a. Traffic Control Algorithm using -M Option

```
                ;       INTERSECTION: BUMPENRAM BLVD / LULLABYE LN.

0004 =          CWCNT   EQU     4       ;SET TO 4 CROSSWALK SWITCHES
0000 =          LULL0   EQU     0       ;NAME FOR TREADLE ZERO
0001 =          LULL1   EQU     1       ;NAME FOR TREADLE ONE

                MACLIB  INTER           ;BASIC INTERSECTION
                MACLIB  TREADLES        ;INCLUDE TREADLES
                MACLIB  BUTTONS         ;INCLUDE PUSHBUTTONS

                CYCLE:  ;ENTER HERE ON EACH MAJOR CYCLE OF THE LIGHT
                        CLOCK?  2,5,NIGHT       ;SPECIAL FLASHING?
0000+DB03
0002+FE05
0004+D20C00
0007+FE02
0009+D2A500
                        ;NOT BETWEEN 2 AND 5 AM
                        SETLITE NS,RED          ;RED LIGHT ON LULLABYE
000C+3E10
000E+D300
                        SETLITE EW,GREEN        ;GREEN ON BUMPENRAM
0010+3E03
0012+D300

                SAMPLE: ;SAMPLE THE BUTTONS AND TREADLES
                        PUSH?   SWITCH  ;ANYONE THERE?
0014+DB00
0016+E60F
0018+C24100
                        TREAD?  LULL0,SWITCH    ;TREADLE 0?
001B+DB01
001D+E601
001F+CA2900
0022+3E01
0024+D301
0026+C34100
                        TREAD?  LULL1,SWITCH    ;TREADLE 1?
0029+DB01
002B+E602
002D+CA3700
0030+3E02
0032+D301
0034+C34100
                        CLOCK?  2,,NIGHT        ;PAST 2 AM?
0037+DB03
0039+FE02
003B+D2A500
                        RETRY   SAMPLE          ;TRY AGAIN IF NOT
003E+C31400
```

Listing 9-3b. Intersection Algorithm with *M in Effect

```
                SWITCH:
                        ;SOMEONE IS WAITING, CHANGE LIGHTS
                        SETLITE EW,YELLOW       ;SLOW 'EM DOWN
0041+3E02               MVI     A,YELLOW SHL EWBITS
0043+D300               OUT     LIGHT
                        TIMER   3               ;WAIT 3 SECONDS
0045+160C               MVI     D,4*3
0047+06FA       ??0005: MVI     B,250
0049+0EB6       ??0006: MVI     C,182
004B+0D         ??0007: DCR     C
004C+C24B00             JNZ     ??0007
004F+05                 DCR     B
0050+C24900             JNZ     ??0006
0053+15                 DCR     D
0054+C24700             JNZ     ??0005
                        SETLITE EW,RED          ;STOP 'EM
0057+3E01               MVI     A,RED SHL EWBITS
0059+D300               OUT     LIGHT
                        SETLITE NS,GREEN        ;LET 'EM GO
005B+3E30               MVI     A,GREEN SHL NSBITS
005D+D300               OUT     LIGHT
                        TIMER   23              ;FOR AWHILE
005F+165C               MVI     D,4*23
0061+06FA       ??0008: MVI     B,250
0063+03B6       ??0009: MVI     C,182
0065+0D         ??0010: DCR     C
0066+C26500             JNZ     ??0010
0069+05                 DCR     B
006A+C26300             JNZ     ??0009
006D+15                 DCR     D
006E+C26100             JNZ     ??0008

                DONE?:  ;IS ALL THE TRAFFIC THROUGH ON LULLABYE?
                        TREAD?  LULL0,NOTDONE   ;TREADLE 0?
0071+DB01               IN      TRINP
0073+E601               ANI     1 SHL LULL0
0075+CA7F00             JZ      ??0011
0078+D301               MVI     A,1 SHL LULL0
007A+D301               OUT     TROUT
007C+C39000             JMP     NOTDONE
                        TREAD?  LULL1,NOTDONE   ;TREADLE 1?
007F+DB01               IN      TRINP
0081+E602               ANI     1 SHL LULL1
0083+CA8D00             JZ      ??0012
0086+3E02               MVI     A,1 SHL LULL1
0088+D301               OUT     TROUT
008A+C39000             JMP     NOTDONE
                        ;NEITHER TREADLE IS SET, CYCLE
                        RETRY   CYCLE           ;FOR ANOTHER LOOP
008D+C30000             JMP     CYCLE
```

Listing 9-3c. Algorithm with Generated Instructions

Macro-based languages of this sort can easily incorporate debugging facilities. In the case of Hornblower, Inc., the principal algorithms are constructed and tested in the CP/M environment by including debugging traces within each macro. In each case, a debug flag is tested and, if true, machine code is generated to trace the operation at the console, rather than actually executing the input/output calls.

Listing 9-4 shows the modification required to the INTER.LIB file to include the debugging code. Although only the SETLITE macro is shown, similar coding is easily included for the remaining macros. Listing 9-4 includes the debug flag at the beginning of the library, initially set to FALSE, along with the appropriate equates for CP/M system calls. If the debug flag is set to true by the application programmer, special trace calls are included. For example, the setlite macro constructs a message of the form

```
DIR changing to COLOR
```

where DIR and COLOR are the parameters sent to the macro. If debug remains false in the application program, this trace code is not assembled.

```
;       macro library for basic intersection
;
;       global definitions for debug processing
true    equ     0ffffh  ;value of true
false   equ     not true;value of false
debug   set     false   ;initially false
bdos    equ     5       ;entry to cp/m bdos
rchar   equ     1       ;read character function
wbuff   equ     9       ;write buffer function
cr      equ     0dh     ;carriage return
lf      equ     0ah     ;line feed
;
;       input/output ports for light and clock
light   equ     00h     ;traffic light control
clock   equ     03h     ;24 hour clock (0,1,...,23)
;
;       bit positions for traffic light control
nsbits  equ     4       ;north south bits
ewbits  equ     0       ;east west bits
;
;       constant values for the light control
off     equ     0       ;turn light off
red     equ     1       ;value for red light
yellow  equ     2       ;value for yellow light
green   equ     3       ;green light
;
setlite macr    dir,color
;;      set light given by "dir" to color given by "color"
        if      debug   ;;print info at console
        local   setmsg,pastmsg
        mvi     c,wbuff ;;write buffer function
        lxi     d,setmsg
        call    bdos    ;;write the trace info
        jmp     pastmsg
setmsg: db      cr,lf
        db      '&DIR changing to &COLOR$'
pastmsg:
        exitm
        endif
        mvi     a,color shl dir&bits    readied
        out     light   ;;sent in proper bit position
        endm
;
;       (remaining macros are identical to the previous figure,
;       but each contains trace information similar to "setlite")
;
```

Listing 9-4. Library Segment with Debug Facility

Listing 9-5a shows an application program for an intersection where the debug flag is set to TRUE after the macro library is included. As a result, each macro expansion assembles a call to the CP/M operating system to trace the light direction and color change, skipping the machine code that is eventually assembled to drive the actual Hornblower hardware.

The application programmer then uses CP/M to trace the operation of the algorithm, resulting in the printout shown in Listing 9-5b. Each trace line corresponds to a SETLITE call with a specific direction and color, with the appropriate wait time between printouts.

```
0100                    ORG     100H    ;READY FOR THE DEBUG RUN
                        MACLIB  INTER   ;BASIC MACRO LIBRARY
FFFF#           DEBUG   SET     TRUE    ;READY DEBUG TOGGLE

0100            CYCLE:  SETLITE NS,RED
0120                    SETLITE EW,GREEN
0142                    TIMER   10
0154                    SETLITE EW,YELLOW
0177                    TIMER   2
1089                    SETLITE EW,RED
01A9                    SETLITE NS,GREEN
01CB                    TIMER   10
01DD                    SETLITE NS,YELLOW
0200                    TIMER   2
0212                    RETRY   CYCLE
```

Listing 9-5a. Sample Intersection Program with Debug

```
NS changing to RED
EW changing to GREEN
EW changing to YELLOW
EW changing to RED
NS changing to GREEN
NS changing to YELLOW
NS changing to RED
EW changing to GREEN
EW changing to YELLOW
EW changing to RED
        . . .
```

Listing 9-5b. Debug Trace Printout

Upon completion of the initial debugging under CP/M, the SET statement in the application program is removed—the ORG can be removed as well—and the program is reassembled. This time, the CP/M traces are not included because the debug flag remains FALSE. As a result, the actual Hornblower hardware interface is assembled instead. The newly assembled program is then placed into PROM in the controller box for that intersection and tested in its target environment.

This approach to macro based language facilities provides a simple tool for rapid development and debugging of programs where high-level languages are not available, but a measure of machine independence is required. The macros are easy to develop, and the application programs are simple to write and debug.

9.2 Machine Emulation

A second application of macro processing is in the emulation of a machine operation code set that is different from the 8080 microprocessor. In particular, a machine architecture is selected, based on an existing or fictitious operation code set, and a macro is written for each opcode, taking the general form:

```
op      MACRO      d-1,d-2,. . .,d-n
        opcode emulation
        ENDM
```

where op is a mnemonic instruction in the emulated machine, and the dummy parameters d-1 through d-n represent the optional operands required by op. The macro body includes 8080 instructions that carry out the operation on the 8080 microprocessor. This means the instructions within the macro body perform the same function as the op with its arguments on the emulated machine.

Upon completion of the opcode macro definitions, a program can be written using these opcodes. These opcodes expand to the equivalent 8080 instructions but perform the emulated machine operations.

For example, consider the situation encountered by Nachtflieger Maschinewerke, an internationally famous manufacturer and distributor of automated machining equipment. Though incorporating microprocessors in controlling their equipment, Nachtflieger expects to build a custom LSI processor for their future products. The processor, called the KDF-10, will be used primarily as an analog sensing and control element in a larger electronic environment. As a result, the KDF-10 word size must accommodate digital values corresponding to analog signals of up to 12 bits. To allow computations on these 12-bit values, Nachtflieger engineers are going to allow a full 16-bit word in the KDF-10, along with a number of primitive operations on these values. Externally, the KDF-10 will provide four analog-to-digital input ports (A-D) that can be read by KDF-10 programs, along with four digital-to-analog output ports (D-A) that can be written by the program. The KDF-10 will automatically perform the A-D and D-A conversion at these ports.

Being forward thinkers, the engineers at Nachtflieger have designed the KDF-10 as a stack machine, similar in concept to the Hewlett-Packard HP-65 handheld programmable calculator, where data can be loaded to the top of a stack of data elements, automatically pushing existing elements deeper onto the stack. Similar to the Reverse Polish Notation (RPN) of an HP-65, arithmetic on the KDF-10 will be performed on the topmost stacked elements, automatically absorbing the stacked operands as the arithmetic is performed. The designers settled on the following three-character operation codes for the KDF-10:

SIZ n	reserves n 16-bit elements as the maximum size of the KDF-10 operand stack. This operation code must be provided at the beginning of the program.
RDM i	reads the analog signal from input port i (0, 1, 2, or 3) to the top of the stack.
WRM o	writes the digital value from the top of the stack to the D-A output port given by o (0, 1, 2, or 3). The value at the stack top is removed.
DUP	duplicates the top of the KDF-10 stack.
SUM	adds the top two elements of the KDF-10 stack. Both operands are removed, and the resulting sum is placed on the top of the stack.

LSR n	performs a logical shift of the topmost stacked element to the right by n bits (1, 2,. . .,15), replacing the original operand by the shifted result. LSR n performs a division of the topmost stacked value by the divisor 2 to the n power.
JMP a	branches directly to the program address given by label a.

Because the KDF-10 does not exist, except in the minds of the Nachtflieger engineers, the software designers decided to use the macro facilities of MAC to emulate the KDF-10, using the 8080 microcomputer.

Listing 9-6 shows an example of a program for the KDF-10 that was processed by MAC using the macro library defined by the Nachtflieger software group. In this situation, the KDF-10 is connected to four temperature sensors attached at strategic places on the machining equipment. The program continuously reads the four input values from the A-D ports and computes their average value by summing and dividing by four. This average value is sent to D-A output port 0 where it is used to set environmental controls.

```
                ;       AVERAGE THE VALUES WHICH ARE READ FROM ANALOG
                ;       INPUT PORTS, WRITE THE RESULTING VALUE TO ALL
                ;       THE D-A OUTPUT PORTS.
                ;
                        MACLIB  STACK   ;READ THE STACK MACHINE OPCODES
0000                    SIZ     20      ;CREATE 20 LEVEL WORKING STACK
012E            LOOP:   RDM     0       ;READ A-D PORT 0
0134                    RDM     1       ;READ A-D PORT 1
0136                    RDM     2       ;READ A-D PORT 2
013A                    RDM     3       ;READ A-D PORT 3

                ;       ALL FOUR VALUES ARE STACKED, ADD THEM UP
013E                    SUM             ;AD3+AD2
0140                    SUM             ;(AD3+AD2)+AD1
0142                    SUM             ;((AD3+AD2)+AD1)+AD0

                ;       SUM IS AT TOP OF THE STACK, DIVIDE BY 4
0144                    LSR     2       ;SHIFT RIGHT TWO = DIV BY 4
0152                    WRM     0       ;WRITE RESULT TO D-A PORT 0
0156 C32E01             JMP     LOOP    ;GO GET ANOTHER SET OF VALUES
```

Listing 9-6. A-D Averaging Program Using Stack Machine

As shown in Listing 9-6, the program begins by reserving a stack of 20 elements, a much larger stack than required for this application, since a maximum of four elements are actually stacked. The program then cycles following LOOP, where the values are read and processed. The four operations RDM 0, RDM 1, RDM 2, and RDM 3 read all four temperature sensors, placing their data values in the stack. The three SUM operations that follow the read operations perform pairwise addition of the temperature values, producing a single sum at the top of the stack. Because the average value is wanted, the LSR 2 operator is applied to the stack top to perform the division by four. Finally, the resulting average is sent to the D-A port using the WRM 0 operation code. Control then transfers back to LOOP, where the entire operation is performed again.

Because Nachtflieger designers are emulating KDF-10s using 8080s, they have created the macro library file, called STACK.LIB, as shown in Listing 9-7. A macro is shown in this listing for each of the KDF-10 opcodes, starting with the SIZ operator. In this case, the program origin is set, since this must be the first opcode in the program, and the stack area is reserved. Note that double words of storage are reserved because a 16-bit word size is assumed. The DUP, SUM, and LSR operators follow the SIZ macro. In each case, the KDF-10 stack top is assumed to be in 8080's HL register pair. Further, each operation that pushes the KDF-10 stack causes the element in the 8080 HL pair to be pushed to the 8080 memory area reserved by the SIZ opcode.

```
siz     macro size
;;      set "org" and create stack
        local   stack   ;;label on the stack
        org     100h    ;;at base of TPA
        lxi     sp,stack
        jmp     stack   ;;past stack
        ds      size*2  ;;double precision
stack:  endm
;
dup     macro
;;      duplicate top of stack
        push    h
        endm
;
```

Listing 9-7. Stack Machine Opcode Macros

```
sum     macro
;;      add the top two stack elements
        pop     d       ;;top-1 to de
        dad     d       ;;back to hl
        endm
;
lsr     macro   len
;;      logical shift right by len
        rept    len     ;;generate inline
        xra     a       ;;clear carry
        mov     a,h
        rar             ;;rotate with high 0
        mov     h,a
        mov     a,l
        rar
        mov     l,a     ;;back with high bit
        endm
        endm
;
adc0    equ     1080h   ;a-d converter 0
adc1    equ     1082h   ;a-d converter 1
adc2    equ     1084h   ;a-d converter 2
adc3    equ     1086h   ;a-d converter 3
;
dac0    equ     1090h   ;d-a converter 0
dac1    equ     1092h   ;d-a converter 1
dac2    equ     1094h   ;d-a converter 2
dac3    equ     1096h   ;d-a converter 3
;
rdm     macro   ?c
;;      read a-d converter number "?c"
        push    h       ;;clear the stack
;       read from memory mapped input address
        lhld    adc&?c
        endm
;
wrm     macro   ?c
;;      write d-a converter number "?c"
        shld    dac&?c  ;;value written
        pop     h       ;;restore stack
        endm
```

Listing 9-7. (continued)

The DUP opcode simply pushes the HL register pair to memory since the HL pair is not altered in the 8080 during this operation. In the case of the SUM operator, it is assumed that the KDF-10 programmer has somehow loaded two values to the KDF-10 stack. So the HL registers contain the most recently loaded value, and the 8080 memory stack contains the next-to-most recently stacked value. The POP D operation loads the second operand to the DE pair in the 8080 CPU. Then the topmost value and next to top value are added, using the DAD D operation. The resulting operand goes into the HL register pair. This is necessary in the KDF-10 emulation because the top of the KDF-10 stack is located in the 8080's HL register pair.

The LSR opcode is more complicated. The values must go through the accumulator because the 8080 does not support a double precision (16-bit) right shift of the HL register pair. Thus, the LSR macro contains a REPT loop that generates inline machine code for each right shift. The inline machine code performs the right shift by first clearing the carry (XRA A), followed by a high-order right shift by one bit (MOV A,H followed by RAR), then by a low-order bit shift (MOV A,L followed by RAR). Note that an intermediate bit can move from the high-order byte to the low-order byte using the carry between high- and low-order byte shifts.

In Listing 9-7, the RDM and WRM operation codes are defined by memory-mapped input/output operations. That is, memory locations 1080H through 1087H are intercepted external to the 8080 microprocessor and treated as external read operations. Thus, a load from locations 1080H and 1081H to HL is treated as a read from A-D device 0, rather than from RAM. This operation is simple to perform in the KDF-10 emulation because all program addresses are assumed to be below 1000H, so any 8080 address bus values beyond 1000H must be memory mapped I/O.

As a result, ADC0 through ADC3 correspond to the locations where A-D values 0 through 3 are obtained. Similarly, the D-A output values that are written to locations 1090H through 1097H are intercepted as memory mapped output values that are sent to the D-A converters rather than to RAM.

The RDM instruction is emulated by simply performing an LHLD from the appropriate memory mapped input address, constructed through concatenation of the dummy parameter. The HL value is first pushed because the KDF-10 RDM opcode performs this task automatically. Then the new value is loaded into the HL register pair.

The WRM opcode definition is similar, except the value to write is assumed to reside at the top of the KDF-10 stack and thus appears in the 8080 HL register pair. The value is written to the memory mapped output location, and the value is removed from the HL pair by restoring HL from the 8080 stack.

To see the actual code generated by each of these macros, Listing 9-8 shows the same averaging program as given in Listing 9-6, except that the generated 8080 instructions are interspersed throughout the listing file. Listing 9-8 is the usual output from MAC; Listing 9-6 was generated using the parameter -M, which suppresses generated mnemonics. Compare Listings 9-6, 9-7, and 9-8, so that you understand the macro expansion processes.

```
                ;       AVERAGE THE VALUES WHICH ARE READ FROM ANALOG
                ;       INPUT PORTS, WRITE THE RESULTING VALUE TO ALL
                ;       THE D-A OUTPUT PORTS.
                ;
                        MACLIB  STACK   ;READ THE STACK MACHINE OPCODES
                        SIZ     20      ;CREATE 20 LEVEL WORKING STACK
0100+                   ORG     100H
0100+312E01             LXI     SP,??0001
0103+C32E01             JMP     ??0001
0106+                   DS      20*2
                LOOP:   RDM     0       ;READ A-D PORT 0
012E+E5                 PUSH    H
012F+2A8010             LHLD    ADC0
                        RDM     1       ;READ A-D PORT 1
0132+E5                 PUSH    H
0133+2A8210             LHLD    ADC1
                        RDM     2       ;READ A-D PORT 2
0136+E5                 PUSH    H
0137+2A8410             LHLD    ADC2
                        RDM     3       ;READ A-D PORT 3
013A+E5                 PUSH    H
013B+2A8610             LHLD    ADC3
```

Listing 9-8. Averaging Program with Expanded Macros

```
                ;       ALL FOUR VALUES ARE STACKED, ADD THEM UP
                        SUM             ;AD3+AD2
013E+D1                 POP     D
013F+19                 DAD     D
                        SUM             ;(AD3+AD2)+AD1
0140+D1                 POP     D
0141+19                 DAD     D
                        SUM             ;((AD3+AD2)+AD1)+AD0
0142+D1                 POP     D
0143+19                 DAD     D
                ;       SUM IS AT TOP OF THE STACK, DIVIDE BY 4
                        LSR     2       ;SHIFT RIGHT TWO = DIV BY 4
0144+AF                 XRA     A
0145+7C                 MOV     A,H
0146+1F                 RAR
0147+67                 MOV     H,A
0148+7D                 MOV     A,L
0149+1F                 RAR
014A+6F                 MOV     L,A
014B+AF                 XRA     A
014C+7C                 MOV     A,H
014D+1F                 RAR
014E+67                 MOV     H,A
014F+7D                 MOV     A,L
0150+1F                 RAR
0151+6F                 MOV     L,A
                        WRM     0       ;WRITE RESULT TO D-A PORT 0
0152+229010             SHLD    DAC0
0155+E1                 POP     H
0156 C32E01             JMP     LOOP    ;GO GET ANOTHER SET OF VALUES
```

Listing 9-8. (continued)

A problem arose at Nachtflieger MW, however, that had to be rectified. Although programs could be effectively written for the KDF-10 computer using the 8080 emulation, they could not be effectively debugged. The program in Listing 9-8, for example, could be tested under the CP/M Dynamic Debugging Tool (see CP/M documentation), but the program required monitoring and tracing at the 8080 machine code level. It became clear that higher level debugging tools were necessary.

As a result, Nachtflieger designers added several pseudo opcodes that allow debugging traces. The opcodes can be interspersed in the program and selectively enabled and disabled, depending on the debugging needs. In production, all debugging traces are disabled, resulting only in absolute port I/O. The additional debugging opcodes are listed below.

PRN msg	Print the message given by "msg" at the debugging console whenever the print trace is enabled. The message must be enclosed in angle brackets.
DMP	Print the value of the top element in the KDF-10 stack in hexadecimal.
TRT t	Set machine code trace option to true. Each time a KDF-10 machine operation is executed, the opcode is printed, followed by the approximate KDF-10 machine code address, followed by the top two elements of the KDF-10 stack, in the format: OPC oploc top top′ where OPC is the opcode, oploc is the location, top is the top element, and top′ is the second to the top element, all in hexadecimal notation.
TRF t	Disable the machine code trace. Only the KDF-10 instructions that physically appear between the TRT and TRF opcodes are shown in the trace.
TRT p	Enable the print/read trace. PRN opcodes that follow produce output at the debugging console, and are otherwise treated as comments. Further, RDM and WRM opcodes prompt and display data at the debugging console.
TRF p	Disable the print/read trace. Only the PRN, RDM, and WRM instructions that physically appear between TRT and TRF interact with the console.

The traces are disabled at the beginning of the program and must be explicitly enabled with TRT opcodes.

```
                ;       AVERAGING PROGRAM WITH INTERSPERSED DEBUG CODE
                ;
                        MACLIB DSTACK ;READ THE STACK MACHINE OPCODES
0000                    SIZ    20     ;CREATE 20 LEVEL WORKING STACK
0103                    TRT    T      ;MACHINE CODE TRACE ON
0103                    TRT    P      ;PRINT TRACE ON
0103                    PRN    <TRACE FOR AVERAGING PROGRAM>
012E            LOOP:   RDM    0      ;READ A-D PORT 0
01F0                    DMP           ;WRITE TOP OF STACK
022C                    RDM    1      ;READ A-D PORT 1
0267                    DMP           ;WRITE TOP OF STACK
026A                    RDM    2      ;READ A-D PORT 2
02A5                    DMP           ;WRITE TOP OF STACK
02A8                    RDM    3      ;READ A-D PORT 3
02E3                    DMP           ;WRITE TOP OF STACK
02E6                    PRN    <FOUR VALUES HAVE BEEN READ>

                ;       ALL FOUR VALUES ARE STACKED, ADD THEM UP
0310                    SUM           ;AD3+AD2
0324                    DMP           ;WRITE FIRST SUM
0327                    SUM           ;(AD3+AD2)+AD1
033B                    DMP           ;WRITE SECOND SUM
033E                    SUM           ;((AD3+AD2)+AD1)+AD0
0352                    PRN    <VALUES HAVE BEEN ADDED>
0378                    DMP           ;WRITE SUM OF VALUES

                ;       SUM IS AT TOP OF THE STACK, DIVIDE BY 4
037B                    LSR    2      ;SHIFT RIGHT TWO = DIV BY 4
0389                    PRN    <AVERAGE VALUE CALCULATED>
03B1                    DMP           ;WRITE AVERAGE VALUE
03B4                    WRM    0      ;WRITE RESULT TO D-A PORT 0
03EE                    BRN    LOOP   ;GO GET ANOTHER SET OF VALUES
03F1                    XIT           ;EMIT EXIT CODE
```

Listing 9-9. Averaging Program with Debugging Statements

Listing 9-9 shows the averaging program of Listing 9-6 with interspersed debugging statements. The opcodes TRT t and TRT p are executed at the beginning of the program, enabling all trace options throughout the execution. The PRN statement above the LOOP label prints the initial sign-on; the DMP statements after each read operation give the value of the A-D port. Upon completion of the four-element read, the PRN opcode indicates this fact. Each SUM operator is followed by a DMP opcode that shows the current sum. Finally, the PRN and DMP opcodes display the final average value that is being sent to D-A port 0. The XIT opcode shown at the end of the program is discussed below.

Listing 9-10 shows the execution of the averaging program under DDT. Note that the program headings appear at the points in the program where PRN opcodes are placed. Further, the console is prompted for input in the case of an RDM opcode, giving the absolute memory mapped input address in decimal, while the WRM instruction produces a "D-A OUTPUT . ." message that shows the absolute memory mapped output address and the data that is written.

The opcodes are also traced showing the opcode mnemonic, address, and top two stacked elements. The RDM trace at the beginning, for example, shows the instruction address 01AD, which is in the range of the first RDM of Listing 9-9 (012E to 01EF), and is followed by the two values 0111 (the value just read) and C21D (garbage value, because only one element is stacked). The trace is easily followed at the KDF-10 level, showing each value that is read in and the operations performed upon these values. Upon completion of the debugging process under CP/M, the TRT opcodes are removed and the program is reassembled, leaving only the 8080 instructions required in the production machine. Nachtflieger systems engineers then take the resulting program and test its operation in a hardware environment.

```
A>ddt aver.hex
DDT VERS 1.4
NEXT PC
0406 0000
-g100

TRACE FOR AVERAGING PROGRAM
A-D INPUT AT 4224 111
RDM 01AD 0111 C21D
(TOP)= 0111
A-D INPUT  AT 4226 222
RDM 0255 0222 0111
(TOP)= 0222
A-D INPUT  AT 4228 555
RDM 0293 0555 0222
(TOP)= 0555
A-D INPUT  AT 4230 444
RDM 02D1 0444 0555
(TOP)= 0444
FOUR VALUES HAVE BEEN READ
SUM 0312 0999 0222
(TOP)= 0999
SUM 0329 0BBB 0111
(TOP)= 0BBB
SUM 0340 0CCC C21D
VALUES HAVE BEEN ADDED
(TOP)= 0CCC
AVERAGE VALUE CALCULATED
(TOP)= 0333
D-A OUTPUT AT 4240 0333
WRM 03DC 793B C21D
A-D INPUT  AT 4224
```

Listing 9-10. Sample Execution of AVER Using DDT

Nachtflieger engineers quickly realized that the KDF-10 design had a number of deficiencies due to the paucity of arithmetic operators and the total absence of conditional branching instructions. Further, there was no provision for variable storage other than the stack. Thus, the KDF-11 naturally evolved from the KDF-10, incorporating these features. Table 9-1 lists the operation codes of the KDF-11.

Table 9-1. KDF-11 Operation Codes

Code	*Meaning*
DCL v,n	Declare (reserve) storage for a variable by the name v, with optional size n. If n is omitted, then n – 1 is assumed. All DCL opcodes must follow the XIT opcode given below.
LIT c	Load the value of the literal constant c to the top of the KDF-11 stack.
VAL v,i,c	Load the value of the variable v optionally indexed by the variable i with the optional constant offset c. VAL V loads the value of V to the top of the stack. VAL V,I loads the value located at the address of V plus the index value contained in I. VAL V,I,3 loads the value at location V plus the index I, plus the constant index 3. In all cases, the value is placed at the top of the KDF-11 stack.
STO v,i,c	Store the value obtained from the KDF-11 stack to the address given by v, plus the optional index i, plus the optional constant index given by c. The top element of the KDF-11 stack is removed.
DIF	Subtract the top element of the KDF-11 stack from the next-to-top element of the stack and replace both operands by their difference.
GEQ a	Test the next-to-top element (top') against the top of stack element (top), and branch to the label given by "a" if top' is greater than or equal to top. If not, program control continues to the next opcode in sequence.
BRN a	Replace the JMP instruction in the KDF-10 architecture to allow complete separation of the KDF-11 and 8080 machines.

Listing 9-11 gives the macro library that was constructed by the Nachtflieger software group for KDF-11 machine emulation. More than half of the macro library implements trace and debugging functions. The remaining components implement the KDF-11 opcodes themselves. Each major section of this macro library, called DSTACK.LIB, is briefly described below, followed by an example of its use.

```
;       macro library for a zero address machine
;       ****************************************
;       *       begin trace/dump utilities     *
;       ****************************************
bdos    equ     0005h   ;system entry
rchar   equ     1       ;read a character
wchar   equ     2       ;write character
wbuff   equ     9       ;write buffer
tran    equ     100h    ;transient program area
data    equ     1100h   ;data area
cr      equ     0dh     ;carriage return
lf      equ     0ah     ;line feed
;
debugt  set     0       ;;trace debug set false
debugp  set     0       ;;print debug set false
;
prn     macro   pr
;;      print message 'pr' at console
        if      debugp  ;;print debug on?
        local   pmsg,msg        ;;local message
        jmp     pmsg            ;;around message
msg:    db      cr,lf           ;;return carriage
        db      '&PR$'          ;;literal message
pmsg:   push    h       ;;save top element of stack
        lxi     d,msg   ;;local message address
        mvi     c,wbuff ;;write buffer 'til $
        call    bdos    ;;print it
        pop     h       ;;restore top of stack
        endif           ;;end test debugp
        endm
;
ugen    macro
;;      generate utilities for trace or dump
        local   psub
        jmp     psub    ;;jump past subroutines
@ch:    ;;write character in reg-a
        mov     e,a
        mvi     c,wchar
        jmp     bdos    ;;return thru bdos
;;
@nb:    ;;write nibble in reg-a
        adi     90h
        daa
        aci     40h
        daa
        jmp     @ch     ;;return thru @ch
;;
```

Listing 9-11. Stack Machine Macro Library

```
@hx:    ;;write hex value in reg-a
        push    psw     ;;save low byte
        rrc
        rrc
        rrc
        rrc
        ani     0fh     ;;mask high nibble
        call    @nb     ;;print high nibble
        pop     psw
        ani     0fh
        jmp     @nb     ;;print low nibble
;;
@ad     ;;write address value in hl
        push    h       ;;save value
        mvi     a,' '   ;;leading blank
        call    @ch     ;;ahead of address
        pop     h       ;;high byte to a
        mov     a,h
        push    h       ;;copy back to stack
        call    @hx     ;;write high byte
        pop     h
        mov     a,l     ;;low byte
        jmp     @hx     ;;write low byte
;
@in:    ;;read hex value to hl from console
        mvi     a,' '   ;;leading space
        call    @ch     ;;to console
        lxi     h,0     ;;starting value
@in0:   push    h       ;;save it for char read
        mvi     c,rchar ;;read character function
        call    bdos    ;;read to accumulator
        pop     h       ;;value being built in hl
        sui     '0'     ;;normalize to binary
        cpi     10      ;;decimal?
        jc      @inl    ;;carry if 0,1,...,9
;;      may be hexadecimal a,...,f
        sui     'A'-'0'-10
        cpi     16      ;;a through f?
        rnc             ;;return with assumed cr
@inl:   ;;in range, multiply by 4 and add
        rept    4
        dad     h       ;;shift 4
        endm
        ora     l       ;;add digit
        mov     l,a     ;;and replace value
        jmp     @in0    ;;for another digit
;;
```

Listing 9-11. (continued)

```
psub:
ugen    macro
;;      redef to include once
        endm
        ugen    ;;generate first time
        endm
;       ****************************************
;       *       end of trace/dump utilities    *
;       *       begin trace (only) utilities   *
;       ****************************************
trace   macro   code,mname
;;      trace macro given by mname,
;;      at location given by code
        local   psub
        ugen            ;;generate utilities
        jmp     psub
@t1:    ds      2       ;;temp for reg-1
@t2:    ds      2       ;;temp for reg-2
;;
@tr:    ;;trace macro call
;;      bc=code address, de=message
        shld    @t1     ;;store top reg
        pop     h       ;;return address
        xthl            ;;reg-2 to top
        shld    @t2     ;;store to temp
        push    psw     ;;save flags
        push    b       ;;save ret address
        mvi     c,wbuff ;;print buffer func
        call    bdos    ;;print macro name
        pop     h       ;;code address
        call    @ad     ;;printed
        lhld    @t1     ;;top of stack
        call    @ad     ;;printed
        lhld    @t2     ;;top-1
        call    @ad     ;;printed
        pop     psw     ;;flags restored
        pop     d       ;;return address
        lhld    @t2     ;;top-1
        push    h       ;;restored
        push    d       ;;return address
        lhld    @t1     ;;top of stack
        ret
;;
```

Listing 9-11. (continued)

```
psub:   ;;past subroutines
;;
trace   macro   c,m
;;      redefined trace, uses @tr
        local   pmsg,msg
        jmp     pmsg
msg:    db      cr,lf   ;;cr,lf
        db      '&M$'   ;;mac name
pmsg:
        lxi     b,c     ;;code address
        lxi     d,msg   ;;macro name
        call    @tr     ;;to trace it
        endm
;;      back to original macro level
        trace   code,mname
        endm
;
trt     macro   f
;;      turn on flag "f"
debug&f         set     1       ;;print/trace on
        endm
;
trf     macro   f
;;      turn off flag "f"
debug&f         set     0       ;;trace/print off
        endm
;
```

Listing 9-11. (continued)

```
?tr     macro   m
;;      check debugt toggle before trace
        if      debugt
        trace   %$,m
        endm
;       ****************************************
;       *       end trace (only) utilities     *
;       *       begin dump (only) utilities    *
;       ****************************************
dmp     macro   vname,n
;;      dump variable vname for
;;      n elements (double bytes)
        local   psub    ;;past subroutines
        ugen            ;;gen inline routines
        jmp     psub    ;;past local subroutines
@dm:    ;;dump utility program
;;      de=msg address, c=element count
;;      hl=base address to print
        push    h       ;;base address
        push    b       ;;element count
        mvi     c,wbuff ;;write buffer func
        call    bdos    ;;message written
@dm0:   pop     b       ;;recall count
        pop     h       ;;recall base address
        mov     a,c     ;;end of list?
        ora     a
        rz              ;;return if so
        dcr     c       ;;decrement count
        mov     e,m     ;;next item (low)
        inx     h
        mov     d,m     ;;next item (high)
        inx     h       ;;ready for next round
        push    h       ;;save print address
        push    b       ;;save count
        xchg            ;;data ready
        call    @ad     ;;print item value
        jmp     @dm0    ;;for another value
;;
@dt:    ;;dump top of stack only
        prn     <(top)=>        ;;"(TOP)="
        push    h
        call    @ad             ;;value of hl
        pop     h               ;;top restored
        ret
;;
```

Listing 9-11. (continued)

```
psub:
;;
dmp     macro   ?v,?n
;;      redefine dump to use @dm utility
        local   pmsg,msg
;;      special case if null parameters
        if      nul vname
;;      dump the top of the stack only
        call    @dt
        exitm
        endif
;;      otherwise dump variable name
        jmp     pmsg
msg:    db      cr,lf   ;;crlf
        db      '&?V=$' ;;message
pmsg:   adr     ?v      ;;hl=address
active  set     0       ;;clear active flag
        lxi     d,msg   ;;message to print
        if      nul ?n  ;;use length 1
        mvi     c,1
        else
        mvi     c,?n
        endif
        call    @dm     ;;to perform the dump
        endm            ;;end of redefinition
        dmp     vname,n
        endm
;
;       ****************************************
;       *       end dump (only) utilities,     *
;       *       begin stack machine opcodes    *
;       ****************************************
active  set     0       ;active register flag
;
siz     macro   size
        org     tran    ;;set to transient area
;;      create a stack when "xit" encountered
@stk    set     size    ;;save for data area
        lxi     sp,stack
        endm
;
```

Listing 9-11. (continued)

```
save    macro
;;      check to ensure "enter" properly set up
        if      stack   ;;is it present?
        endif
save    macro   ;;redefine after initial reference
        if      active  ;;element in hl
        push    h       ;;save it
        endif
active  set     1       ;;set active
        endm
        save
        endm
;
rest    macro
;;      restore the top element
        if      not active
        pop     h       ;;recall to hl
        endif
active  set     1       ;;mark as active
        endm
;
clear   macro
;;      clear the top active element
        rest            ;;ensure active
active  set     0       ;;cleared
        endm
;
dcl     macro   vname,size
;;      label the declaration
vname:
        if      nul size
        ds      2       ;;one word req'd
        else
        ds      size*2  ;;double words
        endm
;
lit     macro   val
;;      load literal value to top of stack
        save            ;;save if active
        lxi     h,val   ;;load literal
        ?tr     lit
        endm
;
```

Listing 9-11. (continued)

```
adr     macro   base,inx,con
;;      load address of base, indexed by inx,
;;      with constant offset given by con
        save            ;;push if active
        if      nul inx&con
        lxi     h,base  ;;address of base
        exitm           ;;simple address
        endif
;;      must be inx and/or con
        if      nul inx
        lxi     h,con*2 ;;constant
        else
        lhld    inx     ;;index to hl
        dad     h       ;;double precision inx
        if      not nul con
        lxi     d,con*2 ;;double const
        dad     d       ;;added to inx
        endif           ;;not nul con
        endif           ;;nul inx
        lxi     d,base  ;;ready to add
        dad     d       ;;base+inx*2+con*2
        endm
;
val     macro   b,i,c
;;      get value of b+i+c to hl
;;      check simple case of b only
        if      nul i&c
        save            ;;push if active
        lhld    b       ;;load directly
        else
;;      "adr" pushes active registers
        adr     b,i,c   ;;address in hl
        mov     e,m     ;;low order byte
        inx     h
        mov     d,m     ;;high order byte
        xchg            ;;back to hl
        endif
        ?tr     val     ;;trace set?
        endm
;
```

Listing 9-11. (continued)

```
sto     macro   b,i,c
;;      store the value of the top of stack
;;      leaving the top element active
        if      nul i&c
        rest            ;;activate stack
        shld    b       ;;stored directly to b

        else
        adr     b,i,c
        pop     d       ;;value is in de
        mov     m,e     ;;low byte
        inx     h
        mov     m,d     ;;high byte
        endif
        clear           ;;mark empty
        ?tr     sto     ;;trace?
        endm
sum     macro
        rest            ;;restore if saved
;;      add the top two stack elements
        pop     d       ;;top-1 to de
        dad     d       ;;back to hl
        ?tr     sum
        endm
;
dif     macro
;;      compute difference between top elements
        rest            ;;restore if saved
        pop     d       ;;top-1 to de
        mov     a,e     ;;top-1 low byte to a
        sub     l       ;;low order difference
        mov     l,a     ;;back to l
        mov     a,d     ;;top-1 high byte
        sbb     h       ;;high order difference
        mov     h,a     ;;back to h
;;      carry flag may be set upon return
        ?tr     dif
        endm
;
```

Listing 9-11. (continued)

```
lsr     macro   len
;;      logical shift right by len
        rest            ;;activate stack
        rept    len     ;;generate inline
        xra     a       ;;clear carry
        mov     a,h
        rar             ;;rotate with high 0
        mov     h,a
        mov     a,l
        rar
        mov     l,a     ;;back with high bit
        endm
        endm
;
geq     macro   lab
;;      jump to lab if (top-1) is greater or
;;      equal to (top) element.
        dif             ;;compute difference
        clear           ;;clear active
        ?tr     geq
        jnc     lab     ;;no carry if greater
        jz      lab     ;;zero if equal
;;      drop through if neither
        endm
;
dup     macro
;;      duplicate the top element in the stack
        rest            ;;ensure active
        push    h
        ?tr     dup
        endm
;
brn     macro   addr
;;      branch to address
        jmp     addr
        endm
;
xit     macro
        ?tr     xit     ;;trace on?
        jmp     0       ;;restart at 0000
        org     data    ;;start data area
        ds      @stk*2  ;;obtained from "siz"
stack:  endm
;
```

Listing 9-11. (continued)

```
;       *****************************************
;       *       memory mapped i/o section       *
;       *****************************************
;       input values which are read as if in memory
adc0    equ     1080h   ;a-d converter 0
adc1    equ     1082h   ;a-d converter 1
adc2    equ     1084h   ;a-d converter 2
adc3    equ     1086h   ;a-d converter 3
;
dac0    equ     1090h   ;d-a converter 0
dac1    equ     1092h   ;d-a converter 1
dac2    equ     1094h   ;d-a converter 2
dac3    equ     1096h   ;d-a converter 3
;
rwtrace         macro   msg,adr
;;      read or write trace with message
;;      given by "msg" to/from "adr"
        prn     <msg at adr>
        endm
;
rdm     macro   ?c
;;      read a-d converter number "?c"
        save            ;;clear the stack
        if      debugp  ;;stop execution in ddt
        rwtrace <a-d input >,% adc&?c
        ugen            ;;ensure @in is present
        call    @in     ;;value to hl
        shld    adc&?c  ;;simulate memory input
        else
;;      read from memory mapped input address
        lhld    adc&?c
        endif
        ?tr     rdm     ;;tracing?
        endm
;
```

Listing 9-11. (continued)

```
wrm     macro   ?c
;;      write d-a converter number "?c"
        rest            ;;restore stack
        if      debugp  ;;trace the output
        rwtrace <d-a output>,% dac&?c
        ugen            ;;include subroutines
        call    @ad     ;;write the value
        endif
        shld    dac&?c
        ?tr     wrm     ;;tracing output?
        clear           ;;remove the value
        endm
;       ****************************************
;       *       end of macro library           *
;       ****************************************
```

Listing 9-11. (continued)

The first portion of the library, which is principally concerned with debugging functions, begins with CP/M system calls, function numbers, and equates for non-graphic characters, similar to the examples given earlier. Although these values are not necessary for operation of the KDF-11, they are necessary for the debugging functions that operate when the TRT opcode is in effect. Following the CP/M equates, the toggles DEBUGT and DEBUGP are set to false (0 value), reflecting the conditions of the debugging switches given by TRT and TRF. When DEBUGT is true (1 value), machine operation codes are traced. Similarly, when DEBUGP is true, PRN, RDM, and WRM operations interact with the console.

The PRN macro, for example, produces an inline message with a call to CP/M to write the message whenever the DEBUGP toggle is true. Otherwise, the PRN produces no generated code.

The UGEN macro that follows PRN is called the first time the debugging subroutines are required by trace or print/read opcodes. When invoked, the UGEN macro produces several inline subroutines that are used throughout the debugging process.

If no trace or print/read functions are invoked during the assembly, UGEN is not invoked. Thus no inline subroutines are included for debugging. If UGEN is invoked, the subroutines shown below are included inline:

@CH	writes a single ASCII character to the console.
@NB	writes a single half byte (nibble) to the console.
@HX	writes a full hexadecimal byte value at the console.
@AD	writes a full address (double byte) value with preceding blank.
@IN	reads a hexadecimal value from the console to HL.

Upon including these subroutines, UGEN then redefines itself to an empty macro body so that the subroutines are not included on subsequent invocations of UGEN. This ensures that the inline subroutines are included only once, and only if they are required by the debugging macros.

The SIZ macro is similar to the opcode defined for the KDF-10, except that the size of the stack is saved for later declaration in the data area (see the XIT opcode). Throughout the opcode macros, the SAVE and REST macros save and restore the HL register pair, based on the ACTIVE flag. The CLEAR macro, however, marks the top element of the KDF-11 stack as deleted.

The DCL macro simply sets up the variable name VNAME as a label and follows the label by a DS that reserves the specified number of double words. The DCL opcodes must all occur at the end of the KDF-11 program, following the XIT opcode.

The LIT opcode is emulated with a macro that first SAVEs the stack top, possibly generating an HL push. The literal value is then loaded directly into the HL register pair. The ACTIVE flag is set on completion of this macro because SAVE always marks HL as active.

The ADR macro is a utility macro used in the VAL, STO, and DMP opcodes to build the address of a particular variable, with optional variable and constant offsets, in the HL register pair. Based on the optional parameters, ADR either loads the base address directly to the HL pair or constructs the address using HL and DE for indexing. Thus, the following invocations of ADR (in the left column) produce the machine code in the right column.

```
ADR   X                         LXI    H,X

ADR   X,I                       LHLD   I
                                DAD    H
                                LXI    D,X
                                DAD    D

ADR   X,I,3                     LHLD   I
                                DAD    H
                                LXI    D,6
                                DAD    D
                                LXI    D,X
                                DAD    D

ADR   X,,3                      LXI    H,6
                                LXI    D,X
                                DAD    D
```

The final address for the optionally indexed variable remains in the HL register pair. The code within the ADR macro can be improved slightly by providing a constant offset. That is, the following invocations in the left column produce the machine code in the right column by redefining the ADR macro.

```
ADR   X,I,3                     LHLD   I
                                LXI    D,X+6
                                DAD    D

ADR   X,,3                      LXI    H,X+6
```

As an exercise, redefine ADR to generate this improved machine code sequence.

The VAL macro loads a variable value to the stack. STO stores the top of stack value to memory. ADR constructs the address of the variable whenever optional indexing is specified. Otherwise, LHLD or SHLD directly accesses the variable. Again, slight improvements in generated code can be obtained by providing a constant offset with no variable index.

The opcodes LIT, VAL, and STO all end with an invocation of the ?TR macro which, as discussed above, checks the DEBUGT flag. If true, the ?TR macro invokes TRACE with the machine code address and opcode name for display at the debugging console. The ?TR macro invocation produces no machine code trace when DEBUGT is false.

The SUM opcode first invokes REST to ensure that the HL register pair contains the topmost KDF-11 element. The second to top element is then loaded to the DE pair and added to HL, producing an active KDF-11 element in HL. ACTIVE is true at this point, because REST always leaves the flag set to true.

The DIF opcode definition is similar to SUM, except that the 8080 accumulator computes the 16-bit difference between the top two KDF-11 stacked elements.

The LSR macro defines the KDF-11 logical shift right operation. The REST macro is first invoked to ensure that HL is active, followed by a repetition of the machine code required to perform a 16-bit right shift of the HL register pair. In the case of a long shift, there is a considerable amount of inline machine code for the operation. Thus, it is a useful exercise to redefine LSR, so that it generates an inline subroutine to perform the shift operation for values of LEN sufficiently large to warrant the subroutine call. Although this requires a subroutine set up and call, the amount of generated code can be reduced significantly for programs that make heavy use of the LSR operator.

The GEQ macro follows the LSR definition and allows conditional branching to the specified label address. GEQ begins by computing the difference between the top two elements of the KDF-11 stack. This has the side-effect of setting the 8080 carry bit if the next to top element exceeds the top element in the KDF-11 stack. The ?TR macro eventually leads to the @TR subroutine where the status flags (including the carry condition) are saved and restored. Otherwise, GEQ could not count on the condition of the carry flag.

Further, the 8080 A register contains the least significant byte of the difference between DE and HL, so the ORA H produces a zero result if the difference is zero. To be complete, the KDF-11 should have a complete range of conditional tests, allowing tests for equality (EQL), inequality (NEQ), less than (LSS), greater than (GTR), and less than or equal (LEQ).

The DUP opcode first ensures that the HL register pair is active, then duplicates this value by pushing the HL pair to the 8080 stack, emulating a KDF-11 stack push operation. Note that the HL pair is active at the end of the DUP macro due to the invocation of REST.

The BRN and XIT macros follow GEQ. The BRN macro simply translates to a jump instruction in the 8080. The XIT macro first invokes the ?TR macro to check for machine code tracing. A JMP 0 is then emitted, corresponding to a system restart in both CP/M and the emulated KDF-11 machine architecture. The XIT macro then produces an ORG statement that restarts the assembly process in the data area of the emulated environment (1000H, or 4096 decimal). The area reserved for the stack is then set up, followed by the declaration of the label STACK at the top of this reserved area. Note that the SAVE macro includes the statement sequence:

```
IF      STACK       ;;is it present?
ENDIF
```

which ensures that both the SIZ and XIT macros have been included in the assembly. If the XIT macro is not included, then the label STACK does not appear unless used in the KDF-11 program, and the IF STACK test produces an undefined operand (U) error. Further, if the XIT operator is used, but the SIZ is not, then the statement DS SIZ*2 within XIT produces an undefined operand message. Although these tests are by no means complete, they detect the most common errors.

Listing 9-11 also contains the definitions of both the RDM and WRM opcodes, based on the memory mapped input/output addresses defined by ADC0 through ADC3 for the A-D ports, and DAC0 through DAC3 for the D-A ports. The RWTRACE (Read-Write Trace) macro is included for tracing the RDM and WRM macros when DEBUGP is true. The MSG argument corresponds either to A-D INPUT for the RDM opcode or to D-A OUTPUT for the WRM opcode. The ADR argument corresponds to the absolute decimal address where the memory mapped input/output is taking place. Thus, RWTRACE simply constructs a trace message from its two arguments and passes this message to PRN for display at the debugging console.

The RDM macro reads the port given by the argument ?C (0, 1, 2, or 3). The HL register pair is pushed, if necessary, by the SAVE macro, leaving the active flag set for the RDM. RDM then generates an invocation of the RWTRACE macro to produce the trace message. Note that the argument "% ADC&?C" produces the numeric value ADC0, ADC1, ADC2, or ADC3, which is included in the trace message. If the % is omitted, only the name, not the value, of the input port address is printed. Following the output message, UGEN is invoked to ensure that the utility subroutines have been included inline. The call to @IN allows you to type a hexadecimal value for the simulated A-D input value. This value is subsequently stored to memory and left in the HL register pair with ACTIVE true. If DEBUGP is not set, then the RDM macro simply loads the HL register pair from the appropriate memory mapped input location. Finally, RDM invokes ?TR to check for possible opcode tracing.

The WRM opcode is similar to the RDM opcode, except that the REST macro is first invoked to ensure that the HL registers contain the top element of the KDF-11 stack. This value is displayed at the debugging console if DEBUGP is true and then sent to the appropriate memory mapped output location.

One application of the emulated KDF-11 machine shows the power of this instruction set. As a small part of a machine control system, a KDF-11 processor monitors the machine tool head motion. Nachtflieger engineers connect A-D port 0 to a KDF-11 processor that reads the instantaneous velocity of the tool head at 1 millisecond (ms) intervals.

The velocity is provided at the A-D port in micrometer (um) increments, and the processor is synchronized with the input, so that it halts until the 1 ms interval has elapsed. Nachtflieger engineers also guarantee that the tool head is in motion for no more than 100 ms before stopping. Thus, with no variations in velocity, if the tool moved at the constant rate of 256 um/ms over 50 intervals of 1 ms each, total distance traveled by the tool is

256 um/ms * 50 ms = 1280 um = 1.280 mm

During its travel, however, the instantaneous velocity of the tool head varies according to the roughness of the cut, wear on the parts, and start/stop intervals.

Nachtflieger uses the data collected during a cut to monitor these factors and displays machine operator information in both digital and analog forms. A primary function of the KDF-11 processor in this case is to collect instantaneous velocities during a single cut and hold these values for analysis as the tool returns to its starting position. Listing 9-12 shows a KDF-11 program that includes the data collection phase and an analysis phase described below.

The data collection phase of Listing 9-12 occurs between the labels MOVE? and COMP; the analysis phase is found between labels COMP and ENDF. The program is bounded by the SIZ operator at the beginning and the XIT operator at the end, followed by DCL opcodes that reserve data areas. This program also includes debugging PRN, DMP, TRT, and TRF opcodes for checking out the program.

As for the DCL statements at the end of Listing 9-12, the vector V is declared with length 100 (double bytes), which holds the collected velocities; I and X are temporary values used during the collection and analysis phase. The variable TOTAL is a result produced by the analysis, as discussed below.

The program collects data by performing the following steps. The variable I is first initialized to 0, corresponding to the first velocity V(0). The program then examines the A-D input port for the first nonzero velocity, waiting for the tool head to begin its travel. When the first nonzero velocity is read, the collection process proceeds by storing the first value at V(0). The index value I is then moved along as data items are read, with values placed into V(1), V(2), continuing until a zero value is read, indicating the tool has ended its travel.

Referring to Listing 9-12, note that the KDF-11 opcodes listed before the label MOVE? initialize the index I by loading a literal 0 value to the KDF-11 stack, followed by a store into the variable I. To follow these operations, the TRT P and TRT T traces are enabled. Note, however, that the TRF T opcode stops the machine code trace immediately before the MOVE? label.

```
                MACLIB  DSTACK   ;STACK MACHINE SIMULATION
0000            SIZ     50       ;50 LEVEL STACK
0103            TRT     P        ;TURN ON PRN TRACK
0103            TRT     T        ;TURN ON CODE TRACE
0103            PRN     <COMPUTATION OF TOOL TRAVEL DISTANCE>
0136            LIT     0        ;INITIALIZE INDEX
01D3            STO     I        ;I=0
01E8            TRF     T        ;TURN CODE TRACE OFF
        ;       LOOK FOR STARTING MOTION (NON ZERO VALUE)
        MOVE?:  ;READ A-D CONVERTER FOR NON ZERO
01E8            RDM     0
0210            STO     X        ;HOLD TEMPORARILY
0213            VAL     X        ;RELOAD FOR TEST
0216            LIT     1        ;X GEQ 1 TEST
021A            GEQ     READ     ;X GEQ 1 ?
0227            BRN     MOVE?    ;RETRY IF NOT

        READ:
022A            PRN     <STORE FIRST/NEXT VALUE>
0250            DMP     X
029C            VAL     X        ;LOAD FIRST/NEXT VALUE
029F            STO     V,I      ;STORE TO THE ITH ELEMENT
02AC            VAL     I        ;INCREMENT I
02AF            LIT     1
02B3            SUM              ;I+1
02B5            STO     I        ;I=I+1
02B8            LIT     0        ;0, FOR 0 GTR X TEST
02BB            VAL     X        ;ZERO VALUE READ?
02BF            GEQ     COMP     ;COMPUTE DISTANCE IF 0
02CC            RDM     0        ;READ ANOTHER DATA ITEM
02F4            STO     X        ;SAVE IT IN X
02F7            BRN     READ     ;TO STORE AND TEST

02FA    COMP:   PRN     <VALUES ARE LOADED>
031A            DMP     V,10
        ;       NOW COMPUTE DISTANCE TRAVELLED BY TOOL
032D            LIT     0
0330            DUP              ;TWO ZEROES
0331            STO     I        ;I=0
0334            STO     TOTAL    ;TOTAL=0
0338    GETNXT: PRN     <COMPUTING NEXT INTERVAL>
035F            DMP     I
0372            DMP     TOTAL
0389            DMP     <V,I>,2
03A3            LIT     0        ;ZERO AT END
03A6            VAL     V,I      ;AT END?
03B3            GEQ     ENDF     ;0 GEQ X(I)?
```

Listing 9-12. Program for Tool Travel Computation

```
      ;         NOT AT END OF INTERVAL, COMPUTE NEXT TRAPEZO
03C0            VAL     V,I
03CC            VAL     V,I,1       ;V(I),V(I+1)
03DD            SUM                 ;V(I)+V(I+1)
03DF            LSR     1           ;(V(I)+V(I+1)/2
03E6            VAL     TOTAL       ;READY TOTAL
03EA            SUM                 ;TOTAL=TOTAL+TRAPEZOID
03EC            STO     TOTAL       ;BACK TO SUM

03EF            VAL     I           ;I=I+1
03F2            LIT     1
03F6            SUM
03F8            STO     I           ;BACK TO I
03FB            BRN     GETNXT

03FE  ENDF:     PRN     <END OF COMPUTATION>
0420            DMP     TOTAL
0437            VAL     TOTAL       ;LOAD FOR D-A OUTPUT
043A            WRM     0           ;WRITE D-A PORT
0462            XIT
      ;
      ;         DATA AREA
1164            DCL     I           ;INDEX
1166            DCL     X           ;TEMPORARY
1168            DCL     V,100       ;VELOCITY VECTOR
1230            DCL     TOTAL       ;TOTAL DISTANCE
```

Listing 9-12. (continued)

Following the MOVE? label, A-D port 0 is read and examined for the first nonzero value. Each time the port is read, it is stored into the temporary variable X, then reloaded and examined for a zero value. Because GEQ is the only comparison operator in the KDF-11 machine, the test is "1 greater than or equal to X." Thus, the branch is taken to READ whenever X is 1 or larger.

Upon encountering the READ label, the value X (just read from port 0) is stored into V(I), where I is zero. The value of I is then incremented by loading I to the top of the KDF-11 stack, adding 1 (LIT 1, SUM), and then storing the sum back into I. After incrementing I, the program proceeds to check the end of the tool travel. X is loaded to the top of the stack, and the test 0 greater than or equal to X is performed. If the condition is true, control transfers to the label COMP, where the analysis phase begins. Otherwise, port 0 is read again, and the value is stored into the temporary X. Control then proceeds back to the READ label to store the next velocity and test for zero.

Before 100 intervals have elapsed, the RDM 0 produces a zero value that is stored into X and subsequently stored into V(I), for the current value of I. Thus, when control arrives at the label COMP, the instantaneous velocities are stored in V, terminated by a zero. At this point, the analysis of these collected velocities can take place.

The single function that takes place in the analysis section of Listing 9-12 is the computation of the distance traveled by the tool through this interval. Nachtflieger engineers have determined that it is sufficient to compute the distance traveled by the tool using the trapezoidal rule that approximates the actual distance by summing the average of each adjacent pair of velocities. The sums are formed as shown below:

$$\frac{V_0+V_1}{2} + \frac{V_1+V_2}{2} + \cdots + \frac{V_{n-1}+V_n}{2}$$

where n is the last interval to sum. Thus, for example, if the velocity is constant at 256 um/ms (which would not occur in practice), then

$$V_1 = V_2 = \cdots = V_n = 256$$

The summing formula given above reduces to 256 * n. Given the preceding example, where n = 50 ms, this formula produces the value 1.280 mm, as given earlier. The velocity values are not usually constant, so the numerical integration given by the trapezoidal rule is used to obtain an approximation.

The KDF-11 instructions shown in Listing 9-12 between the COMP and ENDF labels perform the numeric integration, given by the trapezoidal rule. The temporary I is used to index through the velocity vector V until the final zero value is encountered. For each interval, the values of two adjacent velocities are summed and divided by two. Each result is then summed into TOTAL, where the values are accumulated until the final zero velocity is discovered.

The opcode sequence immediately following COMP places a zero value at the top of the KDF-11 stack, then stores this value into both the index I and the accumulating sum given by TOTAL. Ignoring the trace opcodes, the operations following GETNXT read the starting point of the next interval to process into the stack, using VAL V,I (value of V, indexed by I). If 0 is greater than or equal to this value, then the computation is complete and control goes to the label ENDF. Otherwise, the value of V(I) is loaded to the KDF-11 stack, followed by the value of V(I+1). The loaded values are then summed (SUM) and divided by two (LSR 1), producing a value that remains in the KDF-11 stack. TOTAL is then loaded and added to this partial sum, and the result is stored back to TOTAL. The index value I is then incremented to the next interval and processing continues back at the loop header GETNXT.

Upon processing the final zero velocity, control reaches the ENDF label where the distance traveled is written to D-A output port zero. The output value is sent to external instrumentation, which processes the result and displays the distance traveled in a form that is readable by the tool operator.

Debugging statements have been placed throughout the program. These can be used to trace the program execution. Listing 9-12 also contains TRT operators that have enabled trace code generation. Thus this program, although longer than the final production version, can be used to follow execution under CP/M.

Listing 9-13 shows the execution of the program of Listing 9-12 under DDT. The messages printed at the debugging console are a result of the PRN opcodes distributed throughout the original program that were enabled through the TRT P opcode. Further, the machine code trace was only enabled for the interval of two operation codes (LIT and STO) at the beginning. To test this program, simple A-D values were supplied at the console for the velocities:

$$V_0 = 100H, V_1 = 120H, V_2 = 100H, V_3 = 80H, V_4 = 0$$

Upon detecting the final 0 value, the trace of Listing 9-13 shows the first 10 values of V (the last 5 elements are garbage values), followed by a trace of the sum operations for each interval. In each case, the pairs of values that are being added are displayed (using the DMP opcode), followed by their summed value, along with the running total. Upon completion of the distance computation, the value 320H is sent to the D-A output port and displayed at the console.

After initial checks under CP/M, Nachtflieger programmers remove the TRT and TRF statements from the KDF-11 program and reassemble, producing only the absolute input/output instructions required for machine tool control. The resulting program, which produces much less code than the debugging version, is placed into the equipment for further testing and evaluation.

Listing 9-14 also provides an example of the listing produced when all machine code operators are traced. Although the source program listing is not shown, it is identical to Listing 9-12 except that the TRF T opcode is removed. Because the complete trace is quite extensive, only a partial execution is shown in Listing 9-14.

```
A>DDT INTEG.HEX
DDT VERS 1.4
NEXT  PC
0465 0000
-G100

COMPUTATION OF TOOL TRAVEL DISTANCE
LIT 0139 0000 0F77
STO 01D6 0000 0000
A-D INPUT  AT 4224 0
A-D INPUT  AT 4224 100
STORE FIRST/NEXT VALUE
X= 0100
A-D INPUT  AT 4224 120
STORE FIRST/NEXT VALUE
X= 0120
A-D INPUT  AT 4224 100
STORE FIRST/NEXT VALUE
X= 0100
A-D INPUT  AT 4224 80
STORE FIRST/NEXT VALUE
X= 0080
A-D INPUT  AT 4224 0
STORE FIRST/NEXT VALUE
X= 0000
VALUES ARE LOADED
V= 0100 0120 0100 0080 0000 3EC0 BA11 C1C9 5EE1 5623
COMPUTING NEXT INTERVAL
I= 0000
TOTAL= 0000
V,I= 0100 0120
COMPUTING NEXT INTERVAL
I= 0001
TOTAL= 0110
V,I= 0120 0100
COMPUTING NEXT INTERVAL
I= 0002
V,I= 0100 0080
COMPUTING NEXT INTERVAL
I= 0003
TOTAL= 02E0
V,I= 0080 0000
COMPUTING NEXT INTERVAL
I= 0004
TOTAL= 0320
V,I= 0000 3EC0
END OF COMPUTATION
TOTAL= 0320
D-A OUTPUT AT 4240 0320
```

Listing 9-13. Sample Execution of Distance Using DDT

```
A>ddt integ.hex
DDT VERS 1.4
NEXT  PC
0852 0000
-g100

COMPUTATION OF TOOL TRAVEL DISTANCE
LIT 026E 0000 CAB1
STO 030B 0000 0000
A-D INPUT  AT 128 0
RDM 0344 0000 0000
STO 0359 0000 0000
VAL 036E 0000 0000
LIT 0384 0001 0000
DIF 039D FFFF 0000
GEQ 03AF FFFF 0000
A-D INPUT  AT 128 6
RDM 0344 0006 0000
STO 0359 0006 0000
VAL 036E 0006 0000
LIT 0384 0001 0006
DIF 039D 0005 0000
GEQ 03AF 0005 0000
STORE FIRST/NEXT VALUE
X= 0006
VAL 043F 0006 0000
STO 045E 016F 0000
VAL 0473 0000 0000
LIT 0489 0001 0000
SUM 049D 0001 0000
STO 04B2 0001 0001
VAL 04C7 0006 0001
A-D INPUT  AT 128 0
RDM 0501 0000 0006
STO 0516 0000 0006
LIT 052B 0001 0006
DIF 0544 0005 0001
GEQ 0556 0005 0001
STORE FIRST/NEXT VALUE
X= 0000
VAL 043F 0000 0001
STO 045F 0171 0001
VAL 0473 0001 0001
LIT 0489 0001 0001
SUM 049D 0002 0001
STO 04B2 0002 0002
VAL 04C7 0000 0002
A-D INPUT  AT 128
RDM 0501 0000 0000
```

Listing 9-14. Partial Listing of Distance with Full Trace

In summary, Nachtflieger MW derived several benefits from their emulation of the KDF series stack machines. First, there is very little cost involved in designing and altering their machine architecture. In fact, current prices for 8080 microcomputers might preclude the custom LSI version of the KDF-? machine. A second advantage of the KDF emulation is that the KDF programs are highly independent from the host processor. If a higher performance or less expensive processor becomes available to Nachtflieger, the existing programs can be used intact by changing only the macro definitions for each of the KDF opcodes and reassembling using MAC.

Finally, machine emulation through macro defined operation codes offers a distinct advantage over interpretive approaches because each opcode translates to only a few host machine operations. Interpretive execution often involves ratios of 1000 to 20,000 emulated instructions per host instruction; macro based opcodes are often in a ratio of less than 10 to 1. Further, interpretive processors usually require run-time support consisting of a predefined general purpose subroutine package that is included for each and every program. For a wide variety of microcomputer applications, machine emulation through macro defined opcodes offers distinct advantages over alternative approaches.

9.3 Program Control Structures

Macro facilities can provide program control statements that resemble those found in many high-level languages. In general, program control statements allow Boolean tests and conditional branching based on the outcome of the Boolean test. Further, label names usually provided by you as the destination of a branch are automatically generated for the particular statement.

The following paragraphs discuss three typical control statements that allow simple conditional grouping (WHEN-ENDW), controlled iteration (DO-ENDDO), and case selection (SELECT-ENDSEL). All three statements are program control facilities that allow well-structured programming, resulting in programs that are easier to write, debug, and maintain.

Two libraries are first introduced as a foundation for the discussion. The I/O library shown in Listing 9-15 allows simple character input operations along with full message output. The READ macro accepts a single character from the console keyboard and stores this character into the variable given by the parameter VAR. The WRITE macro shown in Listing 9-15 takes an ASCII message as a parameter and sends this message to the console output device preceded by a carriage return line-feed sequence. These simple I/O macros are stored in the disk in the file SIMPIO.LIB and are used in the examples that illustrate the control structures.

The second library used in the control structure examples is given in Listing 9-16. Collectively, these macros define a number of Boolean operations that are performed on 8-bit operands, providing the basic relational operations on unsigned integer values, including:

LSS	less than
LEQ	less than or equal to
EQL	equal to
NEQ	not equal to
GEQ	greater than or equal to
GTR	greater than

In all cases, the macros accept three actual parameters. The parameters consist of two data values involved in the test (X and Y), along with a program label that receives control if the Boolean test produces a true value (TL). The first operand X can be a labeled memory location containing an 8-bit value, and Y can be either a labeled 8-bit location or a literal numeric value. If the first operand X is not supplied, then the value to be tested is assumed to exist in the 8080 accumulator when the macro is entered. Thus, for example, the macro invocation

```
LSS     ALPHA,BETA,TRUECASE
```

compares the values stored at the labeled memory locations ALPHA and BETA, defined by a DS or DB statement, and transfers to the program step labeled by TRUECASE if ALPHA contains a value less than the value stored at BETA. The invocation

```
LSS     ,BETA,TRUECASE
```

is similar, but it compares the contents of the 8080 accumulator with the value stored at BETA. Finally, the invocation

```
LSS     ALPHA,34,TRUECASE
```

compares ALPHA with the literal value 34 in the relational test.

The macro TEST? is used throughout the macro library to construct the relational test by first loading the initial operand X, if necessary. The second operand type is then examined by executing an IRPC within the TEST? macro of Listing 9-16. This extracts the first character of the Y operand. This first character must be either numeric or alphabetic. If numeric, then the literal value is subtracted from the accumulator, setting the 8080 condition codes. If the first character of Y is nonnumeric, then the value is assumed to reside in memory. In this case, the HL registers are set to the Y operand and the value at Y is subtracted from the accumulator value. In any case, the 8080 condition codes are set as a result of the subtraction operation. These condition codes are then used in the individual macros to produce conditional jumps to the destination labels. These macros are collectively stored on the disk in a file named COMPARE.LIB for use in examples that follow.

```
;       macro library for simple i/o
bdos    equ     0005h   ;bdos entry
conin   equ     1       ;console input function
msgout  equ     9       ;print message til $
cr      equ     0dh     ;carriage return
lf      equ     0ah     ;line feed
;
read    macro   var
;;      read a single character into var
        mvi     c,conin ;console input function
        call    bdos    ;character is in a
        sta     var
        endm
;
write   macro   msg
;;      write message to console
        local   msgl,pmsg
        jmp     pmsg
msgl:   db      cr,lf   ;;leading crlf
        db      '&MSG'  ;;inline message
        db      '$'     ;;message terminator
pmsg:   mvi     c,msgout        ;;print message til $
        lxi     d,msgl
        call    bdos
        endm
```

Listing 9-15. Simple I/O Macro Library

```
test?   macro   x,y
;;      utility macro to generate condition codes
        if      not nul x       ;;then load x
        lda     x       ;; assumed to be in memory
        endif
        irpc    ?y,y    ;;y may be constant operand
tdig?   set     '&?Y'-'0'       ;;first char digit?
        exitm           ;;stop irpc after first char
        endm
        if      tdig? <= 9      ;;y numeric?
        sui     y       ;;yes, so sub immediate
        else
        lxi     h,y     ;;y not numeric
        sub     m       ;;so sub from memory
        endm
;
lss     macro   x,y,tl
;;      x lss than y test,
;;      transfer to tl (true label) if true,
;;      continue if test is false
        test?   x,y     ;;set condition codes
        jc      tl
        endm
;
leq     macro   x,y,tl
;;      x less than or equal to y test
        lss     x,y,tl
        jz      tl
        endm
;
eql     macro   x,y,tl
;;      x equal to y test
        test?   x,y
        jz      tl
        endm
;
neq     macro   x,y,tl
;;      x not equal to y test
        test?   x,y
        jnz     tl
        endm
;
```

Listing 9-16. Macro Library for Simple Comparison Operations

```
geq     macro   x,y,tl
;;      x greater than or equal to y test
        test?   x,y
        jnc     tl
        endm
;
gtr     macro   x,y,tl
;;      x greater than y test
        local   fl      ;;false label
        test?   x,y
        jc      fl
        dcr     a
        jnc     tl
fl:     endm
```

Listing 9-16. (continued)

Listings 9-17a and 9-17b show an example of a program that uses both the SIMPIO and COMPARE libraries. This program successively reads console characters and print messages based on the character typed. The program begins by sending the sign-on message at the label CYCLE. A character is then read and stored into X, using the READ macro. The LSS test determines whether lower- to upper-case translation is required, assuming the input is alphabetic. If X is numerically less than 61H, the value of a lower-case A, then control transfers to the label NOTRAN. Otherwise, the character is loaded to the accumulator, the lower-case bit is stripped from the character, and it is replaced in memory. Following the label NOTRAN, the character is compared with the letters A, B, C, and D. In each case, a message is typed corresponding to each letter. If one of these four letters cannot be found, the message at ERROR is typed.

```
0100                    ORG     100H
                        MACLIB  SIMPIO  ;SIMPLE IO LIBRARY
                        MACLIB  COMPARE ;COMPARISON OPERATORS
                ;
0100            CYCLE:  WRITE   <TYPE A CHARACTER FROM A TO D >
012B                    READ    X
                ;       TEST FOR LOWER CASE ALPHABETIC
0133                    LSS     X,61H,NOTRAN
                ;       ARRIVE HERE IF X IS GREATER OR EQUAL TO
                ;       A LOWER CASE A (=61H), TRANSLATE
013B 3A1102             LDA     X
013E E65F               ANI     5FH     ;CLEAR LOWER CASE BIT
0140 321102             STA     X       ;STORE BACK TO X
                NOTRAN:
                ;       NOW CHECK CASES
                ;
0143                    NEQ     X,%'A',NOTA
014B                    WRITE   <YOU TYPED AN A>
0167 C30001             JMP     CYCLE
                ;
016A            NOTA:   NEQ     X,%'B',NOTB
0172                    WRITE   <YOU TYPED A B>
018D C30001             JMP     CYCLE
                ;
0190            NOTB:   NEQ     X,%'C',NOTC
0198                    WRITE   <YOU TYPED A C>
01B3 C30001             JMP     CYCLE
                ;
01B6            NOTC:   NEQ     X,%'D',ERROR
01BE                    WRITE   <YOU TYPED A D>
01D9                    WRITE   <BYE^!>
01EB C9                 RET
                ;
01EC            ERROR:  WRITE   <NOT AN A, B, C, OR D>
020E C30001             JMP     CYCLE
                ;
0211            X:      DS      1       ;TEMP FOR CHARACTER
0212                    END
```

Listing 9-17a. Single Character Processing using COMPARE

In comparing each letter, the macro NEQ starts with the first argument corresponding to the character typed at the console (X); the second argument corresponds to the letter to match. The % operator in each case produces the numeric value of the character. This is necessary because the TEST? macro expects either a number or a label value in the second argument position. The program processes characters until a D is typed when it returns to the Console Command Processor. The intention here is to show the use of Boolean tests used by the control structure macros that follow.

Listing 9-17b shows a partial expansion of the macros given in the previous example. The first message expansion is shown, along with the READ and NEQ macros. The listing has been abstracted, however, and does not show the macro library statements or the remainder of the program following the NOTA label.

```
              ;       . . .
              ;
               CYCLE: WRITE   <TYPE A CHARACTER FROM A TO D >
0100+C32301           JMP     ??0002
0103+0D0A      ??0001:        DB      CR,LF
0105+5459504520       DB      'TYPE A CHARACTER FROM A TO D '
0122+24               DB      '$'
0123+0E09      ??0002:        MVI     C,MSGOUT
0125+110301           LXI     D,??0001
0128+CD0500           CALL    BDOS
                      READ    X
012B+0E01             MVI     C,CONIN ;CONSOLE INPUT FUNCTION
012D+CD0500           CALL    BDOS    ;CHARACTER IS IN A
0130+321102           STA     X
              ;       TEST FOR LOWER CASE ALPHABETIC
                      LSS     X,61H,NOTRAN
0133+3A1102           LDA     X
0136+D661             SUI     61H
0138+DA4301           JC      NOTRAN
              ;       ARRIVE HERE IF X IS GREATER OR EQUAL TO
              ;       A LOWER CASE A (=61H), TRANSLATE
013B 3A1102           LDA     X
013E E65F             ANI     5FH      ;CLEAR LOWER CASE BIT
0140 321102           STA     X        ;STORE BACK TO X
               NOTRAN:
              ;       NOW CHECK CASES
              ;
                      NEQ     X,%'A',NOTA
0143+3A1102           LDA     X
0146+D641             SUI     65
0148+C26A01           JNZ     NOTA
                      WRITE   <YOU TYPED AN A>
014B+C35F01           JMP     ??0004
014E+0D0A      ??0003:        DB      CR,LF
0150+594F552054       DB      'YOU TYPED AN A'
015E+24               DB      '$'
015F+0E09      ??0004:        MVI     C,MSGOUT
0161+114E01           LXI     D,??0003
0164+CD0500           CALL    BDOS
0167 C30001           JMP     CYCLE
              ;
              NOTA:   NEQ     X,%'B',NOTB
              ;       . . .
```

Listing 9-17b. Partial Trace of Listing 9-17a with Macro Generation

The macro library shown in Listing 9-18, called NCOMPARE, expands upon the basic relational macros by allowing a false branch option. Each macro accepts four arguments: the X and Y operands, as before, a true label (TL), and a false label (FL). It is assumed that either the TL or FL is supplied in any invocation of a relational operator, but not both. If the TL is supplied, then the branch is taken if the relational operator produces a true result. Conversely, if the TL label is absent but the FL label is supplied, then the branch to FL is taken if the relational operation produces a false result. Thus, NCOMPARE expands upon the COMPARE library by allowing all of the relational operation and their negations. Using the NCOMPARE library, for example, the macro invocation

```
LSS     X,20, ,FALSELAB
```

branches to the label FALSELAB if X is not less than the value 20. The negation operations are accomplished within the NCOMPARE library by first testing for a null TL operand and, if empty, the relational operation is reversed by invoking the appropriate negated macro. For example, the LSS macro in Listing 9-18 invokes the GEQ macro, which is equivalent to 'not LSS' when the TL argument is empty and supplies the FL argument to LSS as the TL label to GEQ. These negated relational forms are used within the control structures described below.

```
;       macro library for 8-bit comparison operation
;
test?   macro   x,y
;;      utility macro to generate condition codes
        if      not nul x       ;;then load x
        lda     x       ;;x assumed to be in memory
        endif
        irpc    ?y,y    ;;y may be constant operand
tdig?   set     '&?Y'-'0'       ;;first char digit?
        exitm           ;;stop irpc after first char
        endm
        if      tdig? <= 9      ;;y numeric?
        sui     y       ;;yes, so sub immediate
        else
        lxi     h,y     ;;y not numeric
        sub     m       ;;so sub from memory
        endm
;
```

Listing 9-18. Expanded NCOMPARE Comparison Operators

```
lss     macro   x,y,tl,fl
;;      x lss than y test,
;;      if tl is present, assume true test
;;      if tl is absent, then invert test
        if      nul tl
        geq     x,y,fl
        else
        test?   x,y     ;;set condition codes
        jc      tl
        endm
;
leq     macro   x,y,tl,fl
;;      x less than or equal to y test
        if      nul tl
        geq     x,y,fl
        else
        lss     x,y,tl
        jz      tl
        endm
eql     macro   x,y,tl,fl
;;      x equal to y test
        if      nul tl
        neq     x,y,fl
        else
        test?   x,y
        jz      tl
        endm
;
neq     macro   x,y,tl,fl
;;      x not equal to y test
        if      nul tl
        eql     x,y,fl
        else
        test?   x,y
        jnz     tl
        endm
;
geq     macro   x,y,tl,fl
;;      x greater than or equal to y test
        if      nul tl
        lss     x,y,fl
        else
        test?   x,y
        jnc     tl
        endm
```

Listing 9-18. (continued)

```
;
gtr     macro   x,y,tl,fl
;;      x greater than y test
        if      nul tl
        leq     x,y,fl
        else
        local   gfl       ;;false label
        test?   x,y
        jc      gfl
        dcr     a
        jnc     tl
gfl:    endm
```

Listing 9-18. (continued)

Listing 9-19a is an example of the use of the NCOMPARE library within a program. This program is similar to the previous example, but instead checks to ensure that alphabetic translation occurs only within the proper range of lower-case letters. Following the label CYCLE, the character read from the console is compared with a lower-case a, using the % operation to produce equivalent decimal value 97. Because the negated form of GEQ is used here, the label NOTRAN receives control if X is not greater than or equal to %'a'. If X is greater than or equal to %a, program flow continues to the next test in sequence where X is compared with a lower-case z (%'z'= decimal 122). In this case, the normal form of GTR is used. Control transfers to NOTRAN if X is greater than %'z', which is above the range of lower-case alphabetics. If X is between %'a' and %'z', the character is changed to upper-case, as before, by removing the lower-case bit and replacing X in memory. Note that the indentation levels between the GEQ and GTR operations are included for readability of the program.

Listing 9-19b shows the GEQ-GTR section of the program of Listing 9-19a with full macro trace enabled. (See Section 10.) The trace in this listing shows the transition from GEQ to the LSS operator, substituting the FL label in place of the TL label. Again, the macro library statements are not shown, and the listing following the NOTRAN label is not present.

```
0100                    ORG     100H
                        MACLIB  SIMPIO  ;SIMPLE IO LIBRARY
                        MACLIB  NCOMPARE;COMPARISON OPERATORS
                ;
0100            CYCLE:  WRITE   <TYPE A CHARACTER FROM A TO D >
012B                    READ    X
                ;       TEST FOR LOWER CASE ALPHABETIC
0133                    GEQ     X,%'a',,NOTRAN  ;BRANCH ON FALSE
                ;       X IS GREATER OR EQUAL TO LOWER CASE A
013B                            GTR     X,%'z',NOTRAN
0147 3A1D02                     LDA     X
014A E65F                       ANI     5FH     ;UPPER CASE
014C 321D02                     STA     X       ;BACK TO X
                ;
                NOTRAN:
                ;       NOW CHECK CASES
                ;
014F                    NEQ     X,%'A',NOTA
0157                    WRITE   <YOU TYPED AN A>
0173 C30001             JMP     CYCLE
                ;
0176            NOTA:   NEQ     X,%'B',NOTB
017E                    WRITE   <YOU TYPED A B>
0199 C30001             JMP     CYCLE
                ;
019C            NOTB:   NEQ     X,%'C',NOTC
01A4                    WRITE   <YOU TYPED A C>
01BF C30001             JMP     CYCLE
                ;
01C2            NOTC:   NEQ     X,%'D',ERROR
01CA                    WRITE   <YOU TYPED A D>
01E5                    WRITE   <BYE^!>
01F7 C9                 RET
                ;
01F8            ERROR:  WRITE   <NOT AN A, B, C, OR D>
021A C30001             JMP     CYCLE
                ;
021D            X:      DS      1       ;TEMP FOR CHARACTER
021E                    END
```

Listing 9-19a. Sample Program using NCOMPARE Library

```
                ;       TEST FOR LOWER CASE ALPHABETIC
                        GEQ     X,%'a',,NOTRAN  ;BRANCH ON FALSE
    +                   IF      NUL
    +                   LSS     X,97,NOTRAN
    +                   IF      NUL NOTRAN
    +                   GEQ     X,97,
    +                   ELSE
    +                   TEST?   X,97
    +                   IF      NOT NUL X
0133+3A1D02             LDA     X
    +                   ENDIF
    +                   IRPC    ?Y,97
    +           TDIG?   SET     '&?Y'-'0'
    +                   EXITM
    +                   ENDM
0009+#          TDIG?   SET     '9'-'0'
    +                   EXITM
    +                   IF      TDIG? <= 9
0136+D661               SUI     97
    +                   ELSE
    +                   LXI     H,97
    +                   SUB     M
    +                   ENDM
0138+DA4F01             JC      NOTRAN
    +                   ENDM
    +                   ELSE
    +                   TEST?   X,97
    +                   JNC
    +                   ENDM
                ;       X IS GREATER OR EQUAL TO LOWER CASE A
                                GTR     X,%'z',NOTRAN
    +                   IF      NUL NOTRAN
    +                   LEQ     X,122,
    +                   ELSE
    +                   LOCAL   GFL
    +                   TEST?   X,122
    +                   IF      NOT NUL X
013B+3A1D02             LDA     X
    +                   ENDIF
    +                   IRPC    ?Y,122
    +           TDIG?   SET     '&?Y'-'0'
    +                   EXITM
    +                   ENDM
0001+#          TDIG?   SET     '1'-'0'
    +                   EXITM
    +                   IF      TDIG? <= 9
```

Listing 9-19b. Segment of Listing 9-19a with +M Option

```
013E+D67A               SUI     122
    +                   ELSE
    +                   LXI     H,122
    +                   SUB     M
    +                   ENDM
0140+DA4701             JC      ??0003
0143+3D                 DCR     A
0144+D24F01             JNC     NOTRAN
    +           ??0003:         ENDM
0147 3A1D02                     LDA     X
014A E65F                       ANI     5FH     ;UPPER CASE
014C 321D02                     STA     X       ;BACK TO X
                ;
                NOTRAN:
```

Listing 9-19b. (continued)

Given the SIMPIO and NCOMPARE libraries, it is now possible to define the first complete control structure, called the WHEN-ENDW group. The form of the group is

```
WHEN  condition
statement-1
statement-2
. . .
statement-n
ENDW
```

where condition is a relational expression taking one of the forms

id,rel,id   id,rel,number   ,rel,id   ,rel,number

and id is an identifier; rel is a relational operator (LSS, LEQ, EQL, NEQ, GEQ, GTR), and number is a literal numeric value. Similar in form to the arguments of the individual relational operators of the COMPARE library, the last two forms shown above assume the first argument is present in the 8080 accumulator. The condition following the WHEN is evaluated as a relational expression, according to the rules stated with the COMPARE library. If the condition produces a true result, then statement-1 through statement-n are executed. Otherwise, control transfers to the statement following the ENDW. Nested WHEN-ENDW groups are allowed when they take the form:

```
WHEN . . .
. . .
        WHEN . . .
        . . .
                WHEN . . .
                . . .
                ENDW
        . . .
        ENDW
. . .
ENDW
```

to arbitrary levels, where the ellipses represent interspersed statements. Because of the simplified implementation, nested parallel WHEN-ENDW groups are disallowed when they take the form:

```
WHEN . . .
. . .
        WHEN . . .
        . . .
        ENDW
        . . .
        WHEN . . .
        . . .
        ENDW
. . .
ENDW
```

The implementation of the WHEN-ENDW group is based upon macros that count WHEN-ENDW groups and generate branches and labels at the proper levels in the structure.

Listing 9-20 shows the WHEN macro library, consisting of four macros:

GENWTST	(generate WHEN test)
GENLAB	(generate label)
WHEN	(beginning of WHEN group)
ENDW	(end of WHEN group)

These macros, in turn, use the macros in the NCOMPARE library shown previously and thus are assumed to exist in the user's program as a result of a MACLIB NCOMPARE statement. Label generation is based on the WCNT (WHEN count) and WLEV (WHEN level) counters. WCNT is incremented each time a WHEN is encountered, and WLEV keeps track of the number of WHENs that have occurred without corresponding ENDWs.

Upon encountering the first WHEN, the WCNT and WLEV counters are set to zero, and the WHEN macro is redefined to generate the first WHEN test by invoking GENWTST, using the relation R, operands X and Y, and WHEN counter WCNT. The value of WCNT is passed to GENWTST rather than the characters WCNT themselves. Thus, at the first invocation of GENWTST, the dummy argument NUM has the value 0. The first argument to GENWTST, called TST, corresponds to a relational operation (LSS through GTR) and thus is invoked automatically within the body of GENWTST, using the negated form of the relational because the TL argument is empty.

Again referring to the body of the GENWTST macro in Listing 9-20, the last argument, corresponding to the false label of the relational operation, is the constructed label ENDW&num, where num has the value 0 initially, and successively larger values on later invocations. Each time GENWTST is invoked, it generates a relational test and a branch on false to a generated label. It is the responsibility of the ENDW macro to produce the appropriate balanced label when encountered in the program.

In the body of the WHEN macro in Listing 9-20, the WLEV level counter is set to the current WCNT, and the WCNT is incremented in preparation for the next WHEN statement. Similar to nearly all macros that redefine themselves, the outer macro definition of WHEN invokes the newly created WHEN macro before exit.

Upon encountering the ENDW statement in the source program, the ENDW macro first invokes GENLAB to generate the appropriate ENDW label. The first argument to GENLAB is the label prefix ENDW; the second argument is the evaluated parameter %WLEV corresponding to the current ENDW label. If only one WHEN statement is encountered, for example, the value of WLEV is zero, and thus GENLAB produces the label ENDW0, which is the destination of the earlier branch generated by an invocation of GENWTST. Following the invocation of GENLAB, WLEV is decremented to account for the fact that one more destination label has been resolved.

```
;       macro library for "when" construct
;
;       label generators
genwtst macro tst,x,y,num
;;      generate a "when" test (negated form),
;;      invoke macro "tst" with parameters
;;      x,y with jump to endw & num
        tst x,y,,endw&num
        endm
;
genlab  macro   lab,num
;;      produce the label "lab" & "num"
lab&num:
        endm
;
;       "when" macros for start and end
;
when    macro   xv,rel,yv
;;      initialize counters first time
wcnt    set     0       ;;number of whens
when    macro   x,r,y
        genwtst r,x,y,%wcnt
wlev    set     wcnt    ;;next endw to generate
wcnt    set     wcnt+1  ;;number of ;"when"s
        endm
        when    xv,rel,yv
        endm
;
         endw    macro
;;      generate the ending code for a "when"
        genlab  endw,%wlev
wlev    set     wlev-1  ;;count current level down
;;      wlev must not go below 0 (not checked)
        endm
```

Listing 9-20. Macro Library for the WHEN Statement

As an example of the use of WHEN-ENDW, Listing 9-21a shows a sample program that resembles the previous character scanning function, but uses the WHEN group in place of simple tests and branches. As before, a single character is read from the console and first tested for possible case conversion. The statement WHEN X,GEQ,61H causes the three statements that follow to execute only when X is greater than or equal to 61H (lower-case a). Further, the four WHEN groups that follow test for the specific characters A, B, C, or D. If an A is typed, the corresponding WHEN group executes, and control transfers back to the CYCLE label where another character is read from the console. If the letter D is typed, the program responds with two messages and returns to the console command processor.

Listing 9-21b shows the same program with full macro trace enabled. This portion of the program shows macro processing for the first WHEN-ENDW group only, although the remaining groups are processed in a similar fashion. It is a worthwhile exercise to determine that the nesting rules for WHEN groups are properly stated, and that the restriction on nested parallel groups is necessary.

```
0100                    ORG     100H
                        MACLIB  SIMPIO  ;SIMPLE IO LIBRARY
                        MACLIB  NCOMPARE;EXPANDED COMPARE OPS
                        MACLIB  WHEN    ;WHEN CONSTRUCT
                ;
0100            CYCLE:  WRITE   <TYPE A CHARACTER FROM A TO D >
012B                    READ    X
                ;       TEST FOR LOWER CASE ALPHABETIC
0133                    WHEN    X,GEQ,61H
013B 3A1102             LDA     X
013E E65F               ANI     5FH     ;CLEAR LOWER CASE BIT
0140 321102             STA     X       ;STORE BACK TO X
0143                    ENDW
                ;       NOW CHECK CASES
                ;
0143                    WHEN    X,EQL,%'A'
014B                    WRITE   <YOU TYPED AN A>
0167 C30001             JMP     CYCLE
016A                    ENDW
                ;
016A                    WHEN    X,EQL,%'B'
0172                    WRITE   <YOU TYPED A B>
018D C30001             JMP     CYCLE
0190                    ENDW
                ;
```

Listing 9-21a. Sample WHEN Program with −M in Effect

```
0190                         WHEN    X,EQL,%'C'
0198                         WRITE   <YOU TYPED A C>
01B3 C30001                  JMP     CYCLE
01B6                         ENDW
                   ;
01B6                         WHEN    X,EQL,% 'D'
01BE                         WRITE   <YOU TYPED A D>
01D9                         WRITE   <BYE^!>
01EB C9                      RET
01EC                         ENDW
                   ;
01EC                         WRITE   <NOT AN A, B, C, OR D>
020E C30001                  JMP     CYCLE
                   ;
0211               X:        DS      1          ;TEMP FOR CHARACTER
```

Listing 9-21a. (continued)

```
                    ;        . . .
                    ;
                    ;        TEST FOR LOWER CASE ALPHABETIC
                             WHEN    X,GEQ,61H
0000+#             WCNT      SET     0
    +              WHEN      MACRO   X,R,Y
    +                        GENWTST R,X,Y,%WCNT
    +              WLEV      SET     WCNT
    +              WCNT      SET     WCNT+1
    +                        ENDM
    +                        WHEN    X,GEQ,61H
    +                        GENWTST GEQ,X,61H,%WCNT
    +                        GEQ     X,61H,,ENDW0
    +                        IF      NUL
    +                        LSS     X,61H,ENDW0
    +                        IF      NUL ENDW0
    +                        GEQ     X,61H,
    +                        ELSE
    +                        TEST?   X,61H
    +                        IF      NOT NUL X
0133+3A1102                  LDA     X
    +                        ENDIF
    +                        IRPC    ?Y,61H
    +             TDIG?      SET     '&?Y'-'0'
    +                        EXITM
    +                        ENDM
0006+#             TDIG?     SET     '6'-'0'
    +                        EXITM
    +                        IF      TDIG? <= 9
```

Listing 9-21b. Partial Listing of Listing 9-21a with +M Option

```
0136+D661               SUI     61H
    +                   ELSE
    +                   LXI     H,61H
    +                   SUB     M
    +                   ENDM
0138+DA4301             JC      ENDW0
    +                   ENDM
    +                   ELSE
    +                   TEST?   X,61H
    +                   JNC
    +                   ENDM
    +                   ENDM
0000+#          WLEV    SET     WCNT
0001+#          WCNT    SET     WCNT+1
    +                   ENDM
    +                   ENDM
013B 3A1102             LDA     X
013E E65F               ANI     5FH     ;CLEAR LOWER CASE BIT
0140 321102             STA     X       ;STORE BACK TO X
                        ENDW
                ;       . . .
```

Listing 9-21b. (continued)

A second control structure, called the DOWHILE-ENDDO group, takes the general form:

```
DOWHILE   condition
statement-1
statement-2
. . .
statement-n
ENDDO
```

where the condition and nesting rules are identical to the WHEN-ENDW group. The DOWHILE group is similar in concept to the WHEN group, except that statements 1 through n execute repetitively as long as the condition remains true. That is, the condition is evaluated when the DOWHILE is encountered in normal program flow. If the condition produces a false value, then control transfers to the statement following the ENDDO. Otherwise, the statements within the group execute until the ENDDO is reached. Upon encountering the ENDDO, control transfers back to the DOWHILE, and the condition is evaluated again. Iteration continues through the group until the condition produces a false value.

The macro library for the DOWHILE group is shown in Listing 9-22. The DOWHILE statement invokes the relational operator macros to produce the proper sequence of tests and branches. Upon encountering the ENDDO, the proper label and jump sequence is again generated. The only essential difference in the DOWHILE and WHEN groups is that the location of the DOWHILE test must be labeled, and a JMP instruction must be generated to this label at the end of each group.

```
;       macro library for "dowhile" construct
;
gendtst macro   tst,x,y,num
;;      generate a "dowhile" test
        tst     x,y,,endd&num
        endm
;
gendlab macro   lab,num
;;      produce the label lab & num
;;      for dowhile entry or exit
lab&num:
        endm
;
gendjmp macro   num
;;      generate jump to dowhile test
        jmp     dtest&num
        endm
;
dowhile macro   xv,rel,yv
;;      initialize counter
docnt   set 0          ;number of dowhiles
;;
dowhile macro   x,r,y
;;      generate the dowhile entry
        gendlab dtest,%docnt
;;      generate the conditional test
        gendtst r,x,y,%docnt
dolev   set     docnt   ;;next endd to generate
docnt   set     docnt+1
        endm
        dowhile xv,rel,yv
        endm
;
enddo   macro
;;      generate the jump to the test
        gendjmp %dolev
;;      generate the end of a dowhile
        gendlab endd,%dolev
dolev   set     dolev-1
        endm
```

Listing 9-22. Macro Library for the DOWHILE Statement

In Listing 9-22, GENDTST (generate DOWHILE test), GENDLAB (generate DOWHILE label), and GENDJMP (generate DOWHILE jump) are all label generators used in the macros that follow. Similar to the WHEN macro, DOWHILE uses the counters DOCNT and DOLEV to keep track of the number of DOWHILE groups encountered along with the current DOWHILE level, corresponding to the number of unmatched DOWHILEs. The DOWHILE macro first generates the entry label DTESTn, where n is the DOWHILE count. The conditional test is then generated, similar to the WHEN macro, with a branch on false condition to the ENDDn label that is eventually generated by the ENDDO macro. Finally, the DOWHILE macro increments the DOCNT counter in preparation for the next group.

The ENDDO macro in Listing 9-22 first generates the JMP instruction back to the DOWHILE test, using the GENDLAB utility macro, and then produces the ENDDn label that becomes the target of the jump on false condition. The form of the expanded macros for one nested level thus becomes:

```
DTEST0:
conditional jump to ENDD0
        DTEST1:
        conditional jump to ENDD1
        . . .
        JMP DTEST1
. . .
ENDD1
JMP DTEST0
```

Listing 9-23a shows an example of a program that uses the DOWHILE group. Although this program differs slightly from the previous examples, the principal function is the same: a STOP character is first read from the console, followed by a group of statements that repetitively execute in search of the STOP character. Two DOWHILE groups occur within the program. The first group checks each character typed (X) to see if it matches the STOP character. If not (DOWHILE X,NEQ,STOP), the statements up through the matching ENDDO are processed. If the value of X is the character A, then the message YOU TYPED AN A is sent to the console. Otherwise, the message NOT AN A is typed, followed by a check to see if the STOP character was typed. If so, the messages STOP CHARACTER and BYE! appear at the console. Control continues through the ENDWs to the ENDDO and back to the DOWHILE header. The DOWHILE X,NEQ,STOP produces a false condition, and control transfers to the XRA A instruction following the ENDDO.

```
0100                    ORG     100H
                        MACLIB  SIMPIO  ;SIMPLE IO LIBRARY
                        MACLIB  NCOMPARE;EXPANDED COMPARE OPS
                        MACLIB  WHEN    ;WHEN CONSTRUCT
                        MACLIB  DOWHILE ;DOWHILE STATEMENT
                ;
0100                    WRITE   <TYPE THE STOP CHARACTER: >
0127                    READ    STOP
                ;       X = 0 FOR THE FIRST LOOP

012F                    DOWHILE X,NEQ,STOP      ;LOOK FOR STOP CHARACTER
0139                    WRITE   <TYPE A CHARACTER: >
0159                    READ    X
                ;
0161                    WHEN    X,EQL,%'A'
0169                    WRITE   <YOU TYPED AN A>
0185                    ENDW
                ;
0185                    WHEN    X,NEQ,%'A'
018D                    WRITE   <NOT AN A>
01A3                            WHEN    X,EQL,STOP
01AD                            WRITE   <STOP CHARACTER>
01C9                            WRITE   <BYE^!>
01DB                            ENDW
01DB                    ENDW
01DB                    ENDDO
                ;
                ;       CLEAR THE SCREEN (23 CRLF'S)
01DE AF                 XRA     A
01DF 320002             STA     X       ;X=0
01E2                    DOWHILE X,LSS,23
01EA                    WRITE   <>
01F8 210002             LXI     H,X
01FB 34                 INR     M       ;X=X+1
01FC                    ENDDO
01FF C9                 RET
                ;
0200 00         X:      DB      0       ;EXECUTES "DOWHILE" FIRST TIME
0201            STOP:   DS      1       ;STOP CHARACTER
```

Listing 9-23a. An Example Using the DOWHILE Statement

```
              ;       CLEAR THE SCREEN (23 CRLF'S)
01DE AF               XRA     A
01DF 320002           STA     X       ;X=0
                      DOWHILE X,LSS,23
01E2+3A0002           LDA     X
01E5+D617             SUI     23
01E7+D2FF01           JNC     ENDD1
                      WRITE   <>
01EA+C3F001           JMP     ??0014
01ED+0D0A     ??0013:         DB      CR,LF
01EF+24               DB      '$'
01F0+0E09     ??0014:         MVI     C,MSGOUT
01F2+11ED01           LXI     D,??0013
01F5+CD0500           CALL    BDOS
01F8 210002           LXI     H,X
01FB 34               INR     M       ;X=X+1
                      ENDDO
01FC+C3E201           JMP     DTEST1
01FF C9               RET
```

Listing 9-23b. Partial Listing of Listing 9-23a with Macro Generation

In Listing 9-23a, the second DOWHILE-ENDDO group clears the normal CRT screen size of 23 lines. This is accomplished by first setting X to the value zero, followed by a DOWHILE group that checks the condition X,LSS,23 which iterates until X reaches the value 23. The WRITE statement within the DOWHILE group produces only the carriage return line-feed on each iteration because the character sequence within the brackets is empty. Following the WRITE statement, X is incremented by one, acting as a line counter. When X reaches 23, the RET statement following the matching ENDDO receives control, and the program terminates by returning to the console processor. Note that the DB statement for X provides the initial value zero, so that the first DOWHILE executes at least one time.

Listing 9-23b shows a portion of the program of Listing 9-23a, with partial macro trace enabled. This trace does not show the generated labels ENDD1 and DTEST1 because no machine code was generated on those lines. The +M assembly parameter would show the labels, however. The locations of these labels can be derived from the hex listing to the left; the JNC ENDD1 produces the destination address 01FF corresponding to the RET statement, and the JMP DTEST1 produces the address 01E2 corresponding to the LDA X instruction at the beginning of the DOWHILE group.

The last control structure presented in this section is the SELECT-ENDSEL group, which corresponds to the FORTRAN computed GO-TO, the ALGOL switch statement, and the PL/M case statement. The general form of the SELECT group is

```
SELECT     id
statement-set-0
SELNEXT
statement-set-1
SELNEXT
  . . .
SELNEXT
statement-set-n
ENDSEL
```

where id is a data label corresponding to an 8-bit value in memory, and statement set 0 through n denotes groups of statements separated by SELNEXT delimiters.

The action of the SELECT-ENDSEL group is as follows: the variable given in the SELECT statement is taken as a case number assumed to be in the range 0 through n. If the value is 0, statement-set-0 is executed and, upon completion of the group, control transfers to the statement following the ENDSEL. If the variable has the value 1, then statement-set-1 executes. Similarly, if the variable produces a value i between 0 and n, then statement-set-i receives control. There can be up to 255 groups of statements within each SELECT-ENDSEL group, and any number of distinct SELECT-ENDSEL groups. Nested SELECT-ENDSEL groups are not allowed. That is, a SELECT-ENDSEL group cannot occur within a statement-set that is enclosed in another SELECT-ENDSEL group. As a convenience, the variable following the SELECT can be omitted, in which case the current 8080 accumulator content selects the proper case.

Listings 9-24a and 9-24b show the SELECT macro library that implements the SELECT-ENDSEL group. The general strategy is to count the cases as they occur, starting with the SELECT, delimited by NEXTSEL, and terminated by ENDSEL. As the cases occur, a case label is generated that takes the form CASEn@m where n counts the SELECT-ENDSEL groups, and m is the case number within group n. A jump instruction is generated at the end of each case to the label ENDSn that marks the end of the SELECT group number n. Upon encountering the end of the group, a select-vector is generated that contains the address of each case within the group, headed by the label SELVn, where n is again the group number. Machine code is thus generated at the SELECT entry, which indexes into the select vector, based upon the SELECT variable, to obtain the proper case address. The first statement within the case receives control based upon the value obtained from this vector.

The general form of the machine code generated for the first SELECT group within a program (group n = 0) is:

```
        LDA     id
        LXI     SELV0
        (index HL by id, and
        load the address to HL)
        PCHL
CASE0@0:
        statement-set-0
        JMP     ENDS0
CASE0@1:
        statement-set-1
        JMP     ENDS0
        . . .
CASE@n:
        statement-set-n
        JMP     ENDS0
SELV0:
        DW      CASE0@0
        DW      CASE0@1
        . . .
        DW      CASE0@n
ENDS0:
```

Listing 9-24a contains the label generators GENSLXI (generate SELECT LXI), GENCASE (generate case labels), GENELT (generate select vector element), and GENSLAB (generate SELECT label). Listing 9-24b contains the macro definitions for SELNEXT (select next case), SELECT, and ENDSEL.

In Listing 9-24b, the SELECT macro begins by zeroing CCNT which counts SELECT-ENDSEL groups and then redefines itself, similar to the WHEN and DOWHILE macros. The redefined SELECT macro then generates the select vector indexing operation by loading the indexing variable, if necessary, and then fetches the specific case address. No machine code is generated to check that the indexing variable is within the proper range. The PCHL at the end of this code sequence performs the branch to the selected case.

At the end of the redefined select macro, SELNEXT is invoked automatically, to delimit the first case in the SELECT group (otherwise SELECT would have to be followed immediately by SELNEXT in the user program to generate the proper labels). SELECT also zeros the ECNT variable, which counts the cases until ENDSEL is encountered.

```
;       macro library for "select" construct
;
;       label generators
genslxi macro   num
;;      load hl with address of case list
        lxi     h,selv&num
        endm
;
gencase macro   num,elt
;;      generate jmp to end of cases
        if      elt gt 0
        jmp     ends&num        ;;past addr list
        endif
;;      generate label for this case
case&num&@&elt:
        endm
;
genelt  macro   num,elt
;;      generate one element of case list
        dw      case&num&@&elt
        endm
;
genslab macro   num,elts
;;      generate case list
selv&num:
ecnt    set     0       ;;count elements
        rept    elts    ;;generate dw's
        genelt  num,%ecnt
ecnt    set     ecnt+1
        endm            ;;end of dw's
;;      generate end of case list label
ends&num:
        endm
```

Listing 9-24a. Macro Library for SELECT Statement

```
selnext macro
;;      generate the next case
        gencase %ccnt,%ecnt
;;      increment the case element count
ecnt    set     ecnt+1
        endm
;
select  macro   var
;;      generate case selection code
ccnt    set     0       ;;count "selects"
select  macro   v       ;;redefinition of select
;;      select on v or accumulator contents
        if      not nul v
        lda     v       ;;load select variable
        endif
        genslxi %ccnt   ;;generate the lxi h,selv#
        mov     e,a     ;;create double precision
        mvi     d,0     ;;v in d,e pair
        dad     d       ;;single prec index
        dad     d       ;;double prec index
        mov     e,m     ;;low order branch addr
        inx     h       ;;to high order byte
        mov     d,m     ;;high order branch index
        xchg            ;;ready branch address in hl
        pchl            ;;gone to the proper case
ecnt    set     0       ;;element counter reset
        endm
;;      invoke redefined select the first time
        select  var
        selnext         ;;automatically select case 0
        endm
;
endsel  macro
;;      end of select, generate case list
        gencase %ccnt,%ecnt     ;;last case
        genslab %ccnt,%ecnt     ;;case list
;;      increment "select" count
ccnt    set     ccnt+1
        endm
```

Listing 9-24b. Library for SELECT Statement

You use SELNEXT, shown at the top of Listing 9-24b, to delimit cases. The GENCASE utility macro is invoked which, in turn, generates a JMP instruction for the previous group, if this is not group zero, and then produces the appropriate case entry label. SELNEXT also increments the select element counter ECNT to account for yet another case.

Upon encountering the ENDSEL, the last macro in Listing 9-24b, GENCASE is again called to generate the JMP instruction for the last case. GENSLAB then produces the select vector by first generating the SELVn label, followed by a list of ECNT DW statements that have the case label addresses as operands.

Listing 9-25a gives an example of a simple program that uses two SELECT groups. The first SELECT group executes one of five different MVI instructions based on the value of X. The second SELECT group assumes that the 8080 accumulator contains the selector index and executes one of three different MVI instructions. The program of Listing 9-25a illustrates generated control structures, and does not produce any useful values as output. The sorted Symbol Table shown at the end of the listing gives the generated label addresses for the individual cases.

Listing 9-25b shows a segment of the previous program with generated macro lines. Note the case selection code following SELECT X at the end of the listing.

Listing 9-25c gives a more complete trace of the SELECT-ENDSEL group, showing the actions of the macros as they expand for the second SELECT-ENDSEL group of Listing 9-25a. The listing has been edited to remove the case selection code, which is listed in Listing 9-25b, and the code generated for case number 2. Cross-reference Listing 9-25c with the SELECT macro library given in Listings 9-24a and 9-24b if you are confused about the actions of these macros.

```
                       MACLIB  SELECT
 0000                  SELECT  X
 0010 3E00             MVI     A,0
 0012                  SELNEXT
 0015 3E01             MVI     A,1
 0017                  SELNEXT
 001A 3E02             MVI     A,2
 001C                  SELNEXT
 001F 3E03             MVI     A,3
 0021                  SELNEXT
 0024 3E04             MVI     A,4
 0026                  ENDSEL
               ;
 0033                  SELECT
 0040 0600             MVI     B,0
 0042                  SELNEXT
 0045 0601             MVI     B,1
 0047                  SELNEXT
 004A 0602             MVI     B,2
 004C                  ENDSEL
               ;
 0055          X:      DS      1


0010 CASE0@0  0015 CASE0@1   001A CASE0@2   001F CASE0@3  0024 CASE0 4
0029 CASE0@5  0040 CASE1@0   0045 CASE1@1   004A CASE1@2  004F CASE1 3
0033 ENDS0    0055 ENDS1     0029 SELV0     004F SELV1    0055 X
```

Listing 9-25a. Sample Program Using SELECT with −M +S Options

```
                        MACLIB SELECT
                        SELECT X
0000+3A5500             LDA     X
0003+212900             LXI     H,SELV0
0006+5F                 MOV     E,A
0007+1600               MVI     D,0
0009+19                 DAD     D
000A+19                 DAD     D
000B+5E                 MOV     E,M
000C+23                 INX     H
000D+56                 MOV     D,M
000E+EB                 XCHG
000F+E9                 PCHL
0010 3E00               MVI     A,0
                        SELNEXT
0012+C33300             JMP     ENDS0
0015 3E01               MVI     A,1
                        SELNEXT
0017+C33300             JMP     ENDS0
001A 3E02               MVI     A,2
                        SELNEXT
001C+C33300             JMP     ENDS0
001F 3E03               MVI     A,3
                        SELNEXT
0021+C33300             JMP     ENDS0
0024 3E04               MVI     A,4
                        ENDSEL
0026+C33300             JMP     ENDS0
0029+1000               DW      CASE0@0
002B+1500               DW      CASE0@1
002D+1A00               DW      CASE0@2
002F+1F00               DW      CASE0@3
0031+2400               DW      CASE0@4
```

Listing 9-25b. Segment of Listing 9-25a with Mnemonics

```
                        SELECT
    +                   IF      NOT NUL
    +                   LDA
    +                   ENDIF
    +                   GENSLXI %CCNT
0033+214F00             LXI     H,SELV1
    +                   ENDM
                        . . .
                  (indexing code similar to Fig 50b)
                        . . .
0000+#            ECNT   SET     0
    +                   GENCASE %CCNT,%ECNT
    +                   IF      0 GT 0
    +                   JMP     ENDS1
    +                   ENDIF
    +             CASE1@0:
    +                   ENDM
0001+#            ECNT  SET     ECNT+1
    +                   ENDM
    +                   ENDM
0040 0600               MVI     B,0
                        SELNEXT
    +                   GENCASE %CCNT,%ECNT
    +                   IF      1 GT 0
0042+C35500             JMP     ENDS1
    +                   ENDIF
    +             CASE1@1:
    +                   ENDM
0002+#            ECNT  SET     ECNT+1
    +                   ENDM
                        . . .
                  (remaining cases are similar)
                        . . .
                        ENDSEL
    +                   GENSLAB %CCNT,%ECNT
    +             SELV1:
0000+#            ECNT  SET     0
    +                   REPT    3
    +                   GENELT  1,%ECNT
    +             ECNT  SET     ECNT+1
    +                   ENDM
    +                   GENELT  1,%ECNT
004F+4000               DW      CASE1@0
    +                   ENDM
0001+#            ECNT  SET     ECNT+1
    +                   GENELT  1,%ECNT
```

Listing 9-25c. Segment of Listing 9-25a with +M Option

```
0051+4500               DW      CASE1@1
    +                   ENDM
0002+#          ECNT    SET     ECNT+1
    +                   GENELT  1,%ECNT
0053+4A00               DW      CASE1@2
    +                   ENDM
0003+#          ECNT    SET     ECNT+1
    +                   ENDM
    +           ENDS1:
    +                   ENDM
0002+#          CCNT    SET     CCNT+1
    +                   ENDM
```

Listing 9-25c. (continued)

It is now possible to show a complete program that uses the WHEN, DOWHILE, and SELECT groups. Listing 9-26 shows a program similar in function to a more complicated program that interacts with the console in executing single-character input commands. The two CP/M programs ED and DDT both take this general form. (See the CP/M documentation for details.) A single letter selects a single action that might correspond to an edit request in the ED program or a debug request in DDT. Upon completion of each command, control returns to the main loop to accept another single-letter command.

The program given in Listing 9-26 begins by loading the macro definitions for the SIMPIO, NCOMPARE, WHEN, DOWHILE, and SELECT operations. Several messages are then sent to the console device, followed by a single DOWHILE-ENDDO group that encompasses nearly the entire program. The DOWHILE group is controlled by the X,NEW,%'D' test and thus continues to loop while the X character is not the letter D. On each iteration of the DOWHILE group, a single letter is read from the console and converted to upper-case, if necessary. To ensure that the letter is in the proper range of values, two WHEN groups follow that convert illegal values to the letter E, which subsequently produces an error response.

Following the WHEN tests in Listing 9-26, the character must be in the range A through E. Before indexing into the SELECT group, this value is normalized to the absolute value 0 through 4, corresponding to each of the possible values. The SELECT statement uses the value in the accumulator to select one of the five cases, producing the appropriate response to the letters A through D, or an error response for the last case. Upon completion of the SELECT group, control returns to the DOWHILE where the last character typed is tested against the letter D. If X is not equal to the letter D, the iteration continues. Otherwise, the DOWHILE completes and control returns to the console processor.

The control structures presented in this section are representative of the forms that can be implemented. Additional facilities, such as the controlled iteration found in FORTRAN DO loops or ALGOL FOR loops can be implemented using essentially the same techniques used for the WHEN and DOWHILE. Further, subroutine parameters can also be defined with macro libraries. It is relatively easy to include control substructures for the stack machine given in the previous section, allowing machine independent programming of control structures and arithmetic operations.

```
0100                     ORG     100H    ;BEGINNING OF TPA
                         MACLIB  SIMPIO  ;SIMPLE READ/WRITE
                         MACLIB  NCOMPARE;COMPARISON OPS
                         MACLIB  WHEN    ;"WHEN" CONSTRUCT
                         MACLIB  DOWHILE ;"DOWHILE" CONSTRUCT
                         MACLIB  SELECT  ;"SELECT" CONSTRUCT
                  ;
                  ;      USING THE CCP'S STACK, READ INPUT
                  ;      CHARACTERS, UNTIL A Z IS TYPED
0100                     WRITE <SAMPLE CONTROL STRUCTURES>
0127                     WRITE <TYPED SINGLE CHARACTERS FROM>
0150                     WRITE <A TO D, I^'^'LL STOP ON D>
                  ;
0174                     DOWHILE X,NEQ,%'D'
017C                     WRITE   <TYPE A CHARACTER: >
019C                     READ X

01A4                     WHEN X,GEQ,%'A'
01AC 3ABF02E65F            LDA X! ANI 05FH! STA X ;CONV CASE
01B4                     ENDW

01B4                     WHEN X,LSS,%'A'
01BC 3E4532BF02            MVI A,'E'! STA X ;SET TO ERROR
01C1                     ENDW

01C1                     WHEN X,GTR,%'E'
01CC 3E4532BF02            MVI A,'E'! STA X ;SET TO ERROR
01D1                     ENDW

01D1 3ABF02D641          LDA X! SUI 'A' ;NORMALIZE TO 0-4
01D6                       SELECT ;BASED ON X IN ACCUM
01E3                         WRITE <YOU SELECTED CASE A>
0204                         SELNEXT
0207                         WRITE <YOU SELECTED CASE B>
0228                         SELNEXT
022B                         WRITE <YOU SELECTED CASE C>
024C                         SELNEXT
024F                         WRITE <YOU SELECTED CASE D>
0270                         WRITE <SO I''M GOING BACK^!>
0290                         SELNEXT
0293                         WRITE <BAD CHARACTER>
02AE                       ENDSEL
02BB                     ENDDO

02BE C9                  RET     ;BACK TO CCP

                  ;      DATA    AREA
02BF 00           X:     DB      0       ;X=00 INITIALLY
```

Listing 9-26. Program Using WHEN, DOWHILE, and SELECT

9.4 Operating System Interface

In a general purpose computing environment, macros often provide systematic and simplified mechanisms for programmatic access to operating system functions. Throughout this manual, the examples have shown various low-level calls to the CP/M operating system that implement functions such as single-character input, single-character output, and full message output. In each case, the macros simplify the operations by performing the low-level register setups and calls that perform the function.

This section introduces more comprehensive operating system interface macros and shows a sample macro library that allows simplified disk file operations for sequential stream input/output operations. The principal macros of this library that allow file access are listed below:

FILE	set up a named file for subsequent disk operations.
GET	read a single character from specific data source.
PUT	send a character to a specific data destination.
FINIS	terminate file access for specific group of files.
ERASE	remove a specific disk file.
DIRECT	search for a specific file on the disk.
RENAME	rename a specific disk file.

Before introducing the macro library that performs these functions, the operation of each macro is described, followed by a simple example.

The FILE operation takes the form:

```
FILE  mode,fileid,diskname,filename,filetype,buffsize,buffadr
```

where the individual parameters of the FILE macro describe a file to be accessed in the program. The parameter values for the FILE macro are:

mode	INFILE (input file) OUTFILE (output file) SETFILE (set up filename for ancillary functions)

fileid	file identifier for internal reference throughout the program.
diskname	disk drive name (A, B,. . .) containing the file being accessed, or empty if the default drive is being used.
filename	the filename (up to eight characters) of the disk file being accessed; if "1" or "2" is specified, then the first or second default filename is used, respectively.
filetype	the filetype (up to three characters) of the file being accessed; if "1" or "2" has been specified for the filename parameter and an empty filetype is given, then the filetype is taken from the selected default filename; otherwise, the filetype is set to blanks.
buffsize	the size in bytes of the buffer area used for this file; the value is rounded down to an integral multiple of the disk sector size; if the rounding produces a result that is too small, or if the parameter is empty, then only one sector is buffered.
buffaddr	the address of the buffer area to use during accesses to this file; if empty, then the buffer address is assigned automatically.

For example, the FILE statement

```
FILE     INFILE,ZOT,A,NAMES,DAT
```

sets up the file NAMES.DAT on disk drive A for subsequent access. Internal to the program, this file is referenced by the name ZOT. Further, the buffer address is assigned automatically, and the buffer size is set to one sector (usually 128 bytes). Larger buffers are useful in minimizing rotational delay on the disk due to missed sectors during the file operations. If the NAMES.DAT file does not exist, an error message is sent to the console, and the program aborts. For example, an output file can be created using the statement:

```
FILE     OUTFILE,ZAP,B,ADDRESS,DAT,1000
```

which creates the file ADDRESS.DAT on drive B for subsequent output, referenced internally by the name ZAP. In this case, the buffer size is set to 1000 bytes (rounded down to 7 * 128 = 896 bytes), and the base address of the buffer is set automatically. The sample programs show alternative FILE options.

The GET macro invocation takes the form:

```
GET   device
```

where device specifies a simple peripheral or a disk file defined by a previously executed FILE statement. The GET statement reads one byte of data into the 8080 accumulator from the specified device. The possible device names are:

KEY	console keyboard input
RDR	reader device
fileid	previously defined file identifier given in a FILE statement

The following GET invocations perform the functions shown to the right below.

GET KEY	read one keyboard character.
GET RDR	read one reader character. (See the CP/M documentation for READER entry point definition.)
GET ZOT	read one character from the file given by the internal name ZOT. (The NAMES.DAT file if the above FILE statement had been executed.)

The end-of-data can be detected in two ways: if the file contains character data, the end-of-file is detected by comparing the individual characters with the standard CP/M end-of-file mark, which is a CTRL-Z (hexadecimal 1AH). The GET function also returns with the 8080 zero flag set to true if a real end-of-file is encountered, so that pure binary files can be read to the end-of-data.

The PUT macro performs the opposite function from the GET macro. The PUT invocation takes the form:

```
PUT     device
```

where device specifies a simple output peripheral or a disk file defined previously using the FILE macro. The possible device names are

CON	console display device
PUN	system punch device
LST	system listing device
fileid	previously defined output file identifier

These PUT invocations perform the following functions:

PUT CON	write the accumulator character to the console.
PUT PUN	write the accumulator character to the punch.
PUT LST	write the accumulator character to the list device.
PUT ZAP	write the accumulator character to the file with the internal name ZAP. (The ADDRESS.DAT file in the preceding example.)

Note that the character in the accumulator is preserved during the invocation, so that it can be involved in further tests or macro invocations following the PUT statement.

The FINIS statement closes a file or set of files upon completion of file access. In the case of an output file, the internal buffers are written to disk, and the filename is permanently recorded on the disk for future access. The form of the FINIS invocation takes the form:

```
FINIS    filelist
```

where filelist is a single internal name that appeared previously in a file statement or a list of such filenames, enclosed within angle brackets and separated by commas.

Although it is not necessary to close input files with the FINIS statement, it is good practice, because the file close operation might be required on future versions of the macro library. An example of the FINIS statement is:

```
FINIS ZAP
```

write all buffers for the ZAP file, and record the file in the disk directory; in the above example, the ADDRESS.DAT file is closed.

The ERASE macro allows programmatic removal of a disk file given by the specified file identifier defined in a previous FILE statement. If the file identifier is not used in a GET or PUT statement, then the FILE statement can have the mode SETFILE. This mode requires less program space than an INFILE or OUTFILE parameter. Examples of the ERASE statement are given later in this section. In the example

```
ERASE  ZOT
```

however, the file NAMES.DAT is removed from the disk, given the previous FILE statement that defines ZOT.

The DIRECT macro searches for a specific file on the disk. Similar to the ERASE macro, the file identifier must be previously given in a FILE statement using one of the three possible file modes. The DIRECT invocation sets the 8080 zero flag to false if the file is present on the disk. In both the ERASE and DIRECT macros, the file identifiers can reference filenames and types with embedded ? characters, similar to the normal CP/M DIR command, where the question mark matches any character in the filenames being scanned. The macro invocation

```
DIRECT  ZAP
```

for example, returns with the zero flag cleared if the file ADDRESS.DAT is present, and with the zero flag set if the file is not present, given the original FILE statement involving the ZAP file identifier.

The RENAME macro takes the form:

```
RENAME      newfile,oldfile
```

where newfile and oldfile are file identifiers that have appeared in previous FILE statements. The RENAME macro changes the filename given by oldfile to the filename given to newfile. The file identifiers newfile and oldfile must appear in previously executed FILE statements, but can have a mode of SETFILE if they are not used in GET or PUT macros. If the drive names for oldfile and newfile differ, then the drive name of newfile is assumed. The sequence of macro invocations

```
FINIS      ZAP        ;CLOSE ZAP
ERASE      ZOT        ;REMOVE ZOT
RENAME     ZOT,ZAP    ;CHANGE NAMES
```

for example, first closes the ADDRESS.DAT file on drive B, then erases the NAMES.DAT file on drive A. The RENAME macro then changes the ADDRESS.DAT file to the name NAMES.DAT file on drive A.

Listing 9-27 shows the use of the FILE, GET, PUT, and FINIS macros in a working program. This program reads an input file, specified at the Console Command Processor level as the first filename, and translates each lower-case alphabetic character to upper-case. The output is sent to the file given as the second parameter at the command level. For a program assembled, loaded, and stored as CASE.COM on the disk, a typical execution would be

```
CASE LOWER.DAT UPPER.DAT
```

This causes the CASE.COM file to load and execute in the Transient Program Area. Before execution, the Console Command Processor passes LOWER.DAT as the first default filename, and UPPER.DAT as the second filename. (See the CP/M documentation for exact details.)

In Listing 9-27, the CASE program begins by initializing the stack pointer to a local stack area in preparation for subsequent subroutine calls that occur within the various macros in the SEQIO macro library. The first default file specification is then taken as the SOURCE file, as defined in the first FILE macro. The second FILE statement assigns the second default file specification as an output file with the internal name DEST. In both cases, the FILE statements open the respective files and initialize the buffer areas, consisting of 2000 bytes rounded down to a multiple of the sector size.

Note that if the UPPER.DAT file already exists, the second file statement removes the existing file and creates a new UPPER.DAT file before continuing. In either case, the appropriate error messages appear at the console if the files cannot be accessed or created in the FILE statements.

```
0100                    ORG     100H
                ;       COPY FILE 1 TO FILE 2, CONVERT
                ;       TO UPPER CASE DURING THE COPY
                ;       AND ECHO TRANSACTION TO CONSOLE
                        MACLIB  SEQIO   ;SEQUENTIAL I/O LIB
0000 =          BOOT    EQU     0000H   ;SYSTEM REBOOT
005F =          UCASE   EQU     5FH     ;UPPER CASE BITS
                ;
0100 317003             LXI     SP,STACK
                ;       DEFINE SOURCE FILE:
                ;               INFILE  = INPUT FILE
                ;               SOURCE  = INTERNAL NAME
                ;               (NUL)   = DEFAULT DISK
                ;               1       = FIRST DEFAULT NAME
                ;               (NUL)   = FIRST DEFAULT TYPE
                ;               2000    = BUFFER SIZE
0103                    FILE    INFILE,SOURCE,,1,,2000
                ;
                ;       DEFINE DESTINATION FILE:
                ;               OUTFILE = OUTPUT FILE
                ;               DEST    = INTERNAL NAME
                ;               (NUL)   = DEFAULT DISK
                ;               2       = SECOND DEFAULT NAME
                ;               (NUL)   = SECOND DEFAULT TYPE
                ;               2000    = BUFFER SIZE
01EC                    FILE    OUTFILE,DEST,,2,,2000
                ;
                ;       READ SOURCE FILE, TRANSLATE, WRITE DEST
02EA            CYCLE:  GET     SOURCE
02ED FE1A               CPI     EOF     ;END OF FILE?
02EF CA0C03             JZ      ENDCOPY ;SKIP TO END IF SO
                ;
                ;       NOT END OF FILE, CONVERT TO UPPER CASE
02F2 FE61               CPI     'a'     ;BELOW LOWER CASE "A"?
02F4 DAFE02             JC      NOCONV  ;SKIP IF SO
02F7 FE7B               CPI     'z'+1   ;BELOW LOWER CASE "Z"?
02F9 D2FE02             JNC     NOCONV  ;SKIP IF ABOVE
                ;       MASK OUT LOWER CASE ALPHA BITS
02FC E65F               ANI     UCASE
02FE            NOCONV:         PUT     CON     ;WRITE TO CONSOLE
0306                    PUT     DEST    ;AND TO DESTINATION FILE
0309 C3EA02             JMP     CYCLE   ;FOR ANOTHER CHARACTER
                ;
```

Listing 9-27. Lower- to Upper-case Conversion Program

```
                ENDCOPY:
030C                    FINIS    DEST     ;END OF OUTPUT
034D C30000             JMP      BOOT     ;BACK TO CCP
                ;
0350                    DS       32       ;16 LEVEL STACK
                STACK:
                BUFFERS:
1270 =          MEMSIZE          EQU      BUFFERS+@NXTB    ;PROGRAM SIZE
0370                    END
```

Listing 9-27. (continued)

The CASE program main loop is shown in Listing 9-27 between the CYCLE and ENDCOPY labels. Each successive character is read from the SOURCE file (in this case, LOWER.DAT) and tested to see if the character is in the range of a lower-case a to lower-case z. If in this range, the character is changed to upper-case. At the NOCONV label, the (possibly translated) character in the accumulator is sent to the console device using the PUT CON macro and then sent to the DEST file (in this case, UPPER.DAT). Looping continues back to the CYCLE label where another character is read and translated.

Because the data file is assumed to consist of a stream of ASCII characters, the end-of-file is detected when a CTRL-Z is encountered. When this character is found, control transfers to the label ENDCOPY where the DEST file is closed using the FINIS macro. An error in writing or closing the DEST file produces an error message at the console, and the program aborts immediately. Upon completion of the program, control returns to the console processor through a system reboot (JMP BOOT).

The SEQIO library macros assume that all file buffers are located at the end of the user's program, as shown in Listing 9-27. In particular, the label BUFFERS must appear as the last label in the user's program, and becomes the base of the buffers allocated automatically in the FILE statements. The actual memory requirements for the program can be determined using an EQU as shown in Listing 9-27, with a statement of the form:

```
MEMSIZE     EQU     BUFFERS+@NXTB
```

that produces the equated value 1270H at the left of the listing. In this case, the program does not use the memory area beyond 1270H.

The macro library for SEQIO is shown in Listing 9-28. This listing is the most comprehensive macro library shown in this manual, containing an instance of nearly every macro facility available in MAC. The following discussion of SEQIO outlines the general functions of each macro, but it is left to you to investigate the exact operation of the library.

The SEQIO library begins with generally useful equates and utility macros. The label FILERR at the beginning becomes the destination of transfers upon encountering a file operation error. Because this is a SET statement, it can be changed in the user's program to trap error conditions rather than rebooting. The use of FILERR is apparent throughout the macro library.

```
;       sequential file i/o library
;
filerr  set     0000h   ;reboot after error
@bdos   equ     0005h   ;bdos entry point
@tfcb   equ     005ch   ;default file control block
@tbuf   equ     0080h   ;default buffer address
;
;       bdos functions
@msg    equ     9       ;send message
@opn    equ     15      ;file open
@cls    equ     16      ;file close
@dir    equ     17      ;directory search
@del    equ     19      ;file delete
@frd    equ     20      ;file read operation
@fwr    equ     21      ;file write operation
@mak    equ     22      ;file make
@ren    equ     23      ;file rename
@dma    equ     26      ;set dma address
;
@sect   equ     128     ;sector size
eof     equ     1ah     ;end of file
cr      equ     0dh     ;carriage return
lf      equ     0ah     ;line feed
tab     equ     09h     ;horizontal tab
;
@key    equ     1       ;keyboard
@con    equ     2       ;console display
@rdr    equ     3       ;reader
@pun    equ     4       ;punch
@lst    equ     5       ;list device
;
```

Listing 9-28. Sequential File Input/Output Library

```
;       keywords for "file" macro
infile equ      1       ;input file
outfile         equ     2       ;outputfile
setfile         equ     3       ;setup name only
;
;       the following macros define simple sequential
;       file operations:
;
fillnam         macro   fc,c
;;      fill the file name/type given by fc for c characters
@cnt    set     c       ;;max length
        irpc    ?fc,fc  ;;fill each character
;;      may be end of count or nul name
        if      @cnt=0 or nul ?fc
        exitm
        endif
        db      '&?FC'  ;;fill one more
@cnt    set     @cnt-1  ;;decrement max length
        endm            ;;of irpc ?fc
;;
;;      pad remainder
        rept    @cnt    ;;@cnt is remainder
        db      ' '     ;;pad one more blank
        endm            ;;of rept
        endm
;
filldef         macro   fcb,?fl,?ln
;;      fill the file name from the default fcb
;;      for length ?ln (9 or 12)
        local   psub
        jmp     psub    ;;jump past the subroutine
@def:   ;;this subroutine fills from the tfcb (+16)
        mov     a,m     ;;get next character to a
        stax    d       ;;store to fcb area
        inx     h
        inx     d
        dcr     c       ;;count length down to 0
        jnz     @def
        ret
```

Listing 9-28. (continued)

```
;;      end of fill subroutine
psub:
filldef         macro   ?fcb,?f,?l
        lxi     h,@tfcb+?f      ;;either @tfcb or @tfcb+16
        lxi     d,?fcb
        mvi     c,?l            ;;length = 9,12
        call    @def
        endm
        filldef fcb,?fl,?ln
        endm
;
fillnxt         macro
;;      initialize buffer and device numbers
@nxtb   set     0       ;;next buffer location
@nxtd   set     @lst+1  ;;next device number
fillnxt         macro
        endm
        endm
fillfcb         macro   fid,dn,fn,ft,bs,ba
;;      fill the file control block with disk name
;;      fid is an internal name for the file,
;;      dn is the drive name (a,b..), or blank
;;      fn is the file name, or blank
;;      ft is the file type
;;      bs is the buffer size
;;      ba is the buffer address
        local   pfcb
;;
;;      set up the file control block for the file
;;      look for file name = 1 or 2
@c      set     1       ;;assume true to begin with
        irpc    ?c,fn   ;;look through characters of name
        if      not ('&?C' = '1' or '&?C' = '2')
@c      set     0       ;;clear if not 1 or 2
        endm
;;      @c is true if fn = 1 or 2 at this point
        if      @c      ;;then fn = 1 or 2
;;      fill from default area
        if      nul ft  ;;type specified?
@c      set     12      ;;both name and type
        else
```

Listing 9-28. (continued)

```
@c      set     9       ;;name only
        endif
        filldef fcb&fid,(fn-1)*16,@c    ;;to select the fcb
        jmp     pfcb    ;;past fcb definition
        ds      @c      ;;space for drive/filename/type
        fillnam ft,12-@c        ;;series of db's
        else
        jmp     pfcb    ;;past initialized fcb
        if      nul dn
        db      0       ;;use default drive if name is zero
        else
        db      '&DN'-'A'+1 ·   ;;use specified drive
        endif
        fillnam fn,8    ;;fill file name
;;      now generate the file type with padded blanks
        fillnam ft,3    ;;and three character type
        endif
fcb&fid         equ     $-12    ;;beginning of the fcb
        db      0       ;;extent field 00 for setfile
;;      now define the 3 byte field, and disk map
        ds      20      ;;x,x,rc,dm0...dm15,cr fields
;;
        if      fid&typ<=2      ;;in/outfile
;;      generate constants for infile/outfile
        fillnxt         ;;@nxtb=0 on first call
        if      bs+0<@sect
;;      bs not supplied, or too small
@bs     set     @sect   ;;default to one sector
        else
;;      compute even buffer address
@bs     set     (bs/@sect)*@sect
        endif
;;
;;      now define buffer base address
        if      nul ba
;;      use next address after @nxtb
fid&buf         set     buffers+@nxtb
;;      count past this buffer
@nxtb   set     @nxtb+ bs
        else
fid&buf         set     ba
        endif
;;      fid&buf is buffer address
fid&adr:
        dw      fid&buf
```

Listing 9.28. (continued)

```
;;
fid&siz         equ     @bs     ;;literal size
fid&len:
        dw      @bs     ;;buffer size
fid&ptr:
        ds      2       ;;set in infile/outfile
;;      set device number
@&fid   set     @nxtd   ;;next device
@nxtd   set     @nxtd+1
        endif   ;;of fid&typ<=2 test
pfcb:   endm
;
file    macro   md,fid,dn,fn,ft,bs,ba
;;      create file using mode md:
;;              infile = 1      input file
;;              outfile = 2     output file
;;              setfile = 3     setup fcb
;;      (see fillfcb for remaining parameters)
        local   psub,msg,pmsg
        local   pnd,eod,eob,pnc
;;      construct the file control block
;;
fid&typ         equ     md      ;;set mode for later ref's
        fillfcb fid,dn,fn,ft,bs,ba
        if      md=3    ;;setup fcb only, so exit
        exitm
        endif
;;      file control block and related parameters
;;      are created inline, now create io function
        jmp     psub    ;;past inline subroutine
        if      md=1    ;;input file
get&fid:
        else
put&fid:
        push    psw     ;;save output character
        endif
        lhld    fid&len ;;load current buffer length
        xchg            ;;de is length
        lhld    fid&ptr ;;load next to get/put to hl
        mov     a,l     ;;compute cur-len
        sub     e
        mov     a,h
        sbb     d       ;;carry if next<length
        jc      pnc     ;;carry if len gtr current
;;      end of buffer, fill/empty buffers
        lxi     h,0
        shld    fid&ptr ;;clear next to get/put
```

Listing 9-28. (continued)

```
pnd:
;;      process next disk sector:
        xchg            ;;fid&ptr to de
        lhld    fid&len ;;do not exceed length
;;      de is next to fill/empty, hl is max len
        mov     a,e     ;;compute next-len
        sub     l       ;;to get carry if more
        mov     a,d
        sbb     h       ;;to fill
        jnc     eob
;;      carry gen'ed, hence more to fill/empty
        lhld    fid&adr ;;base of buffers
        dad     d       ;;hl is next buffer addr
        xchg
        mvi     c,@dma  ;;set dma address
        call    @bdos   ;;dma address is set
        lxi     d,fcb&fid       ;;fcb address to de
        if      md=1    ;;read buffer function
        mvi     c,@frd  ;;file read function
        else
        mvi     c,@fwr  ;;file write function
        endif
        call    @bdos   ;;rd/wr to/from dma address
        ora     a       ;;check return code
        jnz     eod     ;;end of file/disk?
;;      not end of file/disk, increment length
        lxi     d,@sect ;;sector size
        lhld    fid&ptr ;;next to fill
        dad     d
        shld    fid&ptr ;;back to memory
        jmp     pnd     ;;process another sector
;;
eod:
;;      end of file/disk encountered
        if      md=1    ;;input file
        lhld    fid&ptr ;;length of buffer
        shld    fid&len ;;reset length
        else
;;      fatal error, end of disk
        local   emsg
        mvi     c,@msg  ;;write the error
        lxi     d,emsg
        call    @bdos   ;;error to console
        pop     psw     ;;remove stacked character
        jmp     filerr  ;;usually reboots
```

Listing 9-28. (continued)

```
emsg:   db      cr,lf
        db      'disk full: &FID'
        db      '$'
        endif
;;
eob:
;;      end of buffer, reset dma and pointer
        lxi     d,@tbuf
        mvi     c,@dma
        call    @bdos
        lxi     h,0
        shld    fid&ptr ;;next to get
;;
pnc:
;;      process the next character
        xchg            ;;index to get/put in de
        lhld    fid&adr ;;base of buffer
        dad     d       ;;address of char in hl
        xchg            ;;address of char in de
        if      md=1    ;;input processing differs
        lhld    fid&len ;;for eof check
        mov     a,l     ;;0000?
        ora     h
        mvi     a,eof   ;;end of file?
        rz              ;;zero flag if so
        ldax    d       ;;next char in accum
        else
;;      store next character from accumulator
        pop     psw     ;;recall saved char
        stax    d       ;;character in buffer
        endif
        lhld    fid&ptr ;;index to get/put
        inx     h
        shld    fid&ptr ;;pointer updated
;;      return with non zero flag if get
        ret
;;
```

Listing 9-28. (continued)

```
psub:   ;;past inline subroutine
        xra     a               ;;zero to acc
        sta     fcb&fid+12      ;;clear extent
        sta     fcb&fid+32      ;;clear cur rec
        lxi     h,fid&siz       ;;buffer size
        shld    fid&len         ;;set buff len
        if      md=1    ;;input file
        shld    fid&ptr ;;cause immediate read
        mvi     c,@opn  ;;open file function
        else            ;;output file
        lxi     h,0     ;;set next to fill
        shld    fid&ptr ;;pointer initialized
        mvi     c,@del
        lxi     d,fcb&fid       ;;delete file
        call    @bdos   ;;to clear existing file
        mvi     c,@mak  ;;create a new file
        endif
;;      now open (if input), or make (if output)
        lxi     d,fcb&fid
        call    @bdos   ;;open/make ok?
        inr     a       ;;255 becomes 00
        jnz     pmsg
        mvi     c,@msg  ;;print message function
        lxi     d,msg   ;;error message
        call    @bdos   ;;printed at console
        jmp     filerr  ;;to restart
msg:    db      cr,lf
        if      md=1    ;;input message
        db      'no &FID file'
        else
        db      'no dir space: &FID'
        endif
        db      '$'
pmsg:
        endm
;
finis   macro   fid
;;      close the file(s) given by fid
        irp     ?f,<fid>
;;      skip all but output files
        if      ?f&typ=2
        local   eob?,peof,msg,pmsg
;;      write all partially filled buffers
```

Listing 9-28. (continued)

```
eob?:   ;;are we at the end of a buffer?
        lhld    ?f&ptr  ;;next to fill
        mov     a,l     ;;on buffer boundary?
        ani     (@sect-1) and 0ffh
        jnz     peof    ;;put eof if not 00
        if      @sect>255
;;      check high order byte also
        mov     a,h
        ani     (@sect-1) shr 8
        jnz     peof    ;;put eof if not 00
        endif
;;      arrive here if end of buffer, set length .
;;      and write one more byte to clear buffs
        shld    ?f&len  ;;set to shorter length
peof:   mvi     a,eof   ;;write another eof
        push    psw     ;;save zero flag
        call    put&?f
        pop     psw     ;;recall zero flag
        jnz     eob?    ;;non zero if more
;;      buffers have been written, close file
        mvi     c,@cls
        lxi     d,fcb&?f        ;;ready for call
        call    @bdos
        inr     a       ;;255 if err becomes 00
        jnz     pmsg
;;      file cannot be closed
        mvi     c,@msg
        lxi     d,msg
        call    @bdos
        jmp     pmsg    ;;error message printed
msg:    db      cr,lf
        db      'cannot close &?F'
        db      '$'
pmsg:
        endif
        endm    ;;of the irp
        endm
;
erase   macro   fid
;;      delete the file(s) given by fid
        irp     ?f,<fid>
        mvi     c,@del
        lxi     d,fcb&?f
        call    @bdos
        endm    ;;of the irp
        endm
;
```

Listing 9-28. (continued)

```
direct macro    fid
;;      perform directory search for file
;;      sets zero flag if not present
        lxi     d,fcb&fid
        mvi     c,@dir
        call    @bdos
        inr     a       ;00 if not present
        endm
;
rename macro    new,old
;;      rename file given by "old" to "new"
        local   psub,ren0
;;      include the rename subroutine once
        jmp     psub
@rens:  ;;rename subroutine, hl is address of
        ;;old fcb, de is address of new fcb
        push    h       ;;save for rename
        lxi     b,16    ;;b=00,c=16
        dad     b       ;;hl = old fcb+16
ren0:   ldax    d       ;;new fcb name
        mov     m,a     ;;to old fcb+16
        inx     d       ;;next new char
        inx     h       ;;next fcb char
        dcr     c       ;;count down from 16
        jnz     ren0
;;      old name in first half, new in second half
        pop     d       ;;recall base of old name
        mvi     c,@ren  ;;rename function
        call    @bdos
        ret             ;;rename complete
psub:
rename macro    n,o     ;;redefine rename
        lxi     h,fcb&o ;;old fcb address
        lxi     d,fcb&n ;;new fcb address
        call    @rens   ;;rename subroutine
        endm
        rename  new,old
        endm
;
get     macro   dev
;;      read character from device
        if      @&dev <= @lst
```

Listing 9-28. (continued)

```
;;      simple input
        mvi     c,@&dev
        call    @bdos
        else
        call    get&dev
        endm
;
;
put     macro   dev
;;      write character from accum to device
        if      @&dev <= @lst
;;      simple output
        push    psw     ;;save character
        mvi     c,@&dev ;;write char function
        mov     e,a     ;;ready for output
        call    @bdos   ;;write character
        pop     psw     ;;restore for testing
        else
        call    put&dev
        endm
```

Listing 9-28. (continued)

The equates that follow define the usual BDOS entry points and functions along with the disk sector size (@SECT) and special nongraphic characters (EOF, CR, LF, and TAB). The equates for @KEY through @LST are used in the GET and PUT macros to determine the source or destination device. The INFILE, OUTFILE, and SETFILE equates are used in the FILE macro as mnemonics for the file mode attribute.

FILLNAM is a utility macro used in the construction of a File Control Block. FILLNAM accepts a filename or filetype along with a field size and builds a sequence of DBs that fill the name or type field with padded blanks.

FILLDEF is a utility macro similar to FILLNAM, but FILLDEF fills the File Control Block name or type field from the default File Control Block at @TFCB or @TFCB+16. FILLDEF is invoked to extract either the default filename (first eight characters) or default filetype (following three-character field).

The FILLDEF macro constructs an inline subroutine to perform the data move operation the first time it is invoked and calls the inline subroutine (@DEF) on subsequent invocations.

FILLNXT initializes two assembly time variables: @NXTB and @NXTD. @NXTB counts the accumulated size of buffers as they are automatically allocated in the FILE statement. @NXTD counts files in the FILE macro for later reference in GET and PUT statements. They are included within a macro, so that they are properly initialized in the two successive passes of the macro assembler. FILLNXT is invoked by the FILE macro where the expansion initializes @NXTB and @NXTD. FILLNXT then redefines itself as an empty macro, so that subsequent FILE invocations do not reset the two counters.

The macro FILLFCB constructs a File Control Block in the CP/M standard format, where FID is the file identifier; DN is the disk name; FN is the filename; FT is the filetype; BS is the buffer size, and BA is the buffer address, as described in the FILE statement above. Recall that some of these parameters might be empty, causing default conditions to be selected.

The FILLFCB macro begins by searching for a "1" or a "2" as the FN parameter, indicating that default name 1 or 2 is to be selected for the file. The IRPC loop involving ?C results in a value of 1 for @C if either FN=1 or FN=2, and a value of 0 for @C if FN is not 1 or 2. The FILLFCB macro then selects either the default name or the user-specified name along with the default or user-specified drive number. The equate for FCB&FID then produces the address of the File Control Block for the file identifier followed by DB 0 for the extent field and DS 20 for the remainder of the File Control Block.

The remainder of the FILLFCB macro is devoted to storage allocation for buffer areas. The @BS variable is set to the buffer size after rounding and size checks. FID&BUF then becomes the address of the file buffer area, and FID&ADR labels a DW containing this literal value. FID&SIZ becomes the literal size of the buffer, and FID&LEN labels a DW containing the literal size. FID&PTR is also allocated as a double byte that subsequently holds the buffer index of the next character to get or put in the file. All of these values are used in the file operations that occur later.

The principal file access macro, FILE, sets up the File Control Block, buffers, and access subroutines for a file. Similar to the FILLFCB macro, the parameters FID, DN, FN, FT, BS, and BA describe the particular characteristics of a file. The MD parameter, however, indicates the file mode and must have the value 1, 2, or 3. The FILE macro begins by assigning the mode value to FID&TYP, so that subsequent macros can determine the type of access for this file. The FILLFCB macro is then invoked to construct the File Control Block for this macro and sets generally useful parameters for the file, as discussed previously. The FILE macro then generates the label GET&FID for input files or PUT&FID for output files, followed by a subroutine that GETs a single character or PUTs a single character for this file.

The GET&FID reads a single character from the input buffer and, when the input buffer is exhausted, fills the buffer area again in preparation for following GET operations. Upon detecting a real end-of-file, the EOF character is returned with the zero flag set. Similarly, the PUT&FID subroutine generated for output files stores the accumulator character into the output buffer at the next character position and, when the buffer is full, writes the sequence of sectors and returns to accept more output characters. In the case of an output error, the appropriate message is printed, and control transfers to FILERR, which usually remains at 0000H, causing a system reboot.

The generated code that follows the label PSUB initializes the file pointers to the proper position for file access. The file extent and next record fields of the File Control Blocks are zeroed for both input and output files. In the case of an input file, the buffer index variable FID&PTR is set to the end of the buffer, causing an immediate read operation when the first character is read. In the case of an output file, the FID&PTR is set to zero, indicating that the next position to fill is the first character of the output buffer. If the file is an output file, any duplicate files are erased, and a new file is created. In both cases, the file is opened upon completion of the FILE operation, and the buffer pointers are set for the next GET or PUT invocation. Note that the FILE statement is executable; it must occur ahead of the GET or PUT statements for the file and performs its function each time control passes through the FILE machine code.

The FINIS macro serves to empty the output buffers and close the file for output. Input files are skipped because no actions need take place to close an input file. The FINIS macro fills the remaining buffer segment (one size sector) with EOFs, then writes the partially filled buffers.

The ERASE macro accepts a file identifier or list of file identifiers and successively calls the BDOS to erase each file, while the DIRECT macro searches for a single file given by the file identifier FID. In the case of the DIRECT macro, the zero flag is cleared if the file exists. No prechecks are made to see if the file exists before the ERASE operation takes place, although erasing a nonexistent file is of no consequence. The DIRECT macro can, of course, be used to check if a file exists before the ERASE is executed.

The RENAME macro allows a file to be renamed by accepting two file identifiers, denoted by NEW and OLD. These file identifiers must correspond to the FCB names created by FILLFCB in an earlier FILE invocation, and have the effect of renaming the OLD file to the NEW filename. This is accomplished within the RENAME macro through an inline subroutine, called @RENS, which is included the first time the RENAME macro is invoked. The inline subroutine moves the new File Control Block information (first sixteen bytes) into the second half of the old File Control Block in the form required for a rename operation under CP/M. (See the CP/M documentation.) The BDOS is then called to perform the rename function. There is no check to ensure the old file exists before the rename takes place.

The GET and PUT macros are similar in structure: both accept a device or file identifier as the formal parameter DEV and perform the corresponding input or output function on that device. If the device is a simple peripheral, the BDOS is called directly to perform the input and output function. If, instead, the device name was created by a FILE macro, the corresponding GET&FID or PUT&FID subroutine is called to accomplish the input or output operation. Note that the accumulator is preserved (PUSH PSW) upon output to a simple peripheral within the PUT macro; the save/restore sequence is performed within the PUT&FID subroutine if the destination is a disk file.

Listings 9-29 shows the full expansion of a segment of the case conversion program of Listing 9-27 (using the "+M" assembly parameter). It begins with the invocation of FILE, followed by FILLFCB, again followed by FILLDEF. The @DEF subroutine is included inline, and the FILLDEF macro is redefined to exclude the subroutine. Upon completion of the FCB construction, the file parameters are generated, as shown in Listing 9-29b, along with the beginning of the GETSOURCE subroutine.

The conditional assembly ignores the portions of this FILE macro expansion that are related to output files but includes the machine code for the input SOURCE file. In each case, the &FID labels result in names with the prefix or suffix SOURCE, associating the generated labels with this internal name. The machine code that initializes the File Control Block fields and buffer pointer follows the label ??0001. Upon completion of the FILE macro, the SOURCE file is ready for access. Each call to GETSOURCE reads one more character into the accumulator. Due to the length of the expanded macro form, the remainder of the case translation program is not shown.

To illustrate the facilities of the SEQIO macro library, two additional programs are given. The first, called PRINT, formats the output from the macro assembler for printing on the system line printer. The second, called MERGE, performs a simple merge operation on two disk files.

```
                        FILE     INFILE,SOURCE,,1,,2000
   +                    LOCAL    PSUB,MSG,PMSG
   +                    LOCAL    PND,EOD,EOB,PNC
0001+=           SOURCETYP       EQU      INFILE
   +                    FILLFCB  SOURCE,,1,,2000,
   +                    LOCAL    PFCB
0001+#            @C    SET      1
   +                    IRPC     ?C,1
   +             @C     IF       NOT ('&?C' = '1' OR '&?C' = '2')
   +             @C     SET      0
   +                    ENDM
   +                    IF       NOT ('1' = '1' OR '1' = '2')
   +             @C     SET      0
   +                    ENDM
   +                    IF       @C
   +                    IF       NUL
000C+#           @C     SET      12
   +                    ELSE
   +             @C     SET      9
   +                    ENDIF
   +                    FILLDEF  FCBSOURCE,(1-1)*16,@C
   +                    LOCAL    PSUB
0103+C30F01             JMP      ??0009
   +             @DEF:
0106+7E                 MOV      A,M
0107+12                 STAX     D
0108+23                 INX      B
0109+13                 INX      D
010A+0D                 DCR      C
010B+C20601             JNZ      @DEF
010E+C9                 RET
   +             ??0009:
   +             FILLDEF         MACRO    ?FCB,?F,?L
   +                    LXI      H,@TFCB+?F
   +                    LXI      D,?FCB
   +                    MVI      C,?L
   +                    CALL     @DEF
```

Listing 9-29. Sample FILE Expansion Segment

```
    +               ENDM
    +               FILLDEF FCBSOURCE,(1-1)*16,@C
010F+215C00         LXI     H,@TFCB+(1-1)*16
0112+111D01         LXI     D,FCBSOURCE
0115+0E0C           MVI     C,@C
0117+CD0601         CALL    @DEF
    +               ENDM
    +               ENDM
011A+C34401         JMP     ??0008
011D+               DS      @C
    +
0000+#      @CNT    SET     12-@C
    +               IRPC    ?FC,
    +               IF      @CNT=0 OR NUL ?FC
    +               EXITM
    +               ENDIF
    +               DB      '&?FC'
    +       @CNT    SET     @CNT-1
    +               ENDM
    +               IF      @CNT=0 OR NUL
    +               EXITM
    +               REPT    @CNT
    +               DB      ' '
    +               ENDM
    +               ENDM
    +               ELSE
    +               JMP     ??0008
    +               IF      NUL
    +               DB      0
    +               ELSE
    +               DB      ''='A'+1
    +               ENDIF
    +               FILLNAM 1,8
    +               FILLNAM ,3
    +               ENDIF
011D+=      FCBSOURCE       EQU     $-12
0129+00             DB      0
012A+               DS      20
    +               IF       SOURCETYP<=2
    +               FILLNXT
0000+#      @NXTB   SET     0
0006+#      @NXTD   SET      @LST+1
    +       FILLNXT          MACRO
    +               ENDM
```

Listing 9-29. (continued)

```
    +                  ENDM
    +                  IF       2000+0<@SECT
    +           @BS    SET      @SECT
    +                  ELSE
0780+#          @BS    SET      (2000/@SECT)*@SECT
    +                  ENDIF
    +                  IF       NUL
0370+#          SOURCEBUF       SET      BUFFERS+@NXTB
0780+#          @NXTB  SET      @NXTB+@BS
    +                  ELSE
    +           SOURCEBUF       SET
    +                  ENDIF
    +           SOURCEADR:
013E+7003              DW       SOURCEBUF
0780+=          SOURCESIZ       EQU      @BS
    +           SOURCELEN:
0140+8007              DW       @BS
    +           SOURCEPTR:
0142+                  DS       2
0006+#          @SOURCE         SET      @NXTD
0007+#          @NXTD  SET      @NXTD+1
    +                  ENDIF
    +           ??0008:         ENDM
    +                  IF       INFILE=3
    +                  EXITM
    +                  ENDIF
0144+C3B401            JMP      ??0001
    +                  IF       INFILE=1
    +           GETSOURCE:
    +                  ELSE
    +           PUTSOURCE:
    +                  PUSH     PSW
    +                  ENDIF
0147+2A4001            LHLD     SOURCELEN
014A+EB                XCHG
014B+2A4201            LHLD     SOURCEPTR
014E+7D                MOV      A,L
014F+93                SUB      E
0150+7C                MOV      A,H
0151+9A                SBB      D
0152+DA9D01            JC       ??0007
0155+210000            LXI      H,0
0158+224201            SHLD     SOURCEPTR
```

Listing 9-29. (continued)

```
    +          ??0004:
015B+EB               XCHG
015C+2A4001           LHLD      SOURCELEN
015F+7B               MOV       A,E
0160+95               SUB       L
0161+7A               MOV       A,D
0162+9C               SBB       H
0163+D28F01           JNC       ??0006
0166+2A3E01           LHLD      SOURCEADR
0169+19               DAD       D
016A+EB               XCHG
016B+0E1A             MVI       C,@DMA
016D+CD0500           CALL      @BDOS
0170+111D01           LXI       D,FCBSOURCE
    +                 IF        INFILE=1
0173+0E14             MVI       C,@FRD
    +                 ELSE
    +                 MVI       C,@FWR
    +                 ENDIF
0175+CD0500           CALL      @BDOS
0178+B7               ORA       A
0179+C28901           JNZ       ??0005
017C+118000           LXI       D,@SECT
017F+2A4201           LHLD      SOURCEPTR
0182+19               DAD       D
0183+224201           SHLD      SOURCEPTR
0186+C35B01           JMP       ??0004
    +          ??0005:
    +                 IF        INFILE=1
0189+2A4201           LHLD      SOURCEPTR
018C+224001           SHLD      SOURCELEN
    +                 ELSE
    +                 LOCAL     EMSG
    +                 MVI       C,@MSG
    +                 LXI       D,EMSG
    +                 CALL      @BDOS
    +                 POP       PSW
    +                 JMP       FILERR
    +          EMSG:  DB        CR,LF
    +                 DB        'disk full:  SOURCE'
    +                 DB        '$'
    +                 ENDIF
```

Listing 9-29. (continued)

```
    +          ??0006:
018F+118000          LXI      D,@TBUF
0192+0E1A            MVI      C,@DMA
0194+CD0500          CALL     @BDOS
0197+210000          LXI      H,0
019A+224201          SHLD     SOURCEPTR
    +          ??0007:
019D+EB              XCHG
019E+2A3E01          LHLD     SOURCEADR
01A1+19              DAD      D
01A2+EB              XCHG
    +                IF       INFILE=1
01A3+2A4001          LHLD     SOURCELEN
01A6+7D              MOV      A,L
01A7+B4              ORA      H
01A8+3E1A            MVI      A,EOF
01AA+C8              RZ
01AB+1A              LDAX     D
    +                ELSE
    +                POP      PSW
    +                STAX     D
    +                ENDIF
01AC+2A4201          LHLD     SOURCEPTR
01AF+23              INX      H
01B0+224201          SHLD     SOURCEPTR
01B3+C9              RET
    +          ??0001:
01B4+AF              XRA      A
01B5+322901          STA      FCBSOURCE+12
01B8_323D01          STA      FCBSOURCE+32
01BB+218007          LXI      H,SOURCESIZ
01BE+224001          SHLD     SOURCELEN
    +                IF       INFILE=1
01C1+224201          SHLD     SOURCEPTR
01C4+0E0F            MVI      C,@OPN
    +                ELSE
    +                LXI      H,0
    +                SHLD     SOURCEPTR
    +                MVI      C,@DEL
    +                LXI      D,FCBSOURCE
    +                CALL     @BDOS
    +                MVI      C,@MAK
    +                ENDIF
```

Listing 9-29. (continued)

```
01C6+111D01             LXI      D,FCBSOURCE
01C9+CD0500             CALL     @BDOS
01CC+3C                 INR      A
01CD+C2EC01             JNZ      ??0003
01D0+0E09               MVI      C,@MSG
01D2+11DB01             LXI      D,??0002
01D5+CD0500             CALL     @BDOS
01D8+C30000             JMP      FILERR
01DB+0D0A       ??0002:          DB      CR,LF
    +                   IF       INFILE=1
01DD+6E6F20534F         DB       'no SOURCE file'
    +                   ELSE
    +                   DB       'no dir space: SOURCE'
    +                   ENDIF
01EB+24                 DB       '$'
    +           ??0003:
    +                   ENDM
```

Listing 9-29. (continued)

The PRINT program, shown in Listing 9-30, executes under the Console Command Processor and takes the following form:

PRINT filename

where filename is the name of a previously assembled program. PRINT assumes that there is a PRN file on the disk and possibly a SYM file on the same disk drive. The PRN file is first printed, with a form-feed at the top of each 56-line page. If the SYM file exists, it is also printed using the same formatting. If the files are successfully printed, they are both erased from the disk.

The PRINT program begins by saving the console processor stack, with the intention of returning directly to the CCP without a system reboot. The input printer file is then defined with a FILE statement that specifies the internal name PRINT and obtains the filename from the console command line. The filetype, however, is set to PRN in this case. After performing an initial page eject, the program loops between the PRCYC (print cycle) and ENDPR (end print) labels by successively reading characters from the PRINT source and writing to the printer through the LISTING subroutine. On detecting an end-of-file character, control transfers to the ENDPR label where the PRN file is erased from the disk.

The program then checks for the presence of the SYM file by invoking the FILE macro with a SETFILE mode. This creates the proper File Control Block for the input file with type SYM but does not create buffers or open the file for access. Following the FILE macro, the DIRECT statement performs a directory search and, if the file is not present, control transfers to the ENDLST (end listing) label where execution terminates.

If the SYM file exists, the program performs another page eject and then opens the SYM file for access. Note that the third FILE macro accesses the SYM file using the internal name SYMBOL but shares the buffer areas of the PRINT file. The PRINT file has been erased at this point, so the buffers are available.

If the SYM file is present, the program loops between the SYCYCLE (symbol cycle) and ENDSY (end symbol) labels where characters are read from the SYMBOL file and again sent to the printer through the LISTING subroutine. Upon detecting the end-of-file, control passes to the ENDSY label where the SYM file is erased from the disk. If no errors occur, control eventually reaches the ENDLST label where the printer page is ejected. The entry stack pointer is then retrieved from OLDSP, and control returns to the Console Command Processor, completing execution of the PRINT program.

```
0100                    ORG     100H
                        MACLIB  SEQIO   ;SEQUENTIAL I/O LIB
                ;       PRINT THE X.PRN AND X.SYM FILES ON THE
                ;       LINE PRINTER WITH PAGE FORMATTING.
                ;
000C =          FF      EQU     0CH     ;FORM FEED
0038 =          MAXLINE         EQU     56      ;MAX LINES PER PAGE
                ;
                ;       SAVE THE ENTRY STACK POINTER
0100 210000             LXI     H,0
0103 39                 DAD     SP      ;ENTRY SP TO HL
0104 22CF03             SHLD    OLDSP   ;SAVE ENTRY SP
0107 31CF03             LXI     SP,STACK;SET TO LOCAL STACK
                ;
010A                    FILE    INFILE,PRINT,,1,PRN,1000
                ;       READ THE PRINT FILE UNTIL END OF FILE
01F2 CD8A03             CALL    EJECT   ;TOP OF PAGE
01F5            PRCYC:  GET     PRINT
01F8 FE1A               CPI     EOF
01FA CA0302             JZ      ENDPR   ;SKIP IF END FILE
```

Listing 9-30. Program for Line Printer Page Formatting

```
01FD CD5103           CALL    LISTING ;WRITE TO LISTING DEV
0200 C3F501           JMP     PRCYC
              ENDPR: ;END OF PRINT FILE, DELETE IT
0203                  ERASE   PRINT
              ;
              ;       CHECK FOR THE OPTIONAL .SYM FILE
020B                  FILE    SETFILE,SYMCHK,,1,SYM
023A                  DIRECT  SYMCHK  ;IS IT THERE?
0243 CA3C03           JZ      ENDLST  ;SKIP SYMBOL IF SO
              ;
              ;       SYMBOL FILE IS PRESENT, PAGE EJECT
0246 CD8A03           CALL    EJECT   ;TO TOP OF PAGE
0249                  FILE    INFILE,SYMBOL,,1,SYM,1000,PRINTBUF
              ;
              SYCYCLE:
0326                  GET     SYMBOL
0329 FE1A             CPI     EOF
032B CA3403           JZ      ENDSY   ;SKIP TO END ON EOF
032E CD5103           CALL    LISTING ;SEND TO PRINTER
0331 C32603           JMP     SYCYCLE ;FOR ANOTHER CHAR
              ;
0334          ENDSY: ERASE    SYMBOL  ;ERASE .SYM FILE
              ;
              ENDLST:         ;END OF LISTING - EJECT AND RETURN
033C CD8A03           CALL    EJECT
033F 2ACF03           LHLD    OLDSP   ;ENTRY STACK POINTER
0342 F9               SPHL            ;RESTORE STACK POINTER
0343 C9               RET             ;TO CCP
              ;
              ;       UTILITY SUBROUTINES
              LISTOUT:
                      ;SEND A SINGLE CHARACTER TO THE PRINTER
0344                  PUT     LST
034C 21D203           LXI     H,CHARC ;CHARACTER COUNTER
034F 34               INR     M       ;INCREMENT POSITION
0350 C9               RET
              ;
```

Listing 9-30. (continued)

```
              LISTING:
                     ;WRITE CHARACTER FROM REG-A TO LIST DEVICE
0351 FE0C            CPI     FF      ;FORM FEED?
0353 C25F03          JNZ     LIST0
0356 AF              XRA     A       ;CLEAR LINE COUNT
0357 32D103          STA     LINEC
035A 32D203          STA     CHARC   ;CLEAR TAB POSITION
035D 3E0C            MVI     A,FF    ;RESTORE FORM FEED
035F FE0A     LIST0: CPI     LF      ;END OF LINE?
0361 C27403          JNZ     LIST1
0364 AF              XRA     A       ;CLEAR TAB POSITION
0365 32D203          STA     CHARC
0368 21D103          LXI     H,LINEC ;LINE COUNTER
036B 34              INR     M       ;INCREMENTED
036C 7E              MOV     A,M     ;CHECK FOR END OF PAGE
036D FE38            CPI     MAXLINE ;LINE OVERFLOW?
036F D8              RC              ;RETURN IF NOT
0370 3600            MVI     M,0     ;CLEAR LINEC
0372 3E0C            MVI     A,FF    ;SEND PAGE EJECT
0374 FE09     LIST1: CPI     TAB     ;TAB CHARACTER?
0376 C28703          JNZ     LIST2
              ;      FEED BLANKS TO NEXT TAB POSITION
0379 3E20     TABOUT:        MVI     A,' '
037B CD4403          CALL    LISTOUT
037E 3AD203          LDA     CHARC   ;CHARACTER POSITION
0381 E607            ANI     7H      ;MOD 8
0383 C27903          JNZ     TABOUT  ;FOR ANOTHER BLANK
              ;      ON CHARACTER BOUNDARY
0386 C9              RET
              LIST2: ;SIMPLE CHARACTER
0387 C34403          JMP     LISTOUT ;PRINT AND RETURN
              ;
              EJECT: ;PERFORM PAGE EJECT
038A 3E0C            MVI     A,FF    ;FORM FEED
038C C34403          JMP     LISTOUT
              ;
              ;      DATA AREAS
038F                 DS      64      ;32 LEVEL STACK
              STACK:
03CF          OLDSP: DS      2       ;ENTRY STACK POINTER
03D1          LINEC: DS      1       ;LINE COUNTER
03D2          CHARC: DS      1       ;CHARACTER COUNTER
              ;
              BUFFERS:
03D3                 END
```

Listing 9-30. (continued)

The next program, MERGE, is more complicated. The MERGE program accepts two filenames as input, taking the general command form

MERGE filename

where filename is the name of a master file, with assumed filetype of MAS, as well as an update name with assumed filetype UPD. The files consist of varying length records, each of which starts with a six-character numeric sequence number followed by textual material and ends with a carriage return line-feed sequence. The lines of information in the master and update files are assumed to be in ascending numeric order according to their sequence numbers. The MERGE program reads these two files and merges the records together to form a new file consisting of numerically ascending, sequence-numbered lines.

Upon completion of the merge operation, the newly merged file becomes the new master file. Update records are properly interspersed within the new master file according to the numeric order, and any update record that matches a master record results in replacement of the master record by the update record. Upon successful completion of the merge operation, the original master file is renamed to have the filetype MBK (master back-up), the original update file is renamed to the filetype UBK (update back-up), and the newly created file becomes the new MAS file. In this way, the operator can return to the back-up files in case of error, so that the source data is not destroyed.

```
0100                    ORG     100H
                ;       FILE MERGE PROGRAM
                        MACLIB  SEQIO   ;SEQUENTIAL FILE I/O
                ;
0000 =          BOOT    EQU     0000H   ;SYSTEM REBOOT
0006 =          SEQSIZ  EQU     6       SIZE OF THE SEQUENCE #'S
03E8 =          USIZE   EQU     1000    ;UPDATE BUFFER SIZE
03E8 =          MSIZE   EQU     USIZE   ;MASTER BUFFER SIZE
07D0 =          NSIZE   EQU     USIZE+MSIZE     ;NEW BUFF SIZE
                ;
0100 31EC05             LXI     SP,STACK
0103 C3C801             JMP     START   ;TO PERFORM THE MERGE
                ;
                ;       UTILITY SUBROUTINES
                ;
```

Listing 9-31. File Merge Program

```
                DIGIT: ;TEST ACCUMULATOR FOR VALID DIGIT
                ;       RETURN WITH CARRY SET IF INVALID
0106 FE30               CPI     '0'
0108 D8                 RC              ;CARRY IF BELOW 0
0109 FE3A               CPI     '9' +1  ;CARRY IF BELOW 10
010B 3F                 CMC             ;NO CARRY IF BELOW 10
010C C9                 RET
                ;
                ;       ERROR MESSAGES FOR READU AND READM
                SEQERRU:
010D 7570646174         DB      'update seq error',0
                SEQERRM:
011E 6D61737465         DB      'master seq error',0
                ;
                ;       GENERATE READU AND READM SUBROUTINES
                        IRPC    ?F,UM
                ;       INLINE SEQUENCE NUMBER BUFFER
                ?F&SEQ:         DB      0       ;TO START PROCESSING
                        DS      SEQSIZ-1;REMAINING SPACE FOR SEQ#
                ;
                READ&?F:
                        LXI     H,?F&SEQ        ;SEQUENCE BUFFER
                        MOV     A,M             ;IS IT FF (END FILE)?
                        INR     A               ;FF BECOMES 00
                        RZ                      ;SKIP THE READ
                ;
                ;       READ THE SEQUENCE NUMBER PORTION
                        MVI     C,SEQSIZ        ;SIZE OF SEQUENCE #
                RD&?F&0:
                        PUSH    H               ;SAVE NEXT TO FILL
                        PUSH    B               ;SAVE NUMBER COUNT
                        GET     ?F&FILE         ;READ THE FILE
                        POP     B               ;RECALL COUNT
                        POP     H               ;RECALL NEXT FILL
                        CPI     EOF             ;END FILE?
                        JZ      EOF&?F
                        CALL    DIGIT           ;ASCII DIGIT?
                        LXI     D,SEQERR&?F     ;ERROR MESSAGE
                        JC      SEQERR          ;SEQUENCE ERROR
                ;       NO SEQUENCE ERROR, FILL NEXT DIGIT POSITION
                        MOV     M,A
                        INX     H               ;NEXT TO FILL
                        DCR     C               ;COUNT=COUNT-1
                        JNZ     RD&?F&0         ;FOR ANOTHER DIGIT
                        RET                     ;END OF FILL
                ;
```

Listing 9-31. (continued)

```
                EOF&?F:        ;END OF FILE, SET SEQ# TO 0FFH
                       MVI     A,0FFH
                       STA     ?F&SEQ          ;SEQ# SET TO FF
                       RET
                       ENDM
                ;
                SEQERR:
                ;      WRITE ERROR MESSAGE FROM (DE) TIL 00
018F 1A                LDAX    D
0190 B7                ORA     A
0191 CA0000            JZ      BOOT
                ;      OTHERWISE, MORE TO PRINT
0194 D5                PUSH    D
0195                   PUT     CON     ;WRITE TO CONSOLE
019D D1                POP     D
019E 13                INX     D
019F C38F01            JMP     SEQERR  ;FOR MORE CHARS
                ;
                WRITESEQ:
                       ;WRITE THE SEQUENCE NUMBER GIVEN BY HL
                       ;TO THE NEW FILE
01A2 0E06              MVI     C,SEQSIZ        ;SIZE OF SEQ#
01A4 7E         WRITO: MOV     A,M
01A5 23                INX     H               ;NEXT TO GET
01A6 E5                PUSH    H               ;SAVE NEXT ADDR
01A7 C5                PUSH    B               ;SAVE COUNT
01A8                   PUT     NEW             ;WRITE TO NEW
01AB C1                POP     B               ;RECALL COUNT
01AC E1                POP     H               ;RECALL ADDRESS
01AD 0D                DCR     C               ;COUNT=COUNT-1
01AE C2A401            JNZ     WRITO           ;FOR ANOTHER CHAR
01B1 C9                RET
                ;
```

Listing 9-31. (continued)

```
                ;       COMPARE THE UPDATE SEQUENCE NUMBER WITH
                ;       THE MASTER SEQUENCE NUMBER, SET:
                ;               CARRY IF UPDATE < MASTER
                ;               ZERO  IF UPDATE = MASTER
                ;               -ZERO IF UPDATE > MASTER
                COMPARE:
01B2 112F01             LXI     D,USEQ          ;UPDATE SEQ#
01B5 215F01             LXI     H,MSEQ          ;MASTER SEQ#
01B8 0E06               MVI     C,SEQSIZ        ;SEQUENCE SIZE
01BA 1A         CLOOP:  LDAX    D               ;UPDATE DIGIT
01BB BE                 CMP     M               ;UPDATE-MASTER
01BC D8                 RC                      ;CARRY IF LESS
01BD C0                 RNZ                     ;NZERO IF GTR
                ;       ITEMS ARE THE SAME, CHECK FOR OFFH
01BE FEFF               CPI     OFFH            ;END OF FILE
01C0 C8                 RZ                      ;BOTH ARE OFFH
01C1 13                 INX     D               ;NEXT UPDATE
01C2 23                 INX     H               ;NEXT MASTER
01C3 0D                 DCR     C               ;COUNT DOWN
01C4 C2BA01             JNZ     CLOOP           ;FOR ANOTHER DIGIT
01C7 C9                 RET                     ;ZERO FLAG IF EQUAL
                ;
                ;       MAIN PROGRAM STARTS HERE
                START:
                        ;UPDATE FILE, WITH ASSUMED .UPD TYPE
01C8                    FILE    INFILE,UFILE,,1,UPD,USIZE
                        ;
                        ;MASTER FILE, WITH ASSUMED TYPE .MAX
02B0                    FILE    INFILE,MFILE,,1,MAS,MSIZE
                        ;
                        ;NEW FILE, TEMP.$$$ (RENAMED UPON EOF'S)
038C                    FILE    OUTFILE,NEW,,TEMP,$$$,NSIZE
                        ;
047D CD3501             CALL    READU   ;INITIALIZE UPDATE RECORD
0480 CD6501             CALL    READM   ;INITIALIZE MASTER RECORD
                MERGE:  ;MAIN MERGING LOOP
0483 CDB201             CALL    COMPARE ;CARRY SET IF UPDATE<MASTER
0486 CAAD04             JZ      SAME    ;ZERO IF IDENTICAL SEQ#
0489 D2C804             JNC     MASLOW  ;MASTER LOW?
                ;
                ;       UPDATE SEQUENCE NUMBER IS LOW
048C 212F01             LXI     H,USEQ  ;COPY SEQUENCE NUMBER
048F CDA201             CALL    WRITESEQ;WRITE THE SEQUENCE #
                ;
```

Listing 9-31. (continued)

```
                ULOOP: ;UPDATE RECORD TO NEW FILE
0492                   GET      UFILE    ;CHARACTER TO A
0495 F5                PUSH     PSW      ;SAVE IT
0496                   PUT      NEW      ;OUTPUT TO NEW FILE
0499 F1                POP      PSW      ;RECALL CHARACTER
049A FE0A              CPI      LF       ;LINE FEED?
049C CAA704            JZ       ENDUP
049F FE1A              CPI      EOF
04A1 CAA704            JZ       ENDUP
04A4 C39204            JMP      ULOOP    ;CYCLE IF NOT END REC
                ;
04A7 CD3501     ENDUP: CALL     READU    ;READ ANOTHER SEQ#
04AA C38304            JMP      MERGE    ;FOR ANOTHER RECORD
                ;
                ;
                SAME:  ;SEQUENCE NUMBERS ARE IDENTICAL
04AD 3A5F01            LDA      MSEQ     ;CHECK FOR OFFH
04B0 FEFF              CPI      0FFH
04B2 CAE904            JZ       ENDMERGE
                ;      NOT THE SAME, DELETE MASTER RECORD
04B5            DELMAS:         GET      MFILE
04B8 FE1A              CPI      EOF      ;END OF FILE?
04BA CAC204            JZ       GETMAS   ;GET SEQ# FF
04BD FE0A              CPI      LF
04BF C2B504            JNZ      DELMAS   ;FOR ANOTHER CHAR
04C2 CD6501     GETMAS:         CALL     READM    ;TO NEXT RECORD
04C5 C38304            JMP      MERGE    ;FOR ANOTHER
                ;
                MASLOW:         ;MASTER SEQUENCE NUMBER IS LOW
04C8 215F01            LXI      H,MSEQ
04CB CDA201            CALL     WRITESEQ;SEQUENCE NUMBER
04CE            MLOOP: GET      MFILE
04D1 F5                PUSH     PSW      ;SAVE MASTER CHARACTER
04D2                   PUT      NEW
04D5 F1                POP      PSW      ;LF OR EOF?
04D6 FE0A              CPI      LF
04D8 CAE304            JZ       ENDMS
04DB FE1A              CPI      EOF
04DD CAE304            JZ       ENDMS
04E0 C3CE04            JMP      MLOOP    ;MORE TO COPY
                ;
04E3 CD6501     ENDMS: CALL     READM    ;READ NEW SEQ NUMBER
04E6 C38304            JMP      MERGE    ;TO MERGE ANOTHER
```

Listing 9-31. (continued)

```
                ;
                ENDMERGE:
                        ;CLOSE ALL FILES FOR RENAMING
04E9                    FINIS   <UFILE,MFILE,NEW>
                        ;OLD MASTER FILE FOR ERASE/RENAME
0529                    FILE    SETFILE,OLDMAS,,1,MBK
0558                    ERASE   OLDMAS
                        ;RENAME MASTER TO .MBK
0560                    RENAME  OLDMAS,MFILE
                        ;
                        ;OLD UPDATE FILE FOR ERASE/RENAME
0580                    FILE    SETFILE,OLDUPD,,1,UBK
05AF                    ERASE   OLDUPD
                        ;RENAME UPDATE TO .UBK
05B7                    RENAME  OLDUPD,UFILE
                        ;
                        ;RENAME NEW TO MASTER FILE
05C0                    RENAME  MFILE,NEW
05C9 C30000             JMP     BOOT
                ;
05CC                    DS      32      ;16 LEVEL STACK
                STACK:
                ;       BUFFER AREA
                BUFFERS:
146C =          MEMSIZE         EQU     BUFFERS+NXTB    ;END OF MEMORY
05EC                    END
```

Listing 9-31. (continued)

The MERGE program, shown in Listing 9-31, begins with utility subroutines, including the DIGIT subroutine that tests for valid decimal digits in sequence numbers. The IRPC that follows the DIGIT subroutine generates two distinct subroutines, called READU and READM, for reading the update and master files, respectively. The generation of these two subroutines has been suppressed in the listing to keep the listing short. (See Section 10.) These two READ subroutines fill their respective sequence number buffers from the input source, so that the merge operation can take place based on the current sequence number values. Upon detecting an end-of-file, the sequence number is set to 0FFH as a signal that the input source has been exhausted.

The SEQERR subroutine reports an error condition when a nonnumeric character is detected in the sequence number field. Although the error reporting is spartan, sequence errors are easily found using the TYPE command on the master or update file. The WRITESEQ subroutine is called whenever the source for the next record has been determined. The COMPARE subroutine determines the next source record (master or update) by comparing the buffered sequence numbers from left to right while they are equal. If a mismatch occurs in the sequence number scan, COMPARE returns with the carry flag and zero flag set to indicate which file holds the next source record.

Execution of the MERGE program begins following the START label where the update, master, and new files are defined. The UFILE and MFILE sources are defined with the same buffer sizes, as determined by the earlier USIZE and MSIZE equates. Both take their primary name from the default value specified at the CCP level by the operator. The new file is created as a temporary, with filename TEMP and filetype $$$, but is renamed upon completion of the program to become the master file.

The merge operation proceeds in Listing 9-31 as follows. First the READU and READM subroutines are called to fill the sequence number buffers. The loop between MERGE and ENDMERGE is then repetitively executed until the merge is complete. On each iteration of this loop, the COMPARE subroutine is called to compare the buffered sequence numbers. If the update sequence number is smaller than the master sequence number, it is moved to the new file, and data is copied from the update file to the new file until the end of the current record is encountered. Upon completion of the copy operation, the READU subroutine is called again to refill the update sequence number buffer.

If the COMPARE subroutine instead detects equal sequence numbers, control transfers to the SAME label, where the master record is deleted. Alternatively, the COMPARE subroutine causes control to transfer to the MASLOW label when the master sequence number is lower than the update sequence number. In this case, the master sequence number and data record are copied to the new file in exactly the same manner as an update record.

Upon completion of the merge operation, indicated by an end-of-file in both the update and master files, control transfers to the ENDMERGE label where the files are closed and renamed. Following the FINIS statement, the previous MBK file (possibly from an earlier execution) is erased so that the current master (MAS) can be renamed to the master back-up (MBK). Similarly, any previous UBK file is erased, and the current update file is renamed to become the new UBK file. Finally, the new file (TEMP.$$$) is renamed to become the new master file (MAS) before execution stops.

Listing 9-32 shows an example of the files involved in a typical merge operation. In this application, the sequence numbers control the ordering of a list of names that is updated periodically. The NAMES.MAS file, which is the original master, is updated by merging with the NAMES.UPD file, also shown in the listing. The merge operation is initiated by typing

```
MERGE NAMES
```

and, upon completion, produces the new NAMES.MAS shown in the righthand column of Listing 9-32.

The SEQIO library is typical of the interface you can construct to provide a higher level interface between assembly language programs and their operating environment. Although the library shown here performs only simple sequential file input/output, you can construct more comprehensive libraries for random access based on this library.

```
        NAMES.MAS

000100 ABERCROMBIE, SIDNEY
000200 CARLSBAD, YOLANDA
000300 EGGBERT ,EBENEZER
000400 GRAVELPAUGH, HORTENSE
000500 ISENEARS, IGNATZ
000600 KRABNATZ, TILLY
000700 MILLYWATZ, RICARDO
000800 OPFATZ, ADOLPHO
000900 QUAGMIRE, DONALD
001000 TWITSWEET, LADNER
001090 VERANDA, VERONICA
001100 WILLOWANDER, PRATNEY
001200 YUPPGANDER, MANNY
```

```
        NAMES.UPD

000110 BERNSWEIGER, ALFRED
000200 CRUENCE, CLARENCE
000210 DENNINGSKI, HUBERT
000330 FINKLESTEIN, FRANK
000410 HILLSENFIELDS, RANDOLPH
000540 JOLLYFELLOW, JUNE
000620 LAMBAA, WILLY
000710 NEEBEND, ASTRID
000820 PRATTWITZ, HEADY
000930 RUBBLEMEYER, RUNYON
000960 SWIGSTITTS, ULYSSES
001010 UMPLANDER, XAVIER
001110 XYLOPH, ERHARDT
001210 ZEPLIPPS, EGGERWORTZ
```

```
            new NAMES.MAS

000100 ABERCROMBIE, SIDNEY
000110 BERNSWEIGER, ALFRED
000200 CRUENCE, CLARENCE
000210 DENNINGSKI, HUBERT
000300 EGGBERT, EBENEZER
000330 FINKLESTEIN, FRANK
000400 GRAVELPAUGH, HORTENSE
000410 HILLSENFIELDS, RANDOLPH
000500 ISENEARS, IGNATZ
000540 JOLLYFELLOW, JUNE
000600 KRABNATZ, TILLY
000620 LAMBAA, WILLY
000700 MILLYWATZ, RICARDO
000710 NEEBEND, ASTRID
000800 OPFATZ, ADOLPHO
000820 PRATTWITZ, HEADY
000900 QUAGMIRE, DONALD
000930 RUBBLEMEYER, RUNYON
000960 SWIGSTITTS, ULYSSES
001000 TWITSWEET, LADNER
001010 UMPLANDER, XAVIER
001090 VERANDA, VERONICA
001100 WILLOWANDER, PRATNEY
001110 XYLOPH, ERHARDT
001200 YUPPGANDER, MANNY
001210 ZEPLIPPS, EGGERWORTZ
```

Listing 9-32. Sample MERGE Disk Files

End of Section 9

Section 10
Assembly Parameters

You can include assembly parameters when you invoke the assembler that controls various assembler functions. The macro assembler is initiated with the name of the source file, followed by a dollar sign ($) and the assembly parameters. The parameters are indicated by single controls that denote particular functions. The character on the left below controls the function shown to the right.

Table 10-1. Assembly Parameters

Character	*Function*
A	the source disk for the .ASM file
H	the destination of the .HEX machine code file
L	the source disk for the .LIB files (see MACLIB)
M	MACRO listings in the .PRN file
P	the destination of the .PRN file containing the listing
Q	the listing of LOCAL symbols
S	the generation and destination of the .SYM file
1	pass 1 listing

Any or all of the above parameters can be included. The A, H, L, and S parameters are followed by the drive name to obtain or receive the data, where the drives are labeled A, B, . . . , Z. By convention, the X disk corresponds to the user's console; the P disk corresponds to the system line printer (logical list device), and the Z disk corresponds to a null file that is not recorded. The following is a valid assembly parameter list following the MAC command and source filename:

```
$PB AA HB SX
```

that directs the .PRN file to disk B, reads the .ASM file from disk A, directs the .HEX file to the B disk, and sends the .SYM file to the user's console. Blanks are optional between parameter specifications.

The parameters L, S, M, Q, and 1 can be preceded by + or - symbols that enable or disable their functions. These functions are

+L	lists input lines read from the macro library (see MACLIB).
−L	suppresses listing of the macro library (default value).
+S	appends the .SYM to the end of the .PRN output.
−S	suppresses the generation of the sorted Symbol Table.
+M	lists all macro lines as they are processed during assembly.
−M	suppresses all macro lines as they are read during assembly.
* M	lists only hex generated by macro expansions.
+Q	lists all LOCAL symbols in the symbol list.
−Q	suppresses all LOCAL symbols in the symbol list.
+1	produces a listing file on first pass (for macro debugging).
−1	suppresses listing on pass 1 (default).

The following is an example of a valid assembly parameter list that uses a number of the parameter specifications given above:

```
$PB+S-M HB
```

In this case, the .PRN file is sent to disk B with the symbol list appended (no .SYM file is created), all macro generations are suppressed, and the .HEX file is sent to disk B with the .PRN file.

The M parameter can be preceded by an asterisk (*), causing the assembler to list only macro generations that produce machine code. The asterisk suppresses the listing of the instructions that are produced; positions beyond the hex fields are not listed. Under normal operation, the macro assembler lists only generations that produce machine code, along with the generated line.

Given that disk d is the currently logged drive, the macro assembler defaults these parameters as follows: the .ASM and .LIB files are assumed to originate on drive d; the .HEX, .PRN, and .SYM files are sent to drive d; a Symbol Table is generated with LOCAL symbols suppressed. This means symbols beginning with ?? are not listed, and macro lines that generate machine code are listed. Note, however, that the filename following the MAC command can be preceded by a drive name, in which case the P parameter overrides the drive name, if supplied. Whenever a parameter is repeated in the assembly parameter specification, the last value is assumed. Valid assembly statements are shown below, assuming the file to be assembled is called SAMPLE.

```
MAC SAMPLE $PX+S-M
```

assembles the file SAMPLE.ASM with listing to the console, symbols at the console, and no listing of generated macros.

```
MAC A:SAMPLE $+S -M+Q
```

assembles sample.ASM from disk A, creating sample.PRN with appended symbols on the currently logged drive, suppressing generated macros, and listing symbols that begin with the characters ?? in addition to the usually listed symbols.

```
MAC SAMPLE
```

assembles SAMPLE.ASM from the currently logged drive, creating SAMPLE.PRN along with sample.SYM (containing the Symbol Table) and SAMPLE.HEX, which holds the Intel format hex file in the ASCII form.

```
MAC SAMPLE $AB HA PB +Q +S +L *M
```

assembles the SAMPLE.ASM file from drive B and produces the file SAMPLE.HEX on drive A, with the SAMPLE.PRN file on drive B. The Symbol Table includes ?? symbols. The Symbol Table is placed at the end of the .PRN file on drive B. The .LIB files are listed with the .PRN file as the .LIB files are read. The instructions that correspond to generated macro lines are not included, although generated machine code is listed.

In addition to the parameters shown above, you can intersperse controls throughout the assembly language source or library files. Interspersed controls are denoted by a $ in the first column of the input line, where the form shown on the left below corresponds to the action described on the right.

$ – PRINT	stops output listing by discarding formatted lines
$ + PRINT	enables the output printing when previously disabled
$ – MACRO	disables generated macro lines, as in – M above
$ + MACRO	enables full macro trace, as in + M above
$ * MACRO	enables partial macro trace, as in *M above

Because MAC allows each line to be optionally prefixed by a line number, the $ control can be included directly following this line number.

End of Section 10

Section 11
Debugging Macros

A number of common debugging practices can be used in developing macros and macro libraries. One technique, called iterative improvement, is often used in the design of programs and is most useful in building macros. The basic idea of iterative improvement is that a small portion of the overall macro set is first implemented and tested before continuing to more complicated macros. In this way, errors can be isolated at each step as the macro evolves. Further, if errors occur in the macro generations after a small portion of the macro set has been improved, it is most likely that the error is being caused by the macros that are changed.

In the case of the Hornblower Highway System macro libraries, for example, iterative improvement was used to evolve the final macro library. Only the simplest macros were first implemented, including the SETLITE, TIMER, and RETRY macros. (See Section 9.) Debugging facilities were then added to these macros, so that the programs could be traced at the console. Upon successful testing of the basic macro facilities, the PUSH?, CLOCK?, and TREAD? macros were individually written and tested, resulting in the final macro library.

At each step, you can use the various assembly parameters to control the debugging information. If the macro generations are not producing the proper machine code, it might be necessary to obtain a full trace, using the +M option when MAC is started. If the program produces too much output with the full trace enabled, you can use the $+MACRO and $-MACRO commands interspersed throughout the assembly language source program, resulting in full macro generation traces only in the regions selected for debugging consideration.

If macro generation errors are caused by macro libraries, you can use the +L parameter when MAC starts to cause the libraries to be included in the listing as they are read.

As a final consideration, it might be necessary to enable the first pass listing of the assembly language using the +1 parameter. In this case, MAC lists the program as it is being read on the first pass as well as the second pass. Note, however, that the listing contains spurious error messages on this pass that might disappear on the second pass. The first pass listing parameter allows you to view the macro generations on the two successive expansion passes to ensure that the assembler is processing the program in the same way in both cases.

If a macro expands improperly, and the source of the error is not evident after examining various traces, it might be necessary to remove the offending macro from the program and create an isolated smaller test case where the error is reproduced. Full traces can then be examined to determine the source of the error and, after fixing the macro, it can be replaced in the larger program and retested.

End of Section 11

Section 12
Symbol Storage Requirements

The maximum program size that can be assembled by MAC is determined only by the Symbol Table storage requirements for the program. The Symbol Table itself occupies the region above the macro assembler in memory, up to the base of the CP/M operating system. Thus, the size of the Symbol Table depends on the size of the current MAC version—approximately 12K program and data, plus 2.5K for I/O buffers—and the size of the user's CP/M configuration. The Symbol Table size is dynamically determined by MAC upon startup and fills as symbols are encountered. To provide some insight regarding storage requirements, the basic item size for identifiers and macros is given below.

A name used as a program label, data label, or variable in a SET or EQUATE requires

$$N = L + 5$$

bytes, where L is the length of the identifier name. Thus, the statement

```
PORTVAL   EQU   37FH
```

makes an entry into the Symbol Table that occupies

$$N = 7 + 5 = 12 \text{ bytes}$$

of Symbol Table space. Recall that LOCAL symbols take the form ??nnnn, which generates a name of length L = 6.

Macro storage is more complicated to compute. The general form is

$$M = L + 7 + H + T$$

where L is the macro name length; H is the parameter header storage requirement, and T is the macro text storage requirement, computed as

$$H = P_1 + P_2 + \ldots + P_n + n$$

where P_1 is the length of the first parameter name. The text length T is the number of characters in the macro body, including tab and end-of-line characters. Reserved symbols, however, are reduced to a single byte from their multicharacter representations. The jump, call, and return on condition operators, however, require their full character representations. Comments starting with double semicolon are not included in the character count. The comment line is backscanned to remove preceding tab or blank characters in this case. For example, the macro

```
LOADR    MACRO    REG,ALPHA ;FILL REGISTER crlf
         MVI      REG,'&ALPHA'    ;;DATA crlf
         ENDM crlf
```

contains a macro header, followed by two macro lines, where each line is written with tab characters (rather than spaces) and terminated by carriage return line-feeds (crlfs).

In this case, the macro name length (LOADR) is five characters (L = 5), and the parameter name lengths are three characters (REG) and five characters (ALPHA), resulting in the following parameter header storage requirement:

$$H = P_1 + P_2 + 2 = 3 + 5 + 2 = 10 \text{ bytes}$$

The first macro line contains a leading tab (one byte), the MVI instruction (reduced to one byte), another tab character (one byte), the operands REG,'&ALPHA' (twelve characters), and the end of line (two characters), for a total of seventeen bytes. Note that the comment, with the preceding tab, is removed from the line. The second line contains a tab (one byte), ENDM (one byte), and end-of-line (two characters) for a total of four bytes. Summing the textual characters, the total is T = 21 bytes. As a result, the total macro storage for LOADP is

$$M = L + 7 + H + T = 5 + 7 + 10 + 21 = 43 \text{ bytes}$$

No permanent storage is required for REPTs, IRPCs, or IRPs, although temporary storage in the Symbol Table is used while the groups are actively iterating. The characters contained within the group bounds (from the header to the corresponding ENDM) are stored in the Symbol Table in their literal form, with no reduction of reserved symbols to single bytes. Upon completion of the iteration, the storage is returned for other purposes. Similarly, active parameters for macro expansions require temporary storage in the Symbol Table. Storage is returned upon completion of the macro expansion.

In any case, a Symbol Table overflow message results if the total amount of free Symbol Table space is used up. As mentioned previously, the user can regenerate the CP/M system, up to the maximum memory space of the 8080 processor, to increase the symbol table area. The percentage of Symbol Table utilization is always printed at the console at the end of assembly. The printout takes the form:

```
0hhH USE FACTOR
```

where hh is a hexadecimal value in the range 00 to FF, where 00 results from an almost empty table, and FF is produced from an almost full table. The value 080H, for example, is printed when the Symbol Table is half full. Keep note of the use factor as a program develops to gauge the relative amount of free space as the program is enhanced.

In many of the examples shown in this manual, macros include inline subroutines that are generated at the first invocation and called upon subsequent invocations. (See the TYPEOUT macro in Listing 6-11, for example.) These subroutines can be included in the mainline program to reduce Symbol Table storage requirements, if necessary. In this case, the subroutines are assumed to exist the first time the macro is invoked, and thus are not generated by the macro.

End of Section 12

Section 13 RMAC, Relocating Macro Assembler

RMAC, the CP/M Relocating Macro Assembler, is a modified version of the CP/M Macro Assembler (MAC). RMAC produces a relocatable object file (REL), rather than an absolute object file (HEX), that can be linked with other modules produced by RMAC, or by other language translators such as PL/I-80, to produce an absolute file ready for execution. The differences between RMAC and MAC are described in the following subsections.

13.1 RMAC Operation

RMAC takes the command form:

RMAC filename.filetype

followed by optional assembly parameters. If the filetype is not specified, ASM is assumed. RMAC produces three files: a list file (PRN), a symbol file (SYM), and a relocatable object file (REL). Characters entered in the source file in lower-case appear in lower-case in the list file, except for macro expansions.

The assembly parameter H in MAC, used to control the destination of the HEX file, has been replaced by R, which controls the destination of the REL file. Directing the REL file to the console or printer (RX or RP) is not allowed, because the REL file does not contain ASCII characters.

The following example directs RMAC to assemble the file TEST.ASM, send the PRN file to the console, and put the symbol file (SYM) and the relocatable object file (REL) on drive B.

```
A>RMAC TEST $PX SB RB
```

13.2 Expressions

The operand field of a statement can consist of a complex arithmetic expression, as described in Section 3, with the following restrictions:

- In the expression A+B, if A evaluates to a relocatable value or an external, then B must be a constant.
- In the expression A-B, if A is an external, then B must be a constant.
- In the expression A-B, if A evaluates to a relocatable value, then B must be a constant, or B must be a relocatable value of the same relocation type as A. That is, both must appear in a CSEG or DSEG, or in the same COMMON block.
- In all other arithmetic and logical operations, both operands must be absolute.

An expression error ('E') is generated if an expression does not follow these restrictions.

13.3 Assembler Directives

The following assembler directives have been added to support relocation and linking of modules:

ASEG	use absolute location counter
CSEG	use code location counter
DSEG	use data location counter
COMMON	use common location counter
PUBLIC	symbol can be referenced in another module
EXTRN	symbol is defined in another module
NAME	name of module

The directives ASEG, CSEG, DSEG, and COMMON allow program modules to be split into absolute, code, data, and common segments. These segments can be rearranged in memory as needed at link time. The PUBLIC and EXTRN directives provide for symbolic references between program modules.

Note: symbol names can be up to 16 characters, but the first six characters of all symbols in PUBLIC, EXTRN, and COMMON statements must be unique, because symbols are truncated to six characters in the object module.

13.3.1 The ASEG Directive

The ASEG statement takes the form:

```
label     ASEG
```

and instructs the assembler to use the absolute location counter until otherwise directed. The physical memory locations of statements following an ASEG are determined at assembly time by the absolute location counter, which defaults to 0 and can be reset to another value by an ORG statement following the ASEG statement.

13.3.2 The CSEG Directive

The CSEG statement takes the form:

```
label     CSEG
```

and instructs the assembler to use the code location counter until otherwise directed. This is the default condition when RMAC begins an assembly. The physical memory locations of statements following a CSEG statement are determined at link time.

13.3.3 The DSEG Directive

The DSEG statement takes the form:

```
label     DSEG
```

and instructs the assembler to use the data location counter until otherwise directed. The physical memory locations of statements following a DSEG statement are determined at link time.

13.3.4 The COMMON Directive

The COMMON statement takes the form:

```
COMMON   /identifier/
```

and instructs the assembler to use the COMMON location counter until otherwise directed. The physical memory locations of statements following a COMMON statement are determined at link time.

13.3.5 The PUBLIC Directive

The PUBLIC statement takes the form:

```
PUBLIC   label{,label,...,label}
```

where each label is defined in the program. Labels appearing in a PUBLIC statement can be referred to by other programs that are linked using LINK-80.

13.3.6 The EXTRN Directive

The EXTRN statement takes the form:

```
EXTRN   label{,label,...,label}
```

The labels appearing in an EXTRN statement can be referenced but must not be defined in the program being assembled. They refer to labels in other programs that have been declared PUBLIC.

13.3.7 The NAME Directive

The NAME statement takes the form:

```
NAME        'text string'
```

The NAME statement is optional. It is used to specify the name of the relocatable object module produced by RMAC. If no NAME statement appears, the filename of the source file is used as the name of the object module. Module names identify modules within a library when using the LIB-80 library manager.

End of Section 13

Section 14
XREF

XREF is an assembly language cross-reference utility program used with the PRN and SYM files produced by MAC or RMAC to provide a summary of variable usage throughout the program.

XREF takes the command form:

```
XREF filename
```

The filename refers to two input files that are created using MAC or RMAC with the assumed (and unspecified) filetypes of PRN and SYM, and one output file with an assumed (and unspecified) filetype of XRF.

XREF reads the file, filename.PRN, line by line, attaches a line number prefix to each line, and writes each prefixed line to the file filename.XRF. During this process, XREF scans each line for any symbols that exist in the file filename.SYM.

After completing this copy operation, XREF appends to the file filename.XRF a cross-reference report that lists all the line numbers where each symbol in filename.SYM appears. It also flags with a # character each line number where the referenced symbol is defined.

XREF also reports the value of each symbol, as it appears in the file filename.SYM.

As an option, the file specification can include a drive name in the standard CP/M format, d:. When the drive name is specified, XREF associates all the files described above with the specified drive. Otherwise, it associates the files with the default drive.

XREF also allows you to direct the output file to the default list device instead of to the file filename.XRF. To use this option, add the string $p to the command line:

```
XREF filename $P
```

XREF allocates space for symbols and symbol references dynamically during execution. If no memory is available for an attempted symbol or symbol reference allocation, XREF issues an error message and terminates.

End of Section 14

Section 15
LINK-80

15.1 Introduction

LINK-80 is a utility program you can use to combine relocatable object modules into an absolute file ready for execution under CP/M or MP/M II.

There are two types of relocatable object modules. The first has a filetype of REL and is produced by PL/I-80, RMAC, or any other language translator that produces relocatable object modules in the Microsoft® format.

The second has a filetype of IRL and is generated by the CP/M library manager LIB-80. An IRL file contains the same information as a REL file but includes an index that enables faster searching of large libraries.

Upon successful completion, LINK-80 lists the following items at the console:

- the Symbol Table
- any unresolved symbols
- a Memory Map
- the Use Factor

The Memory Map shows the size and locations of the different segments. The Use Factor indicates the amount of available memory used by LINK-80 as a hexadecimal percentage.

LINK-80 writes the Symbol Table to a SYM file suitable for use with the CP/M Symbolic Instruction Debugger (SID™) and creates a COM or PRL file for direct execution under CP/M or MP/M II.

15.2 LINK-80 Operation

LINK-80 takes the general command form:

```
link filename1{,filename2,. . .,filenameN}
```

where filename1,. . .,filenameN are the names of the object modules to be linked. If you do not specify a filetype, LINK-80 assumes filetype REL.

LINK-80 produces two files:

- filename1.COM
- filename1.SYM

You can specify a different name for the COM and SYM files with a command of the form:

```
link newfilename=filename1{,filename2,. . .,filenameN}
```

LINK-80 supports a number of optional switches that control the link operation. These switches are described in the following section.

During the link process, LINK-80 can create up to eight temporary files on the default disk. The files are named:

```
XXABS.$$$    XXPROG.$$$    XXDATA.$$$    XXCOMM.$$$

YYABS.$$$    YYPROG.$$$    YYDATA.$$$    YYCOMM.$$$
```

LINK-80 deletes these files following termination. However, they can remain on the disk if LINK-80 halts due to an error condition.

15.3 Multi-line Commands

If a LINK-80 command does not fit on a single line (126 characters), the command can be extended by terminating the command line with an ampersand character. The ampersand can appear after any character in the command and need not follow a filename.

LINK-80 responds with an asterisk on the next line, at which point you can continue the command. LINK-80 allows any number of lines ending with the ampersand. The last line terminates with a carriage return, as in the following example. The Symbol Table and memory map would appear where vertical ellipses are shown.

```
A>link main, iomod1, iomod2, iomod3, iomod4, iomod5, &
LINK 1.3
*lib1[s], lib2[s], lib3[s], lib4&
*[s], lastmod[p2000&
*,d200]
        .
        .
A>
```

Note: you can use XSUB to submit multi-line commands to LINK-80.

15.4 LINK-80 Switches

LINK-80 supports optional run-time parameters called switches that control the link operation. All LINK-80 switches are enclosed in square brackets, separated by commas, and immediately follow one or more of the filenames in the command line.

All switches except the S switch can appear after any filename in the command line. The S switch must follow the filename to which it refers. For example,

```
A>LINK TEST[L4000],IOMOD,TESTLIB[S,NL,GSTART]
```

15.4.1 The Additional Memory (A) Switch

The A switch provides additional space for Symbol Table storage by decreasing the size of LINK-80's internal buffers. Use this switch only when necessary, as indicated by a MEMORY OVERFLOW error. Using the A switch causes LINK-80 to store its internal buffers on the disk, slowing down the linking process considerably, while allowing linking of larger programs.

15.4.2 The BIOS Link (B) Switch

The B switch is used to link a BIOS in a banked CP/M 3 system. LINK-80 aligns the data segment on a page boundary, puts the length of the code segment in the header, and defaults to the SPR filetype.

15.4.3 The Data Origin (D) Switch

The D switch specifies the origin of the data and common segments. If you do not use the D switch, LINK-80 places the data and common segments immediately after the program segment.

The D switch takes the form:

```
Dnnnn
```

where nnnn is the data origin in hexadecimal.

15.4.4 The Go (G) Switch

The G switch specifies the label where program execution begins, if it does not begin with the first byte of the program segment. Using the G switch causes LINK-80 to put a jump to the label at the load address.

The G switch takes the form:

```
G<label>
```

15.4.5 The Load Address (L) Switch

The load address defines the base address of the COM file generated by LINK-80. The load address is usually 100H, which is the base of the Transient Program Area (TPA) in a standard CP/M system. The L switch also sets the program origin to nnnn, unless otherwise set by the P switch.

The L switch takes the form:

```
Lnnnn
```

where nnnn is the desired load address in hexadecimal.

Note: COM files created with a load address other than 100H do not execute properly under a standard CP/M system.

15.4.6 The Memory Size (M) Switch

The M switch can be used when you are creating PRL files to indicate that the program requires additional data space for proper execution.

The M switch takes the form:

```
Mnnnn
```

where nnnn is the amount of additional data space needed in hexadecimal.

15.4.7 The No List (NL) Switch

The NL switch suppresses the listing of the Symbol Table at the console.

15.4.8 The No Recording of Symbols (NR) Switch

The NR switch suppresses the recording of the Symbol Table file on the disk.

15.4.9 The Output COM File (OC) Switch

The OC switch directs LINK-80 to produce a COM file. This is the default condition for LINK-80.

15.4.10 The Output PRL File (OP) Switch

The OP switch directs LINK-80 to produce a page-relocatable PRL file rather than a COM file. See Section 7.1 of the *MP/M II Operating System Programmer's Guide* for more information on creating PRL files.

15.4.11 The Output RSP File (OR) Switch

The OR switch outputs RSP (Resident System Process) files for execution under MP/M.

15.4.12 The Output SPR File (OS) Switch

The OS switch outputs SPR (System Page Relocatable) files for execution under MP/M.

15.4.13 The Program Origin (P) Switch

The P switch specifies the origin of the program segment. If you do not use the P switch, LINK-80 puts the program segment at the load address, which is 100H unless otherwise specified by the L switch.

The P switch takes the form:

```
Pnnnn
```

where nnnn is the program origin in hexadecimal.

15.4.14 The ? Symbol (Q) Switch

Symbols in many run-time subroutine libraries begin with a question mark to avoid conflict with user-defined symbols. LINK-80 usually suppresses listing and recording of these symbols.

The Q switch causes LINK-80 to include these symbols in the Symbol Table listed at the console and recorded on the disk.

15.4.15 The Search (S) Switch

The S switch indicates that the preceding file should be treated as a library. LINK-80 searches the file and includes only those modules containing symbols that are referenced but not defined in the modules already linked.

15.5 The $ Switch

The $ switch controls the source and destination devices. The $ switch takes the general form:

```
$td
```

where t is a type, and d is a drive specification.

LINK-80 recognizes five types:

- C — Console
- I — Intermediate
- L — Library
- O — Object
- S — Symbol

The drive specification can be a letter in the range A through P corresponding to one of sixteen logical drives, or one of the following special characters:

- X — Console
- Y — Printer
- Z — Byte bucket

15.5.1 $Cd - Console

LINK-80 usually sends messages to the console, but messages can be directed to the list device by using $CY, or they can be suppressed by using $CZ. Once $CY or $CZ has been specified, $CX can be used subsequently in the command line to redirect messages to the console device.

15.5.2 $Id - Intermediate

LINK-80 usually places the intermediate files it generates on the default drive. The $I switch allows you to specify another drive for intermediate files.

15.5.3 $Ld - Library

LINK-80 usually searches on the default drive for library files that are automatically linked because of a request item in a REL file. The $L switch instructs LINK-80 to search the specified drive for these library files.

15.5.4 $Od - Object

LINK-80 usually generates an object file on the same drive as the first REL file in the command line, unless an output file with an explicit drive is included in the command. The $O switch instructs LINK-80 to place the object file on the drive specified by the character following the $O, or to suppress the generation of an object file if the character following the $O is a Z.

15.5.5 $Sd - Symbol

LINK-80 usually generates a symbol file on the same drive as the first REL file in the command line, unless an output file with an explicit drive is included in the command. The $S switch instructs LINK-80 to place the symbol file on the drive specified by the character following the $S, or to suppress the generation of a symbol file if the character following the $S is a Z.

15.5.6 Command Line Specification

The td character pairs following a $ switch must not be separated by commas. The entire group of $ switches must be set off from any other switches by a comma. For example, the three command lines shown below are equivalent:

```
A>link part1[$sz,$od,$lb,q],part2

A>link part1[$szodlb,q],part2

A>link part1[$sz od lb],part2[q]
```

The $I switch specifies the drive to be used for intermediate files during the entire link operation, but the other $ switches can be changed in the command line. The value of a $ switch remains in effect until it is changed as LINK-80 processes the command line from left to right. This is especially useful when linking overlays. (See Section 16.) For example, the command

```
A>link root (ov1[$szcz])(ov2)(ov3)(ov4[$sacx])
```

suppresses the SYM files and console output generated when OV1, OV2 and OV3 are linked. When OV4 is linked, LINK-80 places the SYM file on drive A and sends any messages to the console device.

15.6 Creating MP/M II PRL Files

Assembly language programs often contain references to symbols in the Base Page such as BOOT, BDOS, DFCB, and DBUFF. To run properly under CP/M, or as a COM file under MP/M II, these symbols are simply defined in equates as follows:

```
boot    equ    0      ;jump to warm boot
bdos    equ    5      ;jump to bdos entry point
dfcb    equ    5ch    ;default file control block
dbuff   equ    80h    ;default i/o buffer
```

With PRL files, however, the Base Page itself can be relocated at load time, so LINK-80 must know that these symbols, while at fixed locations within the Base Page, are relocatable.

To do this, simply declare these symbols as externals in the modules in which they are referenced:

```
extrn   boot, bdos, dfcb, dbuff
```

and link in another module in which they are declared as publics and defined in equates:

```
        public  boot, bdos, dcfb, dbuff
boot    equ     0       ;jump to warm boot
bdos    equ     5       ;jump to bdos entry point
dfcb    equ     5ch     ;default file control block
dbuff   equ     80h     ;default i/o buffer
        end
```

15.7 The Request Item

Many language translators use the request item, a specific bit pattern in a REL file, to tell LINK-80 to search the appropriate run-time subroutine library file. When LINK-80 processes a library request, it first searches for an IRL file with the specified filename. If there is no IRL file, it searches for a REL file of that name. If both searches fail, then LINK-80 displays the following error message and halts.

NO FILE: filename.REL

Libraries requested in this manner appear in the Symbol Table listed at the console with a value of 'RQST'.

15.8 REL File Format

REL files contain information encoded in a bit stream, which LINK-80 interprets as follows:

- If the first bit is a 0, then the next 8 bits are loaded according to the value of the location counter.
- If the first bit is a 1, then the next 2 bits are interpreted as follows:

  00 — special link item, defined below.

  01 — program relative. The next 16 bits are loaded after being offset by the program segment origin.

  10 — data relative. The next 16 bits are loaded after being offset by the data segment origin.

  11 — common relative. The next 16 bits are loaded after being offset by the origin of the currently selected common block.
- A special item consists of:
  - A 4-bit control field that selects one of 16 special link items described below.
  - An optional value field that consists of a 2-bit address field and a 16-bit address field. The address type field is interpreted as follows:

    00 — absolute
    01 — program relative
    10 — data relative
    11 — common relative
  - An optional name field that consists of a 3-bit name count followed by the name in 8-bit ASCII characters.

The following special items are followed by a name field only.

0000 — entry symbol. The symbol indicated in the name field is defined in this module, so the module should be linked if the current file is being searched, as indicated by the S switch.

0001 — select common block. Instructs LINK-80 to use the location counter associated with the common block indicated in the name field for subsequent common relative items.

0010 — program name. The name of the relocatable module.

0011 — unused.

0100 — unused.

The following special items are followed by a value field and a name field.

0101 — define common size. The value field determines the amount of memory reserved for the common block described in the name field. The first size allocated to a given block must be larger than or equal to any subsequent definitions for that block in other modules being linked.

0110 — chain external. The value field contains the head of a chain that ends with an absolute 0. Each element of the chain is replaced with the value of the external symbol described in the name field.

0111 — define entry point. The value of the symbol in the name field is defined by the value field.

1000 — unused.

The following special items are followed by a value field only.

1001 — external plus offset. The following two bytes in the current segment must be offset by the value of the value field after all chains have been processed.

1010 — define data size. The value field contains number of bytes in the data segment of the current module.

1011 — set location counter. Set the location counter to the value determined by the value field.

1100 — chain address. The value field contains the head of a chain that ends with an absolute 0. Each element of the chain is replaced with the current value of the location counter.

1101 — define program size. The value field contains the number of bytes in the program segment of the current module.

1110 — end module. Defines the end of the current module. If the value field contains a value other than absolute 0, it is used as the start address for the program being linked. That is, the current module is the main module. The next item in the file starts at the next byte boundary.

Item 1111, end file, has no value field or name field. This item follows the end module item of the last module in the file.

15.9 IRL File Format

An IRL file consists of three parts: a header, an index, and a REL section.

The header contains 128 bytes, defined as follows:

- byte 0 — extent number of first record of REL section
- byte 1 — record number of first record of REL section
- bytes 2-127 — currently unused

The index consists of a number of entries corresponding to the entry symbol items in the REL section. The entries take the form:

e	r	b	c1	c2	. . .	cn	d

Figure 15-1. IRL File Index

where:

e = extent offset from start of REL section to start of module.

r = record offset from start of extent to start of module.

b = byte offset from start of record to start of module.

c1-cn = name of symbol.

d = end of symbol delimiter (0FEH).

The index terminates with an entry in which c1 = 0FFH. The remainder of the record containing the terminating entry is unused.

The REL section contains the relocatable object code, as described in Section 15.8.

End of Section 15

Section 16
Overlays

16.1 Introduction

You can use LINK-80 to produce a simple tree structure of overlays as shown in Figure 16-1. Currently, the Overlay Manager is part of the PL/I-80 run-time library.

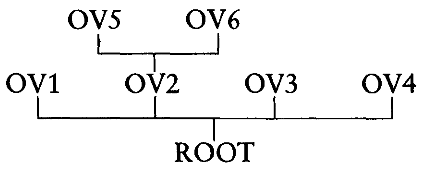

Figure 16-1. Tree-structured Overlay System

In such a system, LINK-80 produces the ROOT.COM and ROOT.SYM files, as well as an OVL file and a SYM file for each overlay specified in the command line.

The OVL file consists of a 256-byte header containing the load address and length of the overlay, followed by the absolute object code. The SYM file contains only those symbols that have not been declared in another module lower in the tree.

The origin of an overlay is the highest address, rounded to the next 128-byte boundary, of the module below it on the tree. The stack and free space for the PL/I program are located at the top of the highest overlay which is, again, rounded to the next 128-byte boundary. LINK-80 displays this address at the console on completion of the entire link process and patches it into the root module in the location '?MEMRY'.

The following restrictions must be observed when producing a system of overlays for a PL/I program using LINK-80:

- Each overlay has only one entry point. The Overlay Manager in the PL/I Run-time system assumes that this entry point is at the base (load address) of the overlay.
- No upward references are allowed from a module to an entry point in an overlay higher on the tree. The only exception is a reference to the main entry point of the overlay, as described above. Downward references to entry points in overlays lower on the tree or in the root module are allowed.
- The overlays are not relocatable, so the root module must be a COM file.
- Common blocks, EXTERNALS in PL/I, that are declared in one module cannot be initialized by a module higher in the tree. LINK-80 ignores any attempt to do so.
- Overlays can be nested to 5 levels.
- The Overlay Manager uses the default buffer located at 80H, so user programs should not depend on data stored in this buffer.

16.2 Using Overlays in PL/I Programs

There are two ways to use overlays in a PL/I program. The first method is straightforward and suffices for most applications. However, it has two restrictions. First, all overlays must be on the default drive, and second, the overlay names cannot be determined at run-time.

The second method does not have these restrictions, but its calling sequence is slightly more complicated.

16.2.1 Overlay Method 1

To use the first method, simply declare an overlay as an entry constant in the module where it is referenced. As an entry constant, it can have parameters declared in a parameter list. The overlay itself is simply a PL/I procedure or group of procedures.

For example, the following program is a root module having one overlay:

```
root: procedure options (main);
      declare ovl entry (char (15));
      put skip list ('root');
      call ov1 ('overlay 1');
      end root;
```

with the overlay OV1.PLI defined as follows:

```
ov1: procedure (c);
     declare c char (15);
     put skip list (c);
     end ov1;
```

Note: when passing parameters to an overlay, you must ensure that the number and type of the parameters are the same in the calling program and the overlay itself.

To link these two programs into an overlay system, use the command:

```
A>LINK ROOT (OV1)
```

This causes LINK-80 to produce four files:

At execution time, ROOT.COM first displays the message 'root' at the console. The 'call ov1' statement then transfers control to the Overlay Manager.

The Overlay Manager loads the file OV1.OVL from the default drive at the proper location above ROOT.COM and transfers control to it, passing the CHARACTER(15) parameter in the usual manner.

The overlay then executes, displaying the message 'overlay 1' at the console. It then returns directly to the statement following the 'call ov1' in ROOT.PLI, and execution continues from that point.

If the Overlay Manager determines that the requested overlay is already in memory, then it does not reload the overlay before transferring control to it.

There are several important points to keep in mind regarding overlay method 1:

- The name associated with the overlay in the call and entry statements is the actual name of the OVL file loaded by the Overlay Manager, so the two names must agree. Because PL/I truncates symbol names to 6 characters in the REL file, the names of the OVL files must be limited to 6 characters.
- The name of the entry point to an overlay (the name of the procedure) need not agree with the name used in the calling sequence. The same name should be used to avoid confusion.
- The Overlay Manager loads overlays only from the drive that was the default drive when the root module began execution. The Overlay Manager disregards any changes in the default drive that occur after the root module begins execution.
- The names of the overlays are fixed. This means the source program must be edited, recompiled, and relinked to change the names of the overlays.
- No nonstandard PL/I statements are needed. Thus the program is transportable to other systems.

16.2.2 Overlay Method 2

In some applications, it is useful to have greater flexibility with overlays, such as the ability to load overlays from different drives, or the ability to determine the name of an overlay at run-time, perhaps from the keyboard or from a disk file.

To do this, a PL/I program must declare an explicit entry point into the Overlay Manager as follows:

```
declare ?ovlay entry (char (10), fixed (1));
```

The first parameter is a character string specifying the name of the overlay to load and an optional drive name in the standard CP/M format, d:filename.

The second parameter is the Load Flag. If the Load Flag is 1, the Overlay Manager loads the specified overlay whether or not it is already in memory. If the Load Flag is 0, then the Overlay Manager loads the overlay only if it is not already in memory.

The 'call ?ovlay' statement signals the Overlay Manager to load the requested overlay, if needed. The Overlay Manager returns to the calling program, which must then perform a dummy call to execute the overlay just processed by the Overlay Manager. This allows a parameter list to be passed to the overlay.

Using this method, the example shown in the first method above appears as follows:

```
root: procedure options (main);
      declare ?ovlay entry (char (10), fixed (1));
      declare dummy entry (char (15));
      declare name char (10);
      put skip list ('root');
      name = 'OV1';
      call ?ovlay (name, 0);
      call dummy ('overlay 1');
      end root;
```

The file OV1.PLI is the same as before.

At run-time, the Overlay Manager loads OV1.OVL from the default drive because that is the current value of the variable 'name', and then returns to the calling program, in this case, 'root.'

At this point, the argument 'overlay 1' is set up according to the PL/I parameter passing conventions. The 'call dummy' statement transfers control to the Overlay Manager, which in turn transfers control to the base address of the overlay the name of which it just processed. When OV1 finishes execution, it returns to the statement following the call dummy statement.

Note that in this example, name is set to 'OV1' in an assignment statement. However, the overlay name can also be supplied as a character string from some other source, such as the console keyboard.

Observe these important points when using overlay method 2:

- A drive name can be specified, so the Overlay Manager can load overlays from drives other than the default drive. If no drive is specified, the Overlay Manager uses the default drive as described in Method 1.
- The name of the overlay can be up to 8 characters in length because it is specified in the character string and not by the entry symbol.
- If there are any parameters in the dummy call following the call ?ovlay, they must agree in number and type with the parameters in the procedure declaration in the overlay.

16.3 Specifying Overlays in the Command Line

The syntax for specifying overlays is similar to that for linking without overlays, except that each overlay specification is enclosed in parentheses.

An overlay specification can take one of the following forms:

```
A>LINK ROOT(OV1)

A>LINK ROOT(OV1,PART2,PART3)

A>LINK ROOT(OV1=PART1,PART2,PART3)
```

The first command produces the file OV1.OVL from a file OV1.REL. The second command produces the file OV1.OVL from OV1.REL, PART2.REL, and PART3.REL. The third command produces the file OV1.OVL from PART1.REL, PART2.REL, and PART3.REL.

Note that a left parenthesis, indicating the start of a new overlay specification, also indicates the end of the group preceding it. Thus the following command line is invalid, and LINK-80 flags it as an error:

```
A>LINK ROOT(OV1),MOREROOT
```

All files to be included at any point on the tree must appear together, without any intervening overlay specifications. Thus the following command is valid:

```
A>LINK ROOT,MOREROOT(OV1)
```

Any filename in the command line can be followed by a number of LINK-80 switches. The overlay specifications are not set off from the root module or from each other with commas. Spaces can be used to improve readability.

To nest overlays, they must be specified in the command line with nested parentheses. For example, the following command line can link the overlay system shown in Figure 16-1:

```
A>LINK ROOT (OV1) (OV2 (OV5) (OV6)) (OV3) (OV4)
```

16.4 Sample LINK-80 Execution

Listing 16-1 shows the console output from a LINK-80 operation. Note that OV1 is flagged as an undefined symbol. LINK-80 indicates that OV1 has not been defined in the current module and assumes it is either the name of an overlay or a dummy entry point to an overlay.

When linking overlays, each entry variable that refers to an overlay, by actual name or a dummy entry, appears as an undefined symbol. No symbols other than these actual or dummy overlay entry points should be undefined.

Listing 16-2 shows the console output when executing the resulting COM file.

```
A>link root(ov1)
LINK 1.3

PLILIB RQST   ROOT 0100   /SYSIN/  1A15   /SYSPRI/ 1A3A

UNDEFINED SYMBOLS:

OV1

ABSOLUTE      0000
CODE SIZE     18BC (0100-19BB)
DATA SIZE     02A9 (1A90-1D38)
COMMON SIZE   00D4 (19BC-1A8F)
USE FACTOR      4E




LINKING OV1.OVL

PLILIB   RQST

ABSOLUTE      0000
CODE SIZE     0024 (1D80-1DA3)
DATA SIZE     0002 (1DA4-1DA5)
COMMON SIZE   0000
USE FACTOR      09

MODULE TOP    1E00
```

Listing 16-1. LINK-80 Console Interaction

```
A>root
root
overlay 1
End of Execution
A>
```

Listing 16-2. Console Interaction with ROOT

Appendix D
Overlay Manager Run-time Error Messages

At run-time, the Overlay Manager can display certain error messages. These messages and a brief explanation of their causes are shown in Table D-1.

Table D-1. Run-time Error Messages

Error	*Cause*
`ERROR (8) OVERLAY, NO FILE d:filename.OVL`	The Overlay Manager cannot find the indicated file.
`ERROR (9) OVERLAY, DRIVE d:filename.OVL`	An invalid drive code was passed as a parameter to ?ovlay.
`ERROR (10) OVERLAY, SIZE d:filename.OVL`	The indicated overlay would overwrite the PL/I stack and/or free space if it were loaded.
`ERROR (11) OVERLAY, NESTING d:filename.OVL`	Loading the indicated overlay would exceed the maximum nesting depth.
`ERROR (12) OVERLAY, READ d:filename.OVL`	Disk read error during overlay load, probably caused by premature EOF.

End of Appendix D

Appendix E
LIB-80 Error Messages

During the course of operation, LIB-80 can display error messages. These error messages and a brief explanation of their causes are given in Table E-1.

Table E-1. LIB-80 Error Messages

Error	*Cause*
`CANNOT CLOSE:`	LIB-80 cannot close the output file. The disk might be write-protected.
`DIRECTORY FULL:`	There is no directory space for the output file.
`DISK READ ERROR:`	LIB-80 cannot read the file properly.
`DISK WRITE ERROR:`	LIB-80 cannot write to the file properly, probably due to a full disk.
`FILE NAME ERROR:`	The form of a source filename is invalid.
`NO FILE:`	LIB-80 cannot find the indicated file.
`NO MODULE:`	LIB-80 cannot find the indicated module.
`SYNTAX ERROR:`	The LIB-80 command line is not properly formed.

End of Appendix E

Appendix F
8080 CPU Instructions

Table F-1. 8080 CPU Instructions

OP Code	*MNEMONIC*		*OP Code*	*MNEMONIC*		*OP Code*	*MNEMONIC*	
00	NOP		1D	DCR	E	3A	LDA	Adr
01	LXI	B,D16	1E	MVI	E,D8	3B	DCX	SP
02	STAX	B	1F	RAR		3C	INR	A
03	INX	B	20	---		3D	DCR	A
04	INR	B	21	LXI	H,D16	3E	MVI	A,D8
05	DCR	B	22	SHLD	Adr	3F	CMC	
06	MVI	B,D8	23	INX	H	40	MOV	B,B
07	RLC		24	INR	H	41	MOV	B,C
08	---		25	DCR	H	42	MOV	B,D
09	DAD	B	26	MVI	H,D8	43	MOV	B,E
0A	LDAX	B	27	DAA		44	MOV	B,H
0B	DCX	B	28	---		45	MOV	B,L
0C	INR	C	29	DAD	H	46	MOV	B,M
0D	DCR	C	2A	LHLD	Adr	47	MOV	B,A
0E	MVI	C,D8	2B	DCX	H	48	MOV	C,B
0F	RRC		2C	INR	L	49	MOV	C,C
10	---		2D	DCR	L	4A	MOV	C,D
11	LXI	D,D16	2E	MVI	L,D8	4B	MOV	C,E
12	STAX	D	2F	CMA		4C	MOV	C,H
13	INX	D	30	---		4D	MOV	C,L
14	INR	D	31	LXI	SP,D16	4E	MOV	C,M
15	DCR	D	32	STA	Adr	4F	MOV	C,A
16	MVI	D,D8	33	INX	SP	50	MOV	D,B
17	RAL		34	INR	M	51	MOV	D,C
18	---		35	DCR	M	52	MOV	D,D
19	DAD	D	36	MVI	M,D8	53	MOV	D,E
1A	LDAX	D	37	STC		54	MOV	D,H
1B	DCX	D	38	---		55	MOV	D,L
1C	INR	E	39	DAD	SP	56	MOV	D,M

Table F-1. (continued)

OP Code	*MNEMONIC*		*OP Code*	*MNEMONIC*		*OP Code*	*MNEMONIC*	
57	MOV	D,A	7B	MOV	A,E	9F	SBB	A
58	MOV	E,B	7C	MOV	A,H	A0	ANA	B
59	MOV	E,C	7D	MOV	A,L	A1	ANA	C
5A	MOV	E,D	7E	MOV	A,M	A2	ANA	D
5B	MOV	E,E	7F	MOV	A,A	A3	ANA	E
5C	MOV	E,H	80	ADD	B	A4	ANA	H
5D	MOV	E,L	81	ADD	C	A5	ANA	L
5E	MOV	E,M	82	ADD	D	A6	ANA	M
5F	MOV	E,A	83	ADD	E	A7	ANA	A
60	MOV	H,B	84	ADD	H	A8	XRA	B
61	MOV	H,C	85	ADD	L	A9	XRA	C
62	MOV	H,D	86	ADD	M	AA	XRA	D
63	MOV	H,E	87	ADD	A	AB	XRA	E
64	MOV	H,H	88	ADC	B	AC	XRA	H
65	MOV	H,L	89	ADC	C	AD	XRA	L
66	MOV	H,M	8A	ADC	D	AE	XRA	M
67	MOV	H,A	8B	ADC	E	AF	XRA	A
68	MOV	L,B	8C	ADC	H	B0	ORA	B
69	MOV	L,C	8D	ADC	L	B1	ORA	C
6A	MOV	L,D	8E	ADC	M	B2	ORA	D
6B	MOV	L,E	8F	ADC	A	B3	ORA	E
6C	MOV	L,H	90	SUB	B	B4	ORA	H
6D	MOV	L,L	91	SUB	C	B5	ORA	L
6E	MOV	L,M	92	SUB	D	B6	ORA	M
6F	MOV	L,A	93	SUB	E	B7	ORA	A
70	MOV	M,B	94	SUB	H	B8	CMP	B
71	MOV	M,C	95	SUB	L	B9	CMP	C
72	MOV	M,D	96	SUB	M	BA	CMP	D
73	MOV	M,E	97	SUB	A	BB	CMP	E
74	MOV	M,H	98	SBB	B	BC	CMP	H
75	MOV	M,L	99	SBB	C	BD	CMP	L
76	HLT		9A	SBB	D	BE	CMP	M
77	MOV	M,A	9B	SBB	E	BF	CMP	A
78	MOV	A,B	9C	SBB	H	C0	RNZ	
79	MOV	A,C	9D	SBB	L	C1	POP	B
7A	MOV	A,D	9E	SBB	M	C2	JNZ	Adr

Table F-1. (continued)

OP Code	*MNEMONIC*	*OP Code*	*MNEMONIC*	*OP Code*	*MNEMONIC*
C3	JMP Adr	D7	RST 2	EB	XCHG
C4	CNZ Adr	D8	RC	EC	CPE Adr
C5	PUSH B	D9	---	ED	---
C6	ADI D8	DA	JC Adr	EE	XRI D8
C7	RST 0	DB	IN D8	EF	RST 5
C8	RZ	DC	CC Adr	F0	RP
C9	RET Adr	DD	---	F1	POP PSW
CA	JZ	DE	SBI D8	F2	JP Adr
CB	---	DF	RST 3	F3	DI
CC	CZ Adr	E0	RPO	F4	CP Adr
CD	CALL Adr	E1	POP H	F5	PUSH PSW
CE	ACI D8	E2	JPO Adr	F6	ORI D8
CF	RST 1	E3	XTHL	F7	RST 6
D0	RNC	E4	CPO Adr	F8	RM
D1	POP D	E5	PUSH H	F9	SPHL
D2	JNC Adr	E6	ANI D8	FA	JM Adr
D3	OUT D8	E7	RST 4	FB	EI
D4	CNC Adr	E8	RPE	FC	CM Adr
D5	PUSH D	E9	PCHL	FD	---
D6	SUI D8	EA	JPE Adr	FE	CPI D8
				FF	RST 7

D8 = constant or logical/arithmetic expression that evaluates to an 8 bit quantity.
Adr = 16-bit address.
D16 = constant or logical/arithmetic expression that evaluates to a 16 bit data quantity.
Reproduced with permission from Intel Corporation, Santa Clara, CA.

End of Appendix F

Index

Reader Comment Form

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date ____________ Manual Title ____________________________ Edition ___________

1. What sections of this manual are especially helpful?

2. What suggestions do you have for improving this manual? What information is missing or incomplete? Where are examples needed?

3. Did you find errors in this manual? (Specify section and page number.)

COMMENTS AND SUGGESTIONS BECOME THE PROPERTY OF DIGITAL RESEARCH

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH™
P.O. Box 579
Pacific Grove, California
93950

Attn: Publication Production