Disclaimer

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

Table of Contents

Foreword

The CP/M macro assembler, called MAC, reads assembly language statements from a diskette file and produces a "hex" format object file on the diskette suitable for processing in the CP/M environment, and is upward compatible from the standard CP/M non-macro assembler (see the Digital Research manual entitled "CP/M Assembler (ASM) User's Guide"). The facilities of MAC include assembly of Intel 8080 microcomputer mnemonics, along with assembly-time expressions, conditional assembly, page formatting features, and a powerful macro processor which is compatible with the standard Intel definition (MAC implements the mid-1977 revision of Intel's definition, which is not compatible with previous versions). In addition, MAC will accept most programs prepared for the Processor Technology Software #1 assembler, normally requiring only minor modifications.

The macro assembler is supplied on a CP/M non-system diskette, along with a number of standard library files. The macro assembler requires approximately 12K of machine code and table space, along with an additional 2.5K of I/O buffer space. Since the BDOS portion of CP/M is coresident with MAC, the minumum usable memory size for MAC is approximately 20K. Any additional memory adds to the available symbol table area, thus allowing larger programs to be assembled.

Upon receiving the MAC diskette, you should follow the steps given below

(a) place the MAC diskette into drive B, with a CP/M system diskette in drive A. Copy the MAC.COM to drive A from drive B using PIP (see the CP/M Features and Facilities Guide for PIP operation).

(b) Copy the SAMPLE.ASM program from drive B to drive A using the PIP program.

(c) Remove the MAC diskette from drive B, and retain the diskette for future backup (there are a number of "LIB" files which may be useful at a later time).

(d) Type "MAC SAMPLE" to execute the macro assembler (see Figure 1). The macro assembler should load and print the signon message. Upon completion, the final program address is printed, followed by the "use factor" which indicates that the assembly is complete.

(e) Type the "SAMPLE.PRN" and "SAMPLE.SYM" files, and compare with Figure 1 to ensure that the assembler is executing properly, thus completing the MAC test.

This manual is organized in three major sections. The first section describes the simple assembler facilities of MAC which involve 8080 mnemonic forms, expressions, and conditional assembly, similar to the discussion found in the ASM User's Guide. If you are familiar with ASM, you may wish to skip over the first section, and start reading Section 6. The second portion of this manual, beginning with Section 6, describes the MAC macro facilities in some detail. Again, if you are familiar with macros, you may wish to briefly skim these sections, and refer primarily to the examples to get the "flavor" of the MAC facility. Section 10 discusses macro applications, where common macro forms and programming practices are discussed. Again, it is useful to skim the examples and refer back to the explanations for detailed discussions of each program.

1. MACRO ASSEMBLER OPERATION UNDER CP/M

The user must first prepare a source program containing assembly language statements using the ED program under CP/M (see the Digital Research manual "CP/M Context Editor (ED) User's Guide"), and then submit the assembly language file for processing under MAC. Although the user may specify certain options (described under "Assembly Parameters"), the usual invocation of MAC is simply

MAC filename

where "filename" corresponds to the assembly language file which was prepared using ED, with an assumed (and unspecified) file type of "ASM." Upon completion of the translation process, MAC leaves a file called "filename.HEX" containing the machine code in Intel hexadecimal format which can subsequently be loaded (see the LOAD command in the "CP/M Features and Facilities" manual), or tested under the CP/M debugger (see the "CP/M Dynamic Debugging Tool (DDT) User's Guide"). In addition to the HEX file, MAC also prepares a file named "filename.PRN" which contains an annotated source listing, along with a file called "filename.SYM" which contains a sorted list of symbols defined in the program.

Figure 1 provides an example of the output from MAC for a sample assembly language program which is stored on the diskette under the name SAMPLE.ASM. The macro assembler is executed by typing "MAC SAMPLE" followed by a carriage return. Upon completion, the PRN, SYM, and HEX files will appear as shown in the figure. The assembler listing file (PRN) includes a 16 column annotation at the left which shows the values of literals, machine code addresses, and generated machine code. Note that an equal sign (=) is used to denote literal values (see the EQU directive) to avoid confusion with machine code addresses. In all cases, output files contain tab characters (ASCII control-I) wherever possible in order to conserve diskette space. Tab positions are assumed to be placed at every eight columns of the output line.

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
```

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 (SAMPLE.SYM)

```
0005 BDOS       0002 WCHAR
```

Assembler "Hex" Output file (SAMPLE.HEX)

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

Figure 1. Sample ASM, PRN, SYM, and HEX Files from MAC.

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 may be present in a particular statement. Each assembly language statement is terminated by a carriage return and line feed (the line feed is inserted automatically by the ED program when the file is prepared), or with the character "!" which is treated as an end of line by the assembler. Thus, multiple assembly language statements can be written on the same physical line if separated by exclamation marks.

Statement elements are delimited by a sequence of one or more blank or tab characters. Tab characters are preferred since the program element alignment is automatically maintained in the output line at every eighth column, without requiring extra blanks in the file. This not only conserves source file space, but also reduces the listing file size since the tab characters are included in the PRN file. The tab characters are not actually 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, which is allowed on any source line in case the program is prepared with a line editor which uses line numbers at the beginning of each statement. In all cases, the optional line# is ignored by the assembler.

The label field takes the form

identifier or identifier :

and is optional, except where noted in particular statement types. The identifier is a sequence of alphanumeric characters (alphabetics, question marks, commercial atsigns, and numbers) where the first character is alphabetic (including "?" and "@"). Identifiers can be freely used by the programmer to label elements such as program steps and assembler directives, but cannot exceed 16 characters in length. All characters are significant in an identifier, except for the embedded dollar sign ($) which can be used to improve readability of the name. Further, all lower case alphabetics are treated as if they are upper case in an identifier. Note that the ":" following the identifier in a label is optional (to maintain compatibility between the Intel and Processor Technology versions). Thus, the following are all valid instances of labels

```
x                   xy              long$name
x?                  xy1:            longer$named$data
xlx2                @123:           ??@@abcDEF
Gamma               @GAMMA          ?ARE$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 below, while the macro calls are delayed for later discussion.

The operand field of the statement, in general, contains an expression formed from constant and label operands, with arithmetic, logical, and relational operations upon these operands. Again, the complete details of properly formed expressions are given in sections which follow.

The comment field is denoted by a leading ";" character, and contains arbitrary characters until the next real or logical end of line. These character are read, listed, and otherwise ignored in the assembly process. In order to maintain compatibility with other assemblers, MAC also treats statements which begin with a "*" in the first position as comment lines.

The assembly language program is thus a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

3. FORMING THE OPERAND

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined into properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression produces a 16-bit value during the assembly. Further, 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. The restrictions on the expression significance are given with the individual instructions.

3.1. Labels.

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g., a MOV instruction or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label is given the value which results from evaluating the operand field. In the case of a macro definition, the label is given a text value (i.e., a sequence of ASCII characters) which 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 (non-macro) label appears in the operand field, its 16-bit value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction. When a macro identifier appears in the operation field of the statement, the text which is stored as the value of the macro name is substituted in place of the name. In this case, the operand field of the statement contains "actual parameters" which are substituted for "dummy parameters" in the body of the macro definition. The exact mechanisms for definition, invocation, and substitution of macro text are given in later sections.

3.2. Numeric Constants.

A numeric constant is a 16-bit value in one of several number bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. 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 since the letter O is easily confused with the digit 0. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is, binary constants must be composed of 0 and 1 digits, octal constants can contain digits in the range 0 - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A through H (corresponding to the decimal numbers 10 through 15). Note, however, that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading 0 will always suffice). A constant composed in this manner will produce a binary number which can be contained within a 16-bit counter, truncated on the right by the assembler. Similar to identifiers, imbedded "$" symbols are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants:

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

3.3. Reserved Words.

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8080 registers are given below which, when encountered, produce the corresponding value.

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

Again, lower case names have the same values as their upper case equivalents. Machine instructions can also be used in the operand field, and result in their internal codes. In the case of instructions which require operands, where the specific operand becomes 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 zeroes in the optional fields. For example, the statement

```
LXI H,MOV
```

assembles an LXI H instruction with an operand equal to 40H (which is the value of the MOV instruction with zeroes as operands).

When the symbol "$" appears in the operand field (not imbedded within identifiers and numbers), its value becomes the address of the beginning of the current instruction. For example, the two statements

```
X:    JMP X
```

and

```
      JMP $
```

both 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 "assembly parameters").

3.4. String Constants.

String constants represent sequences of graphic ASCII characters, and are represented by enclosing the characters within apostrophe symbols ('). All strings must be fully contained within the current physical line, with the "!" character within strings treated as an ordinary string character. Each individual string must not exceed 64 characters in length, otherwise an error is reported. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes "), which become a single apostrophe when read by the assembler.

Note that particular operation codes may require that the string length be no longer than one or two characters. The LXI instruction, for example, will accept a character string operand of one or two characters, while the CPI instruction will accept only a one character string. The DB instruction, however, allows strings of length zero through 64 characters in its list of operands. In the case of single character strings, the value becomes the 8-bit Ascii code for the character (without case translation), while 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. The string constant 'A' for example, is equivalent to 41H, while the two character string 'AB' produces the 16-bit value 4142H. The following strings are valid in various MAC statements:

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

There is one special case which must be considered inside string constants. As discussed in later sections, the character "&" can be used to cause evaluation of dummy arguments within macro expansions when they occur inside of string quotes. The exact details of the substitution process will be given in the discussion of macro definition and call statements.

3.5. Arithmetic, Logical, and Relational Operators.

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expression. The operators recognized by MAC in the operand field are given below. In general, the letters a and b represent operands which are treated as 16-bit unsigned quantities in the range 0-65535. All arithmetic operators (+, -, *, /, MOD, SHL, and SHR) produce a 16-bit unsigned arithmetic result, the relational operators (EQ, LT, LE, GT, GE, and NE) produce a true (0FFFFH) or false (0000H) 16-bit result, and the logical operators (NOT, AND, OR, and XOR) operate bit-by-bit on their operand(s) producing a 16-bit result of 16 individual bit operations. The HIGH and LOW functions alway produce a 16-bit result with a high order byte which is zero.

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 magnitude multiplication of a by b
a/b is the unsigned magnitude 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 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 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)

In general, all computations are performed 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, and 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, 8-bit values which would normally be considered "negative" 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 which has all 1's in the high order byte, and the "sign bit" is set, then the high order byte is truncated and no error is reported. This particular condition arises when a negative sign is placed in front of a constant. The value -2, for example, is defined (and computed) as 0-2 which produces the 16-bit value 0FFFEH, where the high order byte (0FFH) contains extended sign bits which are all 1's, while the low order byte (0FEH) has the sign bit set. Thus, the following instructions do not produce value errors in MAC:

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

while the following instructions do 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, and must be the last operator in the operand field, preceding only a single operand. The use and effects of the NUL operator are delayed until the discussion of macros.

Expressions can generally be formed from simple operands such as labels, numeric constants, string constants, and machine operation codes, or 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.

As a convenience to the programmer, MAC assumes that operators have a relative precedence of application which allows expressions to be written without nested levels of parentheses. The resulting expression has assumed parentheses which are defined by this relative precedence. The order of application of operators in

unparenthesized expressions is listed below. Operators listed first have highest precedence, and 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 LOW
```

Thus, the expressions shown below 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 be used to override the assumed parentheses, and thus the last expression above could 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 which results from inserting the assumed parentheses is well-formed.

As a notational convenience, the following are equivalent:

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

4. ASSEMBLER DIRECTIVES

Assembler directives are used to 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 which appears in the operation field of the statement. The acceptable pseudo operations are given below.

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 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 pages titles and options

In addition to those listed above, there are several pseudo operations which are used in conjunction with the macro processing facilities. Specifically, the MACRO, EXITM, ENDM, REPT, IRPC, IRP, LOCAL, and MACLIB operations are reserved words, and are fully described in separate sections which deal with macro processing. The non-macro 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 (i.e., an identifier followed by an optional ":"), and "expression" is a 16-bit expression consisting of operands which are defined previous to 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 particular program, and there are no checks to ensure that the programmer is not redefining overlapping memory areas. Note that most programs written for CP/M begin with an "ORG 100H" statement which causes machine code generation to begin at the base of the CP/M transient program area.

If a label is specified in the ORG statement, then the label takes on the value given by the expression, which 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, but 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 which results from the assembly. Thus most CP/M assembly language programs end with the statement

```
END   100H
```

resulting in the default starting address of 100H, which is the beginning of the transient program area.

4.3. The EQU Directive.

The EQU (equate) statement is used to name synonyms for particular numeric values. The form is

```
label   EQU   expression
```

where the label must be present, and 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 which describes the value in a more human-oriented manner. Further, this name can be used throughout the program as a parameter for certain functions. Suppose, for example, that data received from a Teletype appears on a particular input port, and data is sent to the Teletype through the next output port in sequence. The series of equate statements that could be used to define these ports for a particular hardware environment are 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 which 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 if the absolute I/O port addresses had been used. 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 be used to assign different values to a name at different parts of the program. In particular, the SET statement gives the label a value which 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 which are to be included or excluded during the assembly process. The IF and ENDIF statements alone can be used to bound a group of statements to be conditionally assembled, as shown below:

```
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 produces a non-zero value then statement#1 through statement#n are assembled. If the expression evaluates to a zero value then the statement are listed but not assembled.

Conditional assembly is often used to write a single "generic" program which 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. Figures 2a and 2b give an example of such a program. Assume that a console device (either a Teletype or CRT) is connected to an 8080 microcomputer through I/O ports. Due to the electronic enviroment, 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. A single program is shown which, when the condition is properly set, produces a program which operates with either a Teletype (TTY is TRUE), or with a CRT (TTY is FALSE), but not both. Figure 2a shows an assembly for the Teletype environment, while Figure 2b shows the assembly for a CRT-based system. Note that the leftmost 16 columns are left blank by the assembler when statements are skipped due to a false condition.

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

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

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

 FFFF =          TRUE     EQU      0FFFFH  ;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
```

Figure 2a. Conditional Assembly with TTY "True."

```
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
                          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
```

Figure 2b. Conditional Assembly with TTY "False."

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

If the expression produces a non-zero (true) value, then statements 1 through n are assembled, as before. In this case, however, statements n+1 through m are skipped in the assembly process. When the expression produces a zero value (false), statements 1 through n are skipped, while statements n+1 through m are assembled. As an example, the conditional assembly shown in Figure 2 could be rewritten as shown in Figure 3a.

Properly balanced IF's, ELSE's, and ENDIF's 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
```

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

Conditional assembly of this sort can be nested up to eight levels (i.e., there can be up to eight pending IF's or ELSE's with unresolved ENDIF's at any point in the assembly), but usually becomes unreadable after two or three levels of nesting. The nesting level restriction also holds, however, for pending IF's and ELSE's during macro evaluation. Nesting level overflow will produce an error during assembly.

4.6. The DB Directive.

The DB directive allows the programmer to define initialized storage areas in single precision (byte) format. The statement form is

```
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
```

Figure 3a. Conditional Assembly Using "ELSE" for Alternate.

```
FFFF =          TRUE    EQU     0FFFFH          ;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
0006                    END
```

Figure 3b. Sample Program using Nested IF, ELSE, and ENDIF

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

where the label is optional, and e#1 through e#n are either expressions which produce 8-bit values (the high order eight bits are zero, or the high order nine sign bits are one's), or are ASCII strings of length no greater than 64 characters each. There is no practical restriction on the number of expressions included on a single source line. The expressions are evaluated and palced sequentially into the machine code following the last program address generated by the assembler. String characters are similarly placed into memory starting with the first character and ending with the last character. Strings of length greater than two characters cannot be used as operands in more complicated expressions (i.e., 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. Examples of valid DB statements are:

```
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 is:

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

where the label is optional, and e#1 through e#n are expressions which produce 16-bit values. Note that Ascii strings of length one or two characters are allowed, but strings longer than 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 DW statements are examples of properly formed 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 is used to reserve 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 sequence:

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

4.9. The PAGE and TITLE Directives.

The PAGE and TITLE pseudo operations give the programmer control over the output formatting which is sent to the PRN file (or directly to the printer device). The forms for the PAGE statement are:

```
PAGE
```

and

```
PAGE expression
```

If the PAGE statement stands alone, as in the first case above, the output page is ejected to the top of form (i.e., an ASCII control-L (form feed) is sent to the output file). The form feed is sent after the statement with PAGE has been printed, thus 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 printer page.

The second form of the PAGE command is used to specify the output page size. In this case, the expression which follows 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, and the print file is simply a continuous sequence of annotated output lines. If the expression is non-zero, then the page size is set to the value of the expression, and 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, thus 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, which does not exceed 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, along with the blank line which follows the title, are not included in the line count for the page. Normally, no more than one TITLE statement is included in a particular program. Similarly, no more than one PAGE statement with the expression option is normally included.

If a TITLE statement is included, and the symbol table is being appended to the PRN file (see "assembly parameters"), then the SYM file also contains the specified 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.

Figure 4 demonstrates the various pseudo operations available in MAC. The sample program, called "typer," is intended to operate in the CP/M environment by performing the simple function of selecting one of three messages for output at the console. This program is created using the ED program, then assembled using MAC, and then placed into "COM" file format using the CP/M LOAD function. Given that these steps have been accomplished, typer is executed 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 "signon" message, which would appear as:

```
'typer' version 1.0
```

The program then retrieves the first character typed at the console following the command "typer" which should be one of the letters 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.

Note that the TITLE and PAGE statements are used to produce a title at the beginning of each page (form feeds were necessarily suppressed here), with a page size of 20 lines, excluding the title lines. A number of EQU statements are used at the beginning to improve readability of the program. Note that the exclaim symbol (!) is used throughout the program to allow 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, which is only necessary if a starting address is specified. The END statement is often included, however, to maintain compatibility with other assemblers.

The DB statements labelled by SIGNON contain simple strings of characters, as well as expressions which 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.

```
                        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
```

Figure 4. "Typer" Program Listing (Part A).

```
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:
```

Figure 4. "Typer" Program Lisitng (Part B).

5. OPERATION CODES

Operation codes, found in the operation field of the statement, form the principal components of assembly language programs. In general, MAC accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly language Programming Manual." 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 in order to be complete, although it is understood that the Intel documents should be referenced for exact operator details. In the discussion which follows, the operation codes are placed into categories for discussion purposes, followed by a sample assembly which shows the hexadecimal codes produced for each operation. The following notation is used throughout the discussion:

e3 represents a 3-bit value in the range 0-7, which 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 (recall that 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 themselves 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. These cases will be noted as they are encountered.

5.1. Jumps, Calls, and Returns.

The jump, call and return instructions allow several different forms, as shown in Figure 5. In some cases, the condition flags are tested to determine whether or not the jump, call, or return is to be taken. The forms are shown below.

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

Thre return instructions are:

RET	RNZ	RZ
RNC	RC	RPO
RPE	RP	RM

The restart instruction takes the form:

```
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
```

Figure 5. Assembly showing Jumps, Calls, Returns, and Restarts.

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

Figure 5 shows the hexadecimal codes for each instruction, along with a short comment on each line which describes the function of the instruction.

5.2. Immediate Operand Instructions.

Several instructions are available which load single or double precision registers or single precision memory cells with constant values, along with instructions which 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 produces 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 in always performed upon 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.

Figure 6 shows the use of the accumulator immediate operations in an assembly language program, along with a short comment describing the use of each instruction.

5.3. Increment and Decrement Instructions.

Instructions are provided in the 8080 repetoire 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 one of the registers A, B, C, D, H, L, or M (corresponding to the byte value at the memory location addressed by HL). The double precision instructions are:

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

                         TITLE   'IMMEDIATE OPERAND INSTRUCTIONS'
                ;
                ;        MVI USES A REGISTER (3BIT) 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
```

Figure 6. Assembly using Immediate Operand Instructions.

```
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
```

Figure 7. Assembly containing Increment and Decrement Instructions.

INX e3 DCX e3

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

Figure 7 shows a sample assembly language program which uses both single and double precision increment and decrement operations.

5.4. Data Movement Instructions.

A number of 8080 instructions are placed in this category which move data from memory to the CPU and from the CPU to memory. A number of register to register move operations are also included. The single precision "move register" instruction takes the form:

MOV e3,e3'

where e3 and e3' are expressions which each produce one of the single precision registers A, B, C, D, E, H, L, or M (corresponding to the memory location addressed by HL). In all cases, the register named by e3 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 which must produce one of the double precision register pairs B or D. The 8-bit value in register A is either loaded (LDAX) or stored (STAX) from/to the memory location addressed by the specified register pair.

The load and store direct instructions operate either upon the A register for single precision operations, or upon the HL register pair for double precision operations, and take the forms:

LHLD e16 SHLD e16 LDA e16 STA e16

where e16 is an expression 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 found in this category, even though they receive and send their data to the electronic environment which is external to the 8080 processor. The input instruction reads data to the A register, while the output instruction sends data from the A register. In both cases, the data port is

```
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
```

Figure 8. Assembly Using Various Register/Memory Moves.

given by the data value which follows the instruction:

IN e8 OUT e8

Various instructions are a part of the instruction set which transfer double precision values between registers and the stack. These instructions are:

XTHL	PCHL	SPHL	XCHG

Figure 8 lists these instructions in an assembly language program, along with a short comment on the use of each instruction.

5.5. Arithmetic Logic Unit Operations.

A number of instructions are included in the 8080 set which operate between the accumulator and single precision registers, including operations upon the A register and carry flag. The accumlator/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 actual function of each instruction is listed in the comment line shown in Figure 9.

The last instruction of this group is the double precision add instruction which performs a 16-bit addition of a register pair (B, D, H, or SP) into the 16-bit value in the HL register pair, producing the 16-bit (unsigned) sum of the two values which 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 categorized as control instructions, and take the forms:

HLT	DI	EI	NOP

and are used to stop the processor (HLT), enable the interrupt system (EI), disable the interrupt system (DI), or perform a "no-operation" (NOP).

```
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
```

Figure 9. Assembly Showing ALU Operations.

6. AN INTRODUCTION TO MACRO FACILITIES

The fundamental difference between the Digital Research "ASM" and "MAC" assemblers is that ASM provides only the fundamental facilities for assembling 8080 operation codes, while MAC includes a powerful macro processing facility. In particular, 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 invocations (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. In addition, the MACLIB (macro library) feature allows the programmer to define a particular set of macros, equates, and sets for automatic inclusion in a program. A macro library can contain an instruction set for another central processor, for example, which is not directly supported by the MAC built-in mnemonics. The macro library may also include general purpose input/output macros which are used in various programs which operate in the CP/M environment to perform peripheral or diskette I/O functions.

IRPC, IRP, and REPT pseudo operations provide repetition of source statements under control of a count or list of characters or items to be substituted each time the statements are re-read by the assembler. 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 file type is changed in each statement.

In order to illustrate the power of a macro facility, consider the macro library shown in Figure 10, which is assumed to reside in a diskette file called "MSGLIB.LIB." This macro library contains macro definitions which have standard instruction sequences for program startup, message typeout, and program termination. The program shown in Figure 11 provides an example of the use of this macro library. The assembly shown in Figure 11 lists both the macro calls and the statements in the macro expansions which generate machine code. The statements which are marked by '+' in Figure 11 are generated from the macro calls, while the remaining statements are a part of the calling program.

As an introduction to MAC features, the macro invocation

```
ENTCCP  10
```

in Figure 11 shows a specific expansion of ENTCCP (enter from CCP) which is defined in the macro library given in Figure 10. The macro call causes MAC to retrieve the definition (i.e., the text between MACRO and ENDM in Figure 10) and substitute this text following the macro call in Figure 11. This particular macro performs the following function: upon entry to the program from the CCP, the stack pointer (SP) is saved 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 which is named in the macro definition as SSIZE (see Figure 10), and filled-in at the call with the value 10. The result is that 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 THE LITERAL MESSAGE AT THE CONSOLE
       LOCAL    PASTSUB ;;JUMP PAST SUBROUTINE INITIALLY
       JMP      PASTSUB
MSGOUT:         ;;THIS SUBROUTINE IS USED TO PRINT 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
;
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
```

Figure 10. A Sample Macro Library.

```
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
```

Figure 11. A Sample Assembly using the MACLIB Facility.

Consider also the special macro statements which are used in Figure 10 within the body of the ENTCCP macro. The "local" statement defines the label START which is used within the macro body. Generally, 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 which do not interfere with one another. To continue the example, ENTCCP also contains a conditional assembly statement which uses the "NUL" operator, which is used to test whether a macro parameter has been supplied or not. In this case, the ENTCCP macro could be invoked by:

```
ENTCCP
```

with no actual parameter, resulting in a default stack size of 32 bytes. If this seems confusing, don't be concerned at this point because the individual sections which follow give exact details and examples.

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

In order to see the use of such a nested structure, consider the purpose of the TYPEOUT macro. Each time it is invoked, 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, we would like to include the subroutine "inline," and 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.

It should be noted that macro definitions are stored in the symbol table area of the assembler and thus each macro reduces the remaining free space. As a result, MAC allows "double semicolon" comments which 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 while comment with two preceding semicolons are listed only when the macro is defined.

Figure 11 gives three examples of TYPEOUT invocations, with three messages which 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 which is included inline between addresses 0126H and 0133H. The utility subroutine is then called, followed by another jump around the console message which is also included inline. Note, however, that subsequent invocations of TYPEOUT use the previously included utility subroutine to type their messages. Again, this may seem confusing, but it is worthwhile studying this example before continuing into the exact details of macro definition and invocation in order to gain some insight into macro facilities.

It should also be noted that, although the example shown here concentrates all macro definitions in a separate macro library, it is often the case that macros are defined in the mainline (.ASM) source program. In fact, many programs which use macros do not use the external macro library facility at all.

There are many applications of macros which will be examined throughout the remainder of this manual. Specifically, macro facilities can be used to simplify the programming task by "abstracting" from the primitive assembly language levels. That is, the programmer can define macros which provide more generalized functions that are allowed at the pure assembly language level, such as macro languages for a given applications (see Section 10), improved control facilities, and general purpose operating systems interfaces. The remainder of this manual first introduces the individual macro forms, then presents several uses of the macro facilities in realistic applications.

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 repetively re-read portions of the source program under control of a counter or list of textual substitutions. These groups are listed below in increasing order of 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 which are enclosed within the group.

Figure 12 shows an example of the use of the REPT group. In this case the REPT-ENDM group is used to generate a short table of the byte values 5, 4, 3, 2, and 1. Upon entry to the REPT, the value of NXTVAL is 5 which is taken as the repeat count (even though NXTVAL changes within the REPT). Note that the macro lines which do not generate machine code are not listed in the repetition, while the lines which do generate code are listed with a "+" sign after the machine code address. Full macro tracing is optional, however, using assembly parameters, as discussed in a later section.

In general, if a label appears on the REPT statement, its value is the first machine code address which follows. This REPT label is not re-read on each repetition of the loop. The optional label on the ENDM is re-read on each iteration and thus constant labels (not generated through concatenation or with the LOCAL pseudo operation) will generate phase errors if the repetion count is greater than 1.

Properly nested macros, including REPT's, can occur within the body of the REPT-ENDM group. Further, nested conditional assembly statements are also allowed, with the added feature that conditionals which begin within the repeat group are automatically terminated 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 re-read a bounded set of statements, taking the form

```
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
```

Figure 12. A Sample Program Using the REPT Group.

```
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, and "character-list" denotes a string of characters, terminated by a delimiter (space, tab, end-of-line, or comment).

The IRPC controls the re-read 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
```

re-reads the statement sequence which follows (to the balancing ENDM) a total of five times, once for each character in the list "ABCDE." On the first iteration, the character "A" is associated with the identifier "?X" and 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 above 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 (&). Again using the above IRPC header, the macro assember would replace "LAB&?X" by "LABA" on the first iteration, while "LABE" would be produced on the final iteration. The concatenation feature is most often used to generate unique label names on each iteration of the IRPC re-read process.

Note, however, that the controlling identifier is not normally substituted within string quotes, since the controlling identifier could quite possibly occur as a part of a quoted message. Thus, the macro assembler performs substitution of the controlling identifier when it is either preceded and/or followed by the ampersand operator. Further, recall that all alphabetics outside string quotes are translated to upper case, while no case translation occurs within string quotes. This requires that the controlling identifier be not only preceded or followed by the concatenation operator within strings, but must also be typed in upper case.

Figure 13 illustrates the use of the IRPC-ENDM group. Figure 13a shows the original assembly language program, before processing by the macro assembler. Note that the program is typed in both upper and lower case. Figure 13b 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 which save the double precision registers BC, DE, and HL. Again note that 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
```

Figure 13a. 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
```

Figure 13b. Resulting (.PRN) file with IRPC Example.

The second IRPC shown in Figure 13 uses the controlling identifier "C" to generate a number of single byte constants with corresponding labels. It is important to observe that although the controlling variable was typed in lower case (see Figure 13a), it has been translated to upper case during assembly. Further, note that the string '&C' occurs within the group and, since 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.

It should be pointed out that substitution of the controlling identifier by its associated value could 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 re-reading the substituted text for another possible occurrence of the controlling identifier. Thus, an IRPC of the form

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

would produce

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

in place of the DB statement at the label DATAA in Figure 13b.

The last IRPC of Figure 13 is used to restore the previously saved double precision registers, and performs the exact opposite function from the IPRC at the beginning of the program.

One special case does occur, however, when the character-list is empty (i.e., when no characters occur following the "identifier," portion of the IRPC header). In this case, the group of statements is read once, and any occurrence of the controlling identifier is deleted when it is read (i.e., it is replaced by the "null string").

7.3. The IRP-ENDM Group.

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

```
label: IRP    identifier,¼cl-1,cl-2,...,cl-n½
       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 "cl-1" is substituted for the identifier wherever the identifier occurs in the bounded statement group (statements 1 through m). On the second iteration, cl-2 becomes the value of the controlling identifier. Iteration continues in this manner

until the last character-list, denoted by cl-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 "&"). One should also note that controlling identifiers are always ignored within comments.

Figure 14 gives several examples of IRP groups. The first occurrence of the IRP in Figure 14 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 re-reading the IRP group, substituting the actual label for the formal name on each iteration.

The second occurrence of the IRP group in Figure 14 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. Note that the controlling identifier is replaced by the character-lists in the two cases "&IS" and "IS&" inside the string quotes since they are both adjacent to the ampersand operator. Note further that "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  ''
```

which 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), while in the last case only one of the double ampersands is absorbed in the '&&?X&' string. In this last case, the two ampersands which surround "?X" are removed since they occur immediately next to the controlling identifier within the string.

Recall that substitution rules outside of string quotes and comments is 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. Further, 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 as well as escape sequences to 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
```

Figure 14. A Sample Program Using IRP.

7.4. The EXITM Statement.

The EXITM pseudo operation can occur within the body of a macro and, 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.

In order to be useful, the EXITM statement normally 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 at the point 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 Figure 15. This figure shows two IRPC's used to generated "DB" statements which do not exceed eight characters in length. These IRPC's 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" is used to count the number of filled characters. If the count ever 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" since 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. Thus, this IRPC terminates normally by exhausing 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 (note that "=" and "EQ" are equivalent operators), resulting in assembly of the EXITM statement. 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 since the conditional begins within the macro body. The ENDM serves the dual purpose of terminating unmatched IF's as well as 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
```

Figure 15. 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 which are unique on each repetition of a macro. This facility is available through the LOCAL statement, which 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 as discussed above (or a MACRO heading as discussed in following sections), and id-1 through id-n represent one or more assembly language identifiers which do not contain embedded "$" separators. The LOCAL statement must occur within the body of a macro definition. Although MAC allows the LOCAL statement to appear anywhere within the macro body, it should appear immediately following the macro header to be compatible with the standard Intel macro facility.

The action of the assembler upon encountering the LOCAL statement is to create a new name 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 one of the identifiers in the list is encountered, the corresponding created name is substituted in its place. Substitution occurs according to the same rules as the controlling identifier in the IRPC and IRP groups.

The user should avoid the use of labels which begin with the two characters "??" so that no conflicting names will accidentally occur. Further, symbols which begin with "??" are not normally included in the sorted symbol list at the end of assembly (see "assembly parameters" to override this default). Lastly, a total of 9999 LOCAL labels can be generated in any assembly, and an overflow error will occur if more generations are attempted.

Figure 16a shows an example of a program which uses the LOCAL statement to generate both data references and jump addresses. This program uses the CP/M disk operating system to print a series of four generated messages, as shown in the output from the program in Figure 16b. The program begins with "equates" which define the disk system primary entry point, along with names for the non graphic ASCII characters CR and LF (carriage return and line feed). The REPT statement which follows contains a LOCAL statement with the identifiers X and Y which are used throughout the body of the REPT group. On the first iteration, X's value becomes ??0001 which is the first generated label, while Y's value becomes ??0002. Note that 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 CALL's 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 disk 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
```

Figure 16a. 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
```

Figure 16b. Output from Program of Figure 16a.

Upon completion of the program, control returns to the console command processor (CCP) for further operations. This particular program uses the default stack which is passed by the CCP (approximately 16 levels are available). Although this example is primarily intended to show operation of the LOCAL statement, the reader may wish to consult the CP/M Interface Guide to determine BDOS interface conventions in order to follow this example completely.

8. DEFINITION AND EVALUATION OF STORED MACROS

The "stored macro" facility of MAC allows the programmer to name a sequence of assembly language "prototype" statements for selective inclusion at various places throughout the assembly process. Macro parameters can be supplied in various forms at the point of expansion which are substituted as the prototype statement are re-read. These parameters are generally used to tailor the individual macro expansion for 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 causes source text to be saved in the assembler's internal tables, and any particular expansion involves manipulation and re-reading of the saved text. These concepts will become clear as the individual macro forms are discussed.

In general, macro features can be combined in various ways to greatly enhance the facilities which are available to the programmer. Specifically, the programmer can easily manipulate generalized data definitions, macros can be defined for generalized operating systems interface, simplified program control structures can be defined and non standard instruction sets (such as the Z-80) can be supported. Finally, well designed macros for a particular application can achieve a measure of machine independence. All of these notions will be covered in the sections which follow.

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 non conflicting assembly language identifier, d-1 through d-n constitutes a (possibly empty) list of assembly identifiers without imbedded "$" separators and statements-1 through m are the macro prototype statements. The identifiers denoted by d-1 through d-n are called "dummy parameters" for this particular macro and, although they must be unique among themselves, can generally be identical to any program identifiers outside the macro body without causing a conflict. The prototype statements may contain any properly balanced assembly language statements or groups, including nested REPT's, IRP's, IRPC's, MACRO's and IF's.

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

As before, the label preceding the ENDM is optional.

8.2. Macro Invocation.

The macro text which is 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) by associating the first dummy parameter with the first actual parameter (d-1 is paired with a-1), the second dummy is associated with the second actual, and so forth until the list is exhausted. If more actuals are provided than dummy parameters then the extras are ignored. If fewer actuals are provided then the extra dummy parameters are associated with the empty string (i.e., a text string of zero length). It is important to realize at this point that 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 re-reads 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.

Figures 17 and 18 provide examples of macro definitions and invocations. Figure 17 begins with the definition of three macros, called SAVE, RESTORE, and WCHAR. The SAVE macro contains prototype statements which save the principal CPU registers (PUSH PSW, B, D, and H), while 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.

Note that the occurrence of the SAVE macro definition between MACRO and ENDM causes the assembler to read and save the PUSH's, but does not assemble the statements into the program. Similarly, the statements between the RESTORE MACRO and corresponding ENDM are saved, as are the statements between the WCHAR MACRO and ENDM group. 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 Figure 17, note that machine code generation starts following the invocation of the SAVE macro. The prototype statements which were previously stored are re-read 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. Note that the SAVE macro has no dummy parameters in the definition and thus there are no actual parameters required at the point of invocation.

The invocation of SAVE 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. Note that the use of CHR is within string quotes and thus must be 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
```

Figure 17. 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 is invoked, causing generation of the POP statement to restore the register state. The RESTORE is followed by a RET to return to the CCP following the character output.

This particular program thus performs the simple function of saving the registers upon entry, typing the two characters "HI" at the console, restoring the registers, and then returns to the Console Command Processor. One should note that the SAVE and RESTORE macros are used here for illustration, and are not required for interface to the CCP since all registers are assumed invalid upon return from a user program. Further, this program uses the CCP's stack throughout, which is only eight levels deep.

Figure 18 shows another macro for printing at the console. In this case, the PRINT macro uses the operating system call which prints the entire message starting at a particular address until the "$" symbol is encountered. The PRINT macro has a slightly more complicated structure: two dummy parameters must be supplied in the invocation. The first parameter, called N, is a count of the number of carriage-return line-feeds to send after the message is printed. The second parameter, called MESSAGE, is the ASCII string to print which must be passed as a quoted string in the invocation. The LOCAL statement within the macro generates two labels denoted by PASTM and MSG. When the macro expands, substitutions will occur for the two dummy parameters by their associated actual textual values, and for PASTM and MSG by their sequentially generated label values. The macro definition contains prototype statements which branch past the message (to PASTM) which is included inline following the label MSG. The message is padded with N pairs of carrriage-return line-feed sequences, followed by the "$" which marks the end of the message. The string address is then sent to the BDOS for printing at the console.

There are two invocations of the PRINT macro included in Figure 18. The invocation sends two actual parameters: the textual value 2 is associated with the dummy N, followed by a quoted string which is associated with the dummy parameter MSG. Note that the second actual parameter includes the string quotes as a part of the textual value. Note also that 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 generations of the carriage-return line-feed pairs.

Similar to Figure 17, the program of Figure 18 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.

8.3. Testing Empty Parameters.

Before continuing the discussion of macro definition and invocation, it is necessary to discuss a particular operator, called the NUL operator, which is specifically designed to allowing testing of null parameters (i.e., actual parameters of length zero). The

```
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
```

Figure 18. Sample Message Print-out Macro.

NUL operator is used in an expression as a unary operator, and produces a true value if its argument is of length zero and a false value if the argument has 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. Note that the NUL differs from other operators since it must appear as the last operator in the expression. This is due to the fact that 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 since NUL absorbs the argument XXX (producing a false value) in the scan for the end of line. The expression

X GT Y AND NUL

is also valid, however, since the argument following the NUL is empty, thus causing NUL to return a true value since the end of line is immediately encountered in the scan. Intervening blanks and tabs are ignored in this scanning process. The expression

X GT Y AND NUL M + Z)

is somewhat deceiving, but nevertheless valid even though it appears as if it is an unbalanced expression. In this case, the argument following the NUL operator is the entire sequence of characters "M + Z)" which is absorbed by the NUL operator in scanning for the end of line. The value of "NUL M + Z)" is "false" since the sequence is not empty.

Figure 19 gives several examples of the use of NUL in a particular program. In the first case, NUL returns true since 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 Figure 19 produces a false value since the argument is non-empty. Both uses of NUL, however, are contrived examples, since NUL is really only useful within a macro group, as shown in the definition of the NULMAC macro.

NULMAC consists of a sequence of three conditional tests which 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. Six invocations of NULMAC follow its definition, giving various combinations of empty and non-empty actual parameters.

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

The second invocation of NULMAC provides only one actual parameter (XXX) which is assigned to the dummy parameter A, while B and C are both assigned the

```
                                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
```

Figure 19. Sample Program using the NUL Operator.

empty sequence. Thus, only the "DB" for the first conditional test is assembled.

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 since 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, while 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, since B&C produces the sequence YYY.

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

The final expansion of NULMAC in Figure 19 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. Note that the value of A is the empty string in this case, while the value assigned to both B and C consists of the two apostrophe characters side-by-side, which 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 produces a false value since A is associated with the empty sequence. The second conditional, however, evaluates the form

```
NOT NUL ' '
```

which is the special case of NUL applied to a length zero quoted string (not a length zero sequence, however). 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 juxtaposed. Considering only the four adjacent apostrophes, the macro assembler considers this a quoted string which happens to contain a single apostrophe, since double apostrophes within strings are always reduced

to a single apostrophe. As a result, the test produces a true value and the conditional segment is assembled. If this all seems confusing, that's because it is. Fortunately, these cases are very specialized, and are included here for completeness. Under normal circumstances, the NUL operator is used only to test for missing arguments, as shown in later examples (see Figure 22 for a particular case).

8.4. Nested Macro Definitions.

The MAC assembler allows the programmer to include nested macro definitions, which 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 which 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.

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

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

Figure 20 gives a practical example of nested macro definition and expansion. This particular program writes characters to either the CP/M console device or the currently assigned list device, according to the value of the LISTDEV flag which is 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 which sends a single character to whatever device is selected.

For purposes of illustration, the macro SETIO is used to construct the OUTPUT macro. Note in Figure 20 that the OUTPUT macro is wholly contained within the SETIO macro and, as a result, remains undefined until SETIO expands. 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. Since LISTDEV is true for this assembly, the OUTPUT macro becomes defined as

```
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
```

Figure 20. Sample Program showing a Nested Macro Definition.

```
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 since 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, respectively.

8.5. Redefinition of Macros.

It is often useful to redefine the prototype statements of a particular macro after the initial prototype statements have been entered. This is often simply a particular case of the previous section, where the inner nested macro carries the same name as the encompassing macro definition. Although this feature may seem somewhat frivolous, there is one particular case where macro redefinition is extremely useful: if the macro uses a subroutine then the subroutine can be included on the first expansion and simply called in any remaining expansions. Thus, if the macro is never invoked then the subroutine is not included in the program.

Figure 21 shows an example of macro redefinition. In this case, the macro MOVE is defined which 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 sufficiently complicated that they should be performed through a subroutine, rather than inline machine code each time MOVE is expanded.

Examining the structure of MOVE in Figure 21, 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
```

The action of the assembler upon encountering the first invocation of MOVE is to begin reading the prototype statements. Note, however, that the first expansion of the MOVE includes the subroutine for the actual move operation, labelled by @MOVE so that there is no name conflict (with a branch around the subroutine). MOVE then redefines itself as a sequence of statements which simply call the out-of-line subroutine each time it expands. In fact, the last statement of the original MOVE macro is an

```
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
                ;
                        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!'
```

Figure 21. Sample Program showing Macro Redefinition.

invocation of the newly defined version. As indicated by this example, once a macro has started expansion, it will continue to completion (or until EXITM is assembled), even if it redefines itself.

It is important to note the use of ?S, ?D, and ?C in the above 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, since they would be substituted by their actual values if they were the same. This is due to the fact that the inner MOVE macro is wholly contained within the outer macro and thus parameter substitution takes place irregardless of the context.

Macro storage is not reclaimed upon redefinition, however, since 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.

A "recursive" macro x has the property that its prototype statements contain invocations of macros which, in turn, invoke macros which eventually lead back to an invocation of x. A particular case of recursion, called "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, note that "macname" is being 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 which prevent infinite recursion. In fact, recursion is only allowed to sixteen levels before returning to complete the expansion of an earlier level.

Figure 22 shows a situation where (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. In particular, CBDOS loads the specified function to register C, then tests to see if 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 in the particular case that 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. Note, however, that 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.

An important observation in the presence of recursion is that the values of the dummy parameters are saved at each successive level of recursion, and restored when

```
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
                        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
```

Figure 22. Sample Program showing a Recursive Macro.

that level of recursion is re-instated. In particular, re-entry into a macro expansion through recursion does not destroy the values of dummy arguments held by previous entry levels.

8.7. Parameter Evaluation Conventions.

There are a number of options which the programmer can exercise in the construction of actual parameters, as well as 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 to become a part of the text. In general, 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 elipses ". . ." represent optional surrounding actual parameters in the invocation of macname. In the case of an IRP group, the occurrence of a character-list x would be

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

where the label is again optional, and the elipses represent optional surrounding character-lists for substitution within the IRP group where the controlling identifier "id" is found. In either case, the statements could be contained within the scope of a surrounding macro expansion. Hence, dummy parameter substitution could 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:

(a) leading blanks and tabs (control-I) are removed if they occur in front of x. After this "deblanking" has occurred,

(b) the leading character of x is examined to determine the type of scan operation which is to take place;

(c) if the leading character is a string quote (apostrophe), then x becomes the text up through 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 their actual parameter values. Note that the string quotes on either end of the string are included in the actual parameter text.

(d) If instead the first character is the left broken bracket "<" then the bracket is removed, and the value of x becomes the sequence of characters up to, but not including, the balancing right broken bracket ">" which does not become a part of x. In this case, left and right broken brackets may 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 (c) above. Note that left and right brackets within quoted strings become a part of the string, and are not counted in the bracket nesting within x. Further, the delimiter

characters comma, blank, semicolon, tab, and exclaim become a part of x when they occur within the bracket nesting.

(e) If the leading character is a percent (%), then the sequence of characters which follows is taken as an expression which 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).

(f) If the leading character is neither a quote nor a left bracket nor a percent, the (possibly empty) sequence of characters which follow, up to the next comma, blank, tab, semicolon, or exclaim symbol, becomes the value of x.

There is one important exception to the above rules: the single character escape, denoted by an up-arrow, causes the macro assembler to read the immediately following special (non alphabetic) character as a part of x without treating the character as significant. The character which follows 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 must also be considered, although this topic has been presented throughout the previous sections. Generally, the macro assembler evaluates dummy parameters as follows:

(a) If a dummy parameter is either preceded or followed by the concatenation operator (&), then the preceding and/or following "&" operator is removed, the actual parameter is substituted for the dummy parameter, and the implied delimiter is removed at the position(s) the ampersand occurs.

(b) Dummy parameters are replaced only once at each occurrence as the encompassing macro expands. This prevents the "infinite substitution" which would occur if a dummy parameter evaluated 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

Figures 23, 24, and 25 show examples of macro definitions and invocations which illustrate these points. In Figure 23, for example, two macros are defined, called MAC1 and MAC2, which each have several dummy parameters. In this case, the macro definitions are headed by "DB" statements in order to reveal the actual values which are passed in each case. There is a single (mainline) invocation of MAC1 with the actual 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
                        ;
000F =                  X       EQU     15
                                MAC2    I  ,,  X+1,  % X + 1, 'kwote'
    +                   ;
    +                   ;       ENTERING MACRO 2:
0000+492020582B                 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
```

Figure 23. Macro Parameter Evaluation Example.

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

which assocates 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. 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 since B's value is M. Note that the concatenation of C with 1 reduces to a concatenation of A with 1 since C's value is A. The replacement of C by A constitutes a substitution of a single occurrence of a dummy parameter, and thus the A which 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.

Figure 24 illustrates the use of bracketed notation, using IRP's (indefinite repeats) within two 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 which are reconstructed as a single list at the IRP heading which 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 invocation of IRPM4, the second level is removed at the nested invocation of IRPM1 inside IRPM4, and the innermost level is required at the IRP heading within IRPM1.

Figure 25 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 which shows 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 which produces an empty value since no characters remain after the brackets are removed. The second parameter includes a quoted string ('string of pearls') and a hexadecimal value which becomes a part of the "DB" in MAC1.

The third invocation of MAC1 passes a bracketed expression, which includes a quoted string (i.e., the pair of adjacent apostrophes), followed immediately by a sequence of ASCII characters. Note that the pair of apostrophes are passed intact since 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. In particular, one must take care in passing quoted strings which themselves contain apostrophes, since 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
                ;
                        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
```

Figure 24. Parameter Evaluation using Bracketed Notation.

```
                        ;       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''''''''s 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)
```

Figure 25. Examples of Macro Parameter Evaluation.

It is worthwhile examining the various parameters and their evaluations in Figure 25 to ensure that the rules for evaluation given in this section are consistent.

8.8. The MACLIB Statement.

The macro assembler allows the programmer to create and reference "macro library" files which are external to the mainline program. The form of the macro library reference is

```
MACLIB    libname
```

where "libname" is an identifier which references a particular file "libname.LIB" which is assumed to exist on the diskette. Macro libraries are in source program form, and can thus be easily created and modified by the programmer using the CP/M system editor (ED).

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:

(a) the statements included in the macro library cannot generate machine code. For example, comments, EQU's, SET's, and MACRO definitions are allowed, while DB statements outside macro definitions are not allowed.

(b) Macro libraries are not normally listed with the source program (although there is an overriding parameter which can be supplied - see Assembly Parameters).

(c) All MACLIB statements must appear before the mainline program macro definitions. Generally, 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 the programmer can predefine macros which enhance the facilities of the assembly language itself. For example, the additional operations codes of the Zilog Z-80 microprocessor can be defined in a macro library which is reference in a single statement

```
MACLIB    Z80
```

which causes the assembler to read the file "Z80.LIB" from the diskette, containing the necessary macros for Z-80 code generation. These macros can then be referenced within the program intermixed with the usual 8080 mnemonics.

Normally, the "libname.LIB" file is assumed to exist on the currently logged disk drive. The programmer can override this default condition using a special parameter (L) when the macro assembler is started which redirects the ".LIB" references to a different diskette (see Assembly Parameters).

Figures 10 and 11 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.

9. APPLICATIONS OF MACROS

The MAC assembler provides a powerful tool for microcomputer systems development through its macro facilities. In order to demonstrate this tool, a number of applications of macros in the solution of practical problems are described in some detail in the following sections. Four particular applications areas are considered: use of macros in implementation of special-purpose languages, emulation of non-standard machine architectures, implementation of additional control structures, and 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 is to carry-out in performing its particular task. In order to avoid unnecesary 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 which suits the particular application. The application programmer then 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 which implement the primitive operations. Further, the macro bodies can incorporate debugging facilities for application development.

In order to illustrate the notion of 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 which are 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 which show a configuration of their standard hardware for each intersection, along with the "algorithms" required for traffic flow at that point.

The standard hardware items which Hornblower manufactures consist of the following. Central and corner traffic lights which 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 which control the lights, and "latches" which holds the sensor input information. The controller box also contains a time of day clock, which 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 a set of hardware configuration plans which intermix the various standard components. Programs are then written and debugged which control each intersection, based upon predicted traffic patterns.

The intersection of Easy St. and Maria Ave., for example, controls minimal traffic and thus consists of a controller box with a single central light. The "algorithm" for this intersection is to simply alternate red and green lights between Easy and Maria, with a "bias" toward Easy St., since 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 last only 15 seconds. Given this situation, the application programmer writes the following program:

```
;       HORNBLOWER HIGHWAYS SYSTEMS, INC.
;       INTERSECTION:
;             EASY ST.(N-S) / MARIA AVE. (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 which implement the "primitive" operations SETLITE and TIMER which set 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. Note that the sequence of operations is easy to write, and is completely machine independent.

Figure 26 gives an example of a macro library for "intersect" which assumes the following hardware with an 8080 processor: the central traffic light is controlled by the 8080 output port 0 (given by "light"), while 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, while 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 Figure 26 accepts both 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 souuth 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
;;      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
```

Figure 26. Macro Library for Basic Intersection.

The TIMER macro in Figure 26 uses the internal cycle time of the 8080 processor to construct an inline timing loop, based on the value of SECONDS. Note that this loop is not generated as a subroutine, since 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. Figure 27a, for example, is included when the intersection contains treadles in the street to detect automobiles, while Figure 27b 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 which 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, labelled 0, 1, through a maximum of 7 treadles. Each sensor and reset position of the treadle ports correspond 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."

Figure 27b shows the macro library which processes pedestrian pushbuttons. Hornblower's hardware is set up to sense the latched pedestrian switches on input port 0 ("cwinp") as a sequence 1's and 0's 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. In addition to these macro libraries, Hornblower has defined several additional libraries which support optional hardware manufactured by their company.

The intersection of Bumpenram Boulevard and Lullabye Lane presents a somewhat more complicated situation. Bumpenram Blvd. carries heavy traffic in an E-W direction to and from the center of town. Lullabye Ln., however, feeds a residential portion of the city, running perpendicular to Bumpenram in a N-S direction. The contracting city has specified that the traffic control should be biased toward Bumpenram Blvd. 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 Ln., allowing all traffic to clear before returning to the major E-W flow along Bumpenram Blvd. Late night traffic along Bumpenram is not very heavy, so the city has also specified that the E-W light flashes yellow and and N-S direction flashes red between the hours of 2 and 5 AM.

The application program created by Hornblower for the Bumpenram Blvd. and Lullabye Ln. intersection is shown in Figure 28. Each major cycle of the traffic light enters at "CYCLE" where the time of day is tested. If between 2 and 5, then control transfers to "NIGHT" where the yellow/red lights are flashed in the appropriate directions. If not between 2 and 5 AM, the switches and treadles are sampled until N-S traffic along Lullabye Ln. is sensed. If cross traffic is detected, the lights switch until all the traffic is through. Sampling also stops if the time of day ever reaches 2 AM.

```
;       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     trout           ;;clear it
        jmp     iftrue          ;;go to true label
iffalse:
        endm
```

Figure 27a. 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
```

Figure 27b. Macro Library for Corner Pushbuttons.

```
                ;       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 2 AM?
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
```

Figure 28a. 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
```

Figure 28b. 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+0EB6       ??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+3E01               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
```

Figure 28c. Algorithm with Generated Instructions.

Figure 28a shows the assembly with no macro generated lines (controlled by the "-M" parameter - see Assembly Parameters). Although the machine code locations are shown to the left, no 8080 machine code is listed. Figure 28b shows a segment of this same program with machine code generation, but no 8080 mnemonics (controlled by "*M"), while Figure 28c shows another segment with normal macro generation. Note that Figure 28a is the most readable to the application programmer, while Figures 28b and 28c would be useful for macro debugging.

It should be noted that the resulting program requires no random access memory for execution, since all temporary values are maintained in the 8080 registers. Further, no subroutine calls take place and thus the 8080 stack is not used. Finally, 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.

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. Figure 29 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. Figure 29 includes the debug flag at the beginning of the library (initially set 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. Note, for example, that 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.

Figure 30a shows an application program for a particular 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 which will eventually be assembled to drive the actual Hornblower hardware.

The application programmer then uses CP/M to trace the operation of the algorithm, which results in the print-out shown in Figure 30b. Each trace line corresponds to an invocation of SETLITE with a specific direction and color, with the appropriate wait time between print-outs.

Upon completion of the initial debugging under CP/M, the SET statement in the application program is removed (the ORG may be removed as well), and the program is re-assembled. This time, the CP/M traces are not included since 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 enviroment.

```
;       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 souuth 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 macro   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")
;
```

Figure 29. Library Segment with Debug Facility.

```
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
0189                    SETLITE EW,RED
01A9                    SETLITE NS,GREEN
01CB                    TIMER   10
01DD                    SETLITE NS,YELLOW
0200                    TIMER   2
0212                    RETRY   CYCLE
```

Figure 30a. 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
        . . .
```

Figure 30b. Debug Trace Printout.

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 desired. 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 found in the "emulation" of a machine operation code set which is different from the 8080 microprocessor. In particular, a machine architecture is selected, based upon 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 which carry-out the operation on the 8080 microprocessor. That is, 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, which expand to the equivalent 8080 instructions, but perform the emulated machine operations.

In order to be specific, consider the situation encountered by Nachtflieger Maschinenwerke, 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 primarly 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 twelve bits. In order to allow computations on these twelve 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 (A-D) input "ports" which can be read by KDF-10 programs, along with four digital to analog output ports (D-A) which can be written by the program. The KDF-10 will automatically perform the A-D and D-A conversion at these ports.

Begin forward thinkers, the engineers at Nachtflieger have designed the KDF-10 as a "stack machine," which is similar in concept to the Hewlett-Packard HP-65 hand held 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. Somewhat simpler than the HP-65, the designers settle upon 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, automatically pushing any

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 — The top of the KDF-10 stack is duplicated.

SUM — The top two elements of the KDF-10 stack are added, 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. Note that LSR n performs a division of the topmost stacked value by the divisor 2^n.

JMP a — Branch directly to the program address given by the label a.

Since the KDF-10 does not exist (except in the fertile minds of Nachtflieger engineers), the software designers have decided to use the macro facilites of MAC to emulate the KDF-10 using the 8080 microcomputer.

Figure 31 shows an example of a program for the KDF-10 which 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 which are 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 then sent to D-A output port 0 where it is used to set environmental controls.

Referring to Figure 31, the program begins by reserving a stack of 20 elements, which is much larger than required for this application (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 which follow the read operations perform pairwise addition of the temperature values, producing a single sum at the top of the stack. Since the average value is desired, 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.

Since Nachtflieger designers are emulating KDF-10's using 8080's, they have created the macro library file, called "STACK.LIB" as shown in Figure 32. A macro is shown in this figure 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

```
                    ;       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
0132                        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
```

Figure 31. A-D Averaging Program using "Stack Machine."

```
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
;
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
```

Figure 32. "Stack Machine" Opcode Macros.

reserved since a 16-bit word size is assumed. The DUP, SUM, and LSR operators follow the SIZ macro. In each case, the KDF-10's stack top is assumed to be in the 8080's HL register pair. Further, each operation which 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.

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. Thus, it must be the case that the HL registers contain the most recently loaded value, while 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, which is necessary in the KDF-10 emulation, since the top of the KDF-10 stack is located in the 8080's HL register pair.

The LSR opcode is somewhat more complicated. Since the 8080 does not support a double precision (16-bit) right shift of the HL register pair, the values must go through the accumulator. Thus, the LSR macro contains a REPT loop which 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 may move from the high order byte to the low order byte using the carry between high and low order byte shifts.

Referring to Figure 32, 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 location 1080H/1081H to HL is treated as a read from A-D device 0, rather than from random access memory. This operation is simple to perform in the KDF-10 emulation, since all program addresses are assumed to be below 1000H, and thus 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 which are written to locations 1090H through 1097H are intercepted as memory mapped output values which are sent to the D-A converters rather than random access memory. 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, since 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.

In order to see the actual code generated by each of these macros, Figure 33 shows the same averaging program as given in Figure 31, except that the generated 8080 instructions are interspersed throughout the listing file (Figure 33 is the usual output from MAC, while Figure 31 was generated using the parameter "-M" which suppresses generated mnemonics). It is worthwhile cross-referencing Figures 31, 32, and 33 to ensure that the macro expansion processes are clearly understood.

```
                ;       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

                ;       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
```

Figure 33. Averaging Program with Expanded Macros.

A particular problem arose at Nachtflieger MW, however, which 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 of Figure 33, for example, could be tested under the CP/M debugger (see the CP/M DDT Users Guide), but 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" which allow debugging traces. The opcodes can be interspersed in the program, and selectively enabled and disabled depending upon the debugging needs. In production, all debugging traces would, of course, be 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 broken 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 which physically appear between the TRT and TRF opcodes are shown in the trace.

TRT p — Enable the print/read trace. PRN opcodes which 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 which physically appear between TRT and TRF interact with the console.

The convention is also taken that the traces are initially disabled at the beginning of the program, and must be explicitly enabled with TRT opcodes.

Figure 34 shows the averging program of Figure 31 with interspersed debugging statements. Note that the opcodes TRT t and TRT p are executed at the beginning

```
                ;       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
```

Figure 34. Averaging Program with Debugging Statements.

of the program, thus enabling all trace options throughout the execution. The PRN statement above the LOOP label prints the initial sign-on, while 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 is used to indicate this fact. Each SUM operator is followed by a DMP opcode which shows the current sum. Finally, the PRN and DMP opcodes are used to display the final average value which is being sent to D-A port 0. The "XIT" opcode shown at the end of the program will be introduced in the paragraphs which follow.

Figure 35 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 which shows the absolute memory mapped output address as well as the data which 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 Figure 34 (012E and 01EF), and is followed by the two values 0111 (i.e., the value just read) and C21D ("garbage" value, since only one element is stacked). The trace is easily followed at the KDF-10 level, showing each value which 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.

Forward thinking though they were, 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, which incorporates these features. In particular, the operation codes of the KDF-11 include:

DCL v,n — Declare (i.e., 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, while 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 — Similar to the VAL operator, the STO opcode stores the value obtained from the KDF-11 stack to the

```
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
```

Figure 35. Sample Execution of "Average" using DDT.

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 — The DIF opcode subtracts the top element of the KDF-11 stack from the next-to-top element of the stack, and replaces both operands by their difference.

GEQ a — The GEQ opcode tests the next to top element (top') against the top of stack element (top), and branches 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 — The BRN instruction replaces the JMP instruction in the KDF-10 architecture to allow complete separation of the KDF-11 and 8080 machines.

Figures 36a, 36b, 36c, and 36d give the macro library which was constructed by the Nachtflieger software group for KDF-11 machine emulation. Note that over half of the macro library implements trace and debugging functions (Figures 36a and 36b) while the remaining components implement the KDF-11 opcodes themselves. A brief description is given below for each major section of this macro library, called "DSTACK.LIB," before giving an example of its use.

Figure 36a shows the first portion of the macro library. Since this portion of the library is principally concerned with debugging functions, it 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 which operate when the TRT opcode is in effect. Following the CP/M equates, the "toggles" DEBUGT and DEBUGP are set to false (0 value), which reflect 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 shown in Figure 36a (left), 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 which follows PRN in Figure 36a is invoked the first time that the debugging subroutines are required by trace or print/read opcodes. When invoked, the UGEN macro produces several inline subroutines which are used throughout the debugging process. If no trace or print/read functions are invoked during the assembly, UGEN is not invoked and 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

```
;       macro library for a zero address machine
;       ****************************************
;       *       begin trace/dump utilities     *
;       ****************************************
bdos    equ     0005h   ;system entry
rchar   equu    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
;;
@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      @in1    ;;carry if 0,1,...,9
;;      may be hexadecimal a,...,f
        sui     'A'-'0'-10
        cpi     16      ;;a through f?
        rnc             ;;return with assumed cr
@in1:   ;;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
;;
psub:
ugen    macro
;;      redef to include once
        endm
        ugen    ;;generate first time
        endm
;       ****************************************
;       *       end of trace/dump utilities    *
```

Figure 36a. Stack Machine Macro Library.

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

Figure 36b. Stack Machine Library (Con't).

```
;       *       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
;
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
;
```

```
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
;
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
```

Figure 36c. Stack Machine Library (Con't).

```
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
;
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
;
;       ****************************************
;       *       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
;
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           *
;       ****************************************
```

Figure 36d. Stack Machine Library (Con't).

Upon including these subroutines, UGEN then redefines itself (see lower right of Figure 36a) to an empty macro body so that the subroutines will not be included upon subsequent invocations of UGEN. This ensures that the inline subroutines will only be included once, and only if they are required by the debugging macros.

Referring again to Figure 36c, the SIZ macro is similar 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). The SAVE and REST macros are used throughout the opcode macros to save and restore the HL register pair, based upon the ACTIVE flag. The CLEAR macro, however, is used to mark the top element of the KDF-11 stack as deleted.

Continuing with Figure 36c (left), the DCL macro simply sets up the variable name VNAME as a label, and follows the label by a DS which 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 which first SAVEs the stack top (possibly generating an HL push). The literal value is then loaded directly into the HL register pair. Note that the ACTIVE flag is set upon completion of this macro, since SAVE always marks HL as active.

The ADR macro in Figure 36c (right) is a utility macro which is 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 upon 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 invocations of ADR shown to the left below produce the machine code to the right below.

```
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
```

thus leaving the final address for the optionally indexed variable in the HL register pair. Note that the code within the ADR macro could be improved slightly in the case that a constant offset is provided. That is, the invocations to the left below could produce the machine code shown to the right below by redefining the ADR macro.

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

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

It is a worthwhile exercise for the reader at this point to redefine ADR to generate this improved machine code sequence.

The VAL and STO macros are shown in Figure 36c (right) which load a variable value to the stack, or store the top of stack value to memory, respectively. Note that ADR is used to construct the address of the variable whenever optional indexing is specified. Otherwise, an LHLD or SHLD is used to directly access the variable. Again, slight improvements in generated code could be obtained when only a constant offset is provided with no variable index.

Note that 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.

Figure 36d contains a listing of the remainder of the "DSTACK.LIB" macro library. The SUM opcode shown on the left 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. Note that ACTIVE is true at this point, since REST always leaves the flag set to true.

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

Referring to Figure 36d (left), 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 will be a considerable amount of inline machine code for the operation. Thus, it is a useful exercise for the reader to redefine LSR so that it generates an inline subroutine to perform the shift operation for values of LEN which are sufficiently large to warrant the subroutine call. Although this will require a subroutine set up and call, the amount of generated code could be reduced significantly for programs which 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 which 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. Note that 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 generally count on the condition of the carry flag. Further, the 8080 A register contains the least significant difference between DE and HL, hence 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). Although Nachtflieger designers intend to include these opcodes in the KDF-12, it may be a worthwhile exercise for the reader to implement these additional macros.

The DUP opcode in Figure 36d (bottom left) first ensures that the HL register pair is active, then duplicates this value by pushing the HL pair to the 8080 stack, thus 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 in Figure 36d. The BRN macro simply translates to a jump instruction in the 8080 while the XIT is slightly more complicated. 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 which 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 (recall that the SIZ macro saves the value of SIZE), followed by the declaration of the label "STACK" at the base of this reserved area. Referring back to Figure 36c (middle left), 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 had not been included, then the label "STACK" would not appear (unless used in the KDF-11 program), and the "IF STACK" test would produce an undefined operand (U) error. Further, if the XIT operator had been used, but the SIZ had not, then the statement "DS SIZ*2" within XIT would produce an undefined operand message. Although these tests are by no means complete, they will detect the most common errors.

Figure 36d (right) also contains the definitions of both the RDM and WRM opcodes, based upon 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 to either "A-D INPUT" for the RDM opcode, or "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 argments 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 of one of ADC0, ADC1, ADC2, or ADC3 which is included in the trace message. If the % were omitted, only the name, not the value, of the input port address would be printed. Following the output message, UGEN is invoked to ensure that the utility subroutines have been included inline. The call to @IN allows the programmer to type a hexadecimal value for the simulated A-D input value, which 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 then displayed at the debugging console if DEBUGP is true, and then sent to the appropriate memory mapped output location.

One particular application of the emulated KDF-11 machine shows the power of this particular 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 which 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, the total distance travelled 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 particular 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 particular case is to collect the instantaneous velocities during a single cut, and hold these values for analysis as the tool returns to its starting postition. Figure 37 shows a KDF-11 program which includes the data collection phase, as well as an analysis phase described below.

The data collection phase of Figure 37 occurs between the labels MOVE? and COMP, while the analysis phase is found between labels COMP and ENDF. Note that the program is bounded by the SIZ operator at the beginning, along with the XIT operator at the end, followed by DCL opcodes which reserve data areas. This particular program also includes debugging PRN, DMP, TRT, and TRF opcodes for checking out the program.

Referrring to the DCL statements at the end of Figure 37, the "vector" V is declared with length 100 (double bytes), which will hold the collected velocities, while 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 non-zero velocity, waiting for the tool head to begin its travel. When the first non-zero 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), and so-forth, until a zero value is read, indicating the tool has ended its travel.

Referring to Figure 37, 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

```
                MACLIB  DSTACK  ;STACK MACHINE SIMULATION
0000            SIZ     50      ;50 LEVEL STACK
0103            TRT     P       ;TURN ON PRN TRACE
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     <VALUE 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)?

        ;       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
```

Figure 37. Program for Tool Travel Computation.

by a store into the variable I. In order 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.

Following the MOVE? label, A-D port 0 is read and examined for the first non zero value. Each time the port is read it is stored into the temporary variable X, then reloaded and examined for a zero value. Since 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 which 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 which takes place in the analysis section of Figure 37 is the computation of the distance travelled by the tool through this interval. In particular, Nachtflieger engineers have determined that it is sufficient to compute the distance travelled by the tool using the "trapezoidal rule" which approximates the actual distance by summing the average of each adjacent pair of velocites. 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 wouldn't occur in practice), then

$$V_1 = V_2 = \ldots = V_n = 256,$$

and the summing formula given above reduces to 256 * n. Given the example above where n = 50 ms, the above formula produces the value 1.280 mm, as given earlier. In general, the velocity values will not be constant, hence the numerical integration given by the trapedzoidal rule is used to obtain an approximation.

The KDF-11 instructions shown in Figure 37 between the COMP and ENDF labels perform the numeric integration given by the trapedzoidal rule. In general, 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 which 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 travelled is written to D-A output port zero. The output value is sent to external instrumentation which processes the result and displays the distance travelled in a form which is readable by the tool operator.

Note that debugging statements have been placed throughout the program which can be used to trace the program execution. Figure 37 also contains TRT operators which have enabled trace code generation, and thus this particular program, although longer than the final production version, can be used to follow execution under CP/M.

Figure 38 shows the execution of the program of Figure 37 under DDT. The messages printed at the debugging console are a result of the PRN opcodes distributed throughout the original program which 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. In order 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 Figure 38 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 which 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.

Upon completion of 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.

Figure 39 is also provided as an example of the listing which is produced when all machine code operators are traced. Although the source program listing is not shown, it is identical to Figure 37 except that the TRF T opcode is removed. Since the complete trace is quite extensive, only a partial execution is shown in Figure 39.

In summary, Nachtflieger MW has derived several benefits from their emulation of the KDF series stack machines. First, there is very little cost involved in designing

```
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
VALUE 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
TOTAL= 0220
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
```

Figure 38. Sample Execution of "Distance" using DDT.

```
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 045E 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
```

Figure 39. Partial Listing of "Distance" with Full Trace.

and altering their machine architecture. In fact, current prices for 8080 microcomputers may 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. That is, given that a higher performance or less expensive processor becomes available to Nachtflieger, the existing programs can be used intact by only changing the macro definitions for each of the KDF opcodes and reassembling using MAC or an equivalent macro processor. Lastly, machine emulation through macro defined operation codes offers a distinct advantage over interpretive approaches since 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, while 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 which is included for each and every program. Thus, for a wide variety of microcomputer applications, machine emulation through macro defined op-codes offers distinct advantages over alternative approaches.

9.3. Program Control Structures.

Macro facilities can be used to provide program control statements which resemble those found in many high-level languages. In general, program control statements allow boolean tests and conditional branching based upon the outcome of the boolean test. Further, label names which would normally be provided by the programmer as the destination of a branch are automatically generated for the particular statement.

In the paragraphs which follow, three typical control statements are presented which allow simple conditional grouping (WHEN-ENDW), controlled iteration (DO-ENDDO), and case selection (SELECT-ENDSEL). In all three cases, the intention is to define program control facilities which allow well-structured programming, resulting in programs which are easier to write, debug, and maintain.

Two libraries are first introduced in order to provide a foundation for further discussion. The I/O library shown in Figure 40 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 Figure 40 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 on the diskette in the file "SIMPIO.LIB" and are used in the examples which illustrate the control structures.

The second library used in the control structure examples is given in Figure 41. Collectively, these macros define a number of boolean operations which are performed upon 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 or Equal
GTR	Greater Than

```
;       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
```

Figure 40. Simple I/O Macro Library.

```
test?   macro   x,y
;;      utiltity 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
;
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,v,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
;
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
```

Figure 41. Macro Library for Simple Comparison Operations.

In all cases, the macros accept three actual parameters, consisting of two data values involved in the test (X and Y), along with a program label which receives control if the boolean test produces a true value (TL). The first operand X can be a labelled memory location containing an 8-bit value, and Y can be either a labelled 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 labelled memory locations ALPHA and BETA (defined by a DS or DB statement), and transfers to the program step labelled by TRUECASE if ALPHA contains a value less than the value stored at BETA. The invocation

```
LSS     ,BETA,TRUECASE
```

is similar, but 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.

Note that 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 Figure 41 which 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 non-numeric 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 diskette in a file named "COMPARE.LIB" for use in examples which follow.

Figure 42 shows an example of a program which uses both the SIMPIO and COMPARE libraries. The purpose of this program is to successively read console characters and print messages based upon the character which is 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 is used to determine if lower-to-upper case translation is required (assuming the input is alphabetic). If X is numerically less than 61H, which is the value of an upper case "A," then control transfers to the label NOTRAN. Otherwise, the character is loaded to the accumulator, the "upper 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.

In comparing each letter, the macro NEQ is invoked with the first argument corresponding to the character typed at the console (X), while the second argument corresponds to the letter to match. Note that the "%" operator is used in each case

```
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
```

Figure 42. Single Character Processing using COMPARE.

to produce the numeric value of the character. This is necessary since 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 at which time it returns to the console command processor. The intention here is to show the use of boolean tests used by the control structure macros which follow.

Figure 42b 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.

The macro library shown in Figures 43a and 43b, called NCOMPARE, expands upon the basic relational macros by allowing a "false branch" option. That is, each macro accepts four arguments: the X and Y operands, as before, as well as a "true label" (TL) and "false label" (FL). It is assumed that either the TL or FL will be supplied in any particular 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 as well as 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. One should note that 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 Figure 43a 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 will be used within the control structures which are described below.

Figure 44a gives an example of the use of the NCOMPARE library within a particular program. This program is similar to the previous example, but instead checks to insure that alphabetic translation only occurs 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 the equivalent decimal value 97). Since 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 and thus 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.

Figure 44b shows the GEQ-GTR section of the program of Figure 44a with full macro trace enabled (see Assembly Parameters). The trace in this figure shows the transition from GEQ to the LSS operator, substituting the FL label in the place of the TL label. Again, the macro library statements are not shown, and the listing following the NOTRAN label is not present.

```
                ;       . . .
                ;
                 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
                ;       . . .
```

Figure 42b. Partial Trace of Fig 42a with Macro Generation.

```
;       macro library for 8-bit comparison operation
;
test?   macro   x,y
;;      utiltity 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
;
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
```

Figure 43a. Expanded NCOMPARE Comparison Operators.

```
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
;
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
```

Figure 43b. Expanded NCOMPARE Comparison Operators (Con't).

```
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
```

Figure 44a. 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
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:
```

Figure 44b. Segment of Fig 44a with "+M" Option.

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 meaning of the WHEN-ENDW group is as follows: 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 ". . ." 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 which "count" WHEN-ENDW groups and generate branches and labels at the proper levels in the structure.

Figure 45 shows the WHEN macro library, consisting of four macros GENWTST (generate WHEN test), GENLAB (generate label), WHEN (beginning of WHEN group), and 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 upon 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 WHEN's which have occurred without corresponding ENDW's.

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. Note that 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 since the TL argument is empty. Again referring to the body of the GENWTST macro in Figure 45, note that 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.

Referring back to the body of the WHEN macro in Figure 45, 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 which redefine themselves, the outer macro definition of WHEN invokes the newly created WHEN macro before exit.

Upon encountering the an 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, while the second argument is the evaluated parameter %WLEV corresponding to the current ENDW label. If only one WHEN statement had been encountered, for example, the value of WLEV would be zero, and thus GENLAB would produce 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.

As an example of the use of WHEN-ENDW, Figure 46a shows a sample program which resembles the previous character scanning function, but uses the WHEN group in the 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 which follow to be executed when X is greater than or equal to 61H (lower case "a") and skipped otherwise. Further, the four WHEN groups which follow each test for the specific characters A, B, C, or D. If an "A"

```
;       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     Ø       ;;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 Ø (not checked)
        endm
```

Figure 45. Macro Library for the WHEN Statement.

```
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
                ;
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
```

Figure 46a. Sample WHEN Program with "-M" in Effect.

is typed, the corresponding WHEN group is executed, 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.

Figure 46b shows the same program with full macro trace enabled. This particular 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 for the reader to determine that the nesting rules for WHEN groups are properly stated, and that the restriction on nested parallel groups is, in fact, necessary.

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 are executed 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 are executed 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 Figure 47. In general, 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. Note that the only essential difference in the DOWHILE and WHEN groups is that the location of the DOWHILE test must be labelled and a JMP instruction must be generated to this label at the end of each group.

Referring to Figure 47, GENDTST (generate DOWHILE test), GENDLAB (generate DOWHILE label), and GENDJMP (generate DOWHILE jump) are all "label generators" used in the macros which follow. Similar to the WHEN macro, DOWHILE uses the counters DOCNT and DOLEV to keep track of the number of DOWHILE groups which have been encountered along with the current DOWHILE level, corresponding to the number of unmatched DOWHILE's. 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 which will eventually be generated by the ENDDO macro. Finally, the DOWHILE macro increments the DOCNT counter in preparation for the next group.

The ENDDO macro in Figure 47 first generates the JMP instruction back to the DOWHILE test, using the GENDLAB utility macro, and then produces the ENDDn label which becomes the target of the jump on false condition. The form of the expanded macros for one nested level thus becomes:

```
                ;       . . .
                ;
                ;       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
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
                ;       . . .
```

Figure 46b. Partial Listing of Fig 46a with "+M" Option.

```
;       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
```

Figure 47. Macro Library for the DOWHILE Statement.

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

Figure 48a shows an example of a program which 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 which repetitively execute in search for 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. In this case, control continues through the ENDW's to the ENDDO and back to the DOWHILE header. In this case, the "DOWHILE X,NEQ,STOP" produces a false condition, and control transfers to the "XRA A" instruction following the ENDDO.

Referring again to Figure 48a, a second DOWHILE-ENDDO group is executed which 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 which 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 interation, since the character sequence within the brackets is empty. Following the WRITE statement, X is incremented by one, thus 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.

Figure 48b shows a portion of the program of Figure 48a, with partial macro trace enabled. Note in particular that this trace does not show the generated labels ENDD1 and DTEST1 since 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 by noting that the "JNC ENDD1" produces the destination address "01FF" corresponding to the "RET" statement, while 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

```
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
```

Figure 48a. 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
```

Figure 48b. Partial Listing of Fig 48a with Macro Generation.

```
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 denote groups of statement 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 is executed. 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, however. That is, a SELECT-ENDSEL group cannot occur within a statement-set enclosed within an encompassing SELECT-ENDSEL group. As a convenience, the variable following the SELECT can be omitted in which case the current 8080 accumulator content is used to select the proper case.

Figures 49a and 49b show the SELECT macro library which 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 which 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 which marks the end of the SELECT group number n. Upon encountering the end of the group, a "select-vector" is generated which 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 particular 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
```

```
;       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
```

Figure 49a. 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
```

Figure 49b. Library for SELECT Statement (Con't).

```
        . . .
CASE0@n:
        statement-set-n
        JMP   ENDS0
SELV0:
        DW    CASE0@0
        DW    CASE0@1
        . . .
        DW    CASE0@n
ENDS0:
```

Figure 49a contains the label generators GENSLXI (generate SELECT LXI), GENCASE (generate case labels, GENELT (generate select vector element), and GENSLAB (generate SELECT label). Figure 49b contains the macro definitions for SELNEXT (select next case), SELECT, and ENDSEL. Referring to Figure 49b, 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. Note that 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 zeroes the ECNT variable which counts the cases until ENDSEL is encountered.

SELNEXT, shown at the top of Figure 49b, is invoked by the programmer 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 Figure 49b, GENCASE is again invoked 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 which have the case label addresses as operands.

Figure 50a gives an example of a simple program which uses two SELECT groups. The first SELECT group executes one of five different MVI instructions based upon 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 Figure 50a is used only to illustrate the generated control structures, and does not itself 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.

Figure 50b shows a segment of the previous program with generated macro lines. Note the case selection code following "SELECT X" and the selection vector at the end of the listing.

Figure 50c 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

```
                        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
```

Figure 50a. 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
```

Figure 50b. Segment of Fig 50a 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
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
```

Figure 50c. Segment of Fig 50a with "+M" Option.

Figure 50a. The listing has been edited to remove the case selection code, which is listed in Figure 50b, as well as the code generated for case number 2. Figure 50c should be cross-referenced with the SELECT macro library given in Figures 49a and 49b if confusion remains as to the actions of these macros.

It is now possible to show a complete program which uses the WHEN, DOWHILE, and SELECT groups. Figure 51 shows a program which is similar in function to a more complicated program which interacts with the console in executing single character input commands. In fact, the two CP/M programs ED and DDT both take this general form (see the ED and DDT Users Guides for details). That is, a single letter is used to select a single action which may correspond to an edit request in the ED program, or a debug request in DDT. Upon completion of each command, control returns back to the main loop to accept another single letter command.

The program given in Figure 51 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 which encompasses nearly the entire program. The DOWHILE group is controlled by the X,NEQ,%'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. In order to ensure that the letter is in the proper range of values, two WHEN groups follow which convert illegal values to the letter E, which will subsequently produce an error response.

Following the WHEN tests in Figure 51, 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 which 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 parameter mechanisms which pass actual values to subroutines for assignment to formal parameters can also be defined with macro libraries. Note also that it would be relatively easy to include control structures for the stack machine given in the previous section, thus allowing machine independent programming of control structures as well as 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 <TYPE 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
```

Figure 51. Program using WHEN, DOWHILE, and SELECT.

9.4. Operating Systems Interface.

In a general-purpose computing environment, macros are often used to provide systematic and simplified mechanisms for programmatic access to operating system functions. Throughout this document, the examples have shown various low-level calls to the CP/M operating system which implement function 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 set-ups and calls which perform the function.

The purpose of this section is to introduce more comprehensive operating system interface macros, and specifically show a sample macro library which allows simplified diskette file operations for sequential "stream" input/output operations. The principal macros of this library which allow file access are listed below:

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

Before introducing the macro library which 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,buffaddr
```

where the individual parameters of the FILE macro describe a particular 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 file name 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 (up to eight character) file name of the diskette file being accessed; if "1" or "2" is specified, then the first or second default file name is used, respectively

filetype - the (up to three character) file type of the file being accessed; if "1" or "2" has been specified for the filename parameter and an empty filetype is given,

then the file type is taken from the selected default file name, otherwise the type 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 diskette sector size; if the rounding produces a result which is too small, or if the parameter is empty, then only one sector is buffered.

buffaddr - the address of the buffer area to be used during accesses to this file; if empty, then the buffer address is assigned automatically

The FILE statement

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

for example, sets up the file "NAMES.DAT" on diskette drive A for subsequent access. Internal to the program, this file will be referenced by the name ZOT. Further, the buffer address is assigned automatically, and the buffer size is set to one sector (normally 128 bytes). In general, larger buffers are useful in minimizing rotational delay on the diskette 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 is aborted. An output file can be created using the statement

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

for example, 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 diskette 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 CP/M Interface and
            Alteration Guides for READER entry point definition)
GET  ZOT  - read one character from the file given by the in-
            ternal name ZOT (i.e., 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 control-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 diskette 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
```

The following PUT invocations perform the functions shown to the right below:

```
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
            whose internal name is ZAP (i.e., the ADDRESS.DAT
            file in the above example)
```

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

The FINIS statement is used to close 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 file name is permanently recorded on the diskette for future access. The form of the FINIS invocation is

```
FINIS       filelist
```

where "filelist" is a single internal name which appeared previously in a file statement, or a list of such file names enclosed within broken left and right brackets, and separated by commas. Although it is not necessary to close input files with the FINIS statement, it is good practice, since the file close operation may 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 diskette directory;  in the above example,
              the ADDRESS.DAT file is closed.
```

The ERASE macro allows programmatic removal of a diskette 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" which requires less program space than an "infile" or "outfile" parameter. Specific cases of the ERASE statement will be given in the examples which follow. In the simple case

```
ERASE  ZOT
```

however, the file NAMES.DAT would be removed from the diskette, given the previous FILE statement which defines ZOT.

The DIRECT macro is used to search for a specific file on the diskette. 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 diskette. In both the ERASE and DIRECT macros, the file identifiers can reference file names and types with embedded "?" characters, similar to the normal CP/M "DIR" command, where the question mark will match any character in the file names being scanned. The macro invocation

```
DIRECT  ZAP
```

for example, returns a non-zero flag if the file ADDRESS.DAT is present, and a zero flag 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 which have appeared in previous FILE statements. The rename macro changes the file name given by newfile to the file name given by oldfile. Similar to the ERASE and DIRECT macros, the file identifiers "newfile" and "oldfile" must appear in previously executed FILE statements, but may have a mode of "setfile" if they are not used in GET or PUT macros. If the drive names for the oldfile and newfile differ, then the drive name of the 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.

Figure 52 shows the use of the FILE, GET, PUT, and FINIS macros in a working program. The purpose of this program is to read an input file, specified at the console command processor level as the first file name, and translate each lower case alphabetic character to upper case. The output is sent to the file given as the second parameter at the command level. Given that this program has been assembled, loaded, and stored as "CASE.COM" on the diskette, a typical execution would be

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

```
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
                ;
                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
```

Figure 52. Lower to Upper Case Conversion Program.

which causes the CASE.COM file to be loaded and executed in the transient program area. Before execution, the console command processor passes LOWER.DAT as the first default file name, and UPPER.DAT as the second file name (see the CP/M Interface Guide for exact details). Referring to Figure 52, the CASE program begins by intializing the stack pointer to a local stack area in preparation for subsequent subroutine calls which occur within the various macros in the SEQIO macro library. The first default file name is then taken as the SOURCE file, as defined in the first FILE macro. The second FILE statement assigns the second default file name 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 will appear at the console if the files cannot be accessed or created in the FILE statements.

The CASE program's main loop is shown in Figure 52 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. Since the data file is assumed to consist of a stream of Ascii characters, the end of file is detected when a control-Z is encountered. When this character is found, control transfers to the label ENDCOPY where the DEST file is closed using the FINIS macro. Again note that errors in writing or closing the DEST file will produce an error message at the console, and the program execution will be aborted immediately. Upon completion of the program, control is returned 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 Figure 52. 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 Figure 52, with a statement of the form

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

which produces the equated value 1270H at the left of the listing. In this particular case, the memory area beyond 1270H is not used by the program.

The macro library for SEQIO is shown in Figures 53a, 53b, 53c, 53d, and 53e, which constitute the most comprehensive macro library shown in this manual. The particular macro library contains an instance of nearly every macro facility available in MAC, and thus it is useful to read and understand the operations contained in the listing. The discussion below of SEQIO outlines the general functions of each macro, but it is left to the reader to investigate the exact operation of the library.

The SEQIO segment shown in Figures 53a and 53b contain generally useful equates and utility macros. The label FILERR at the beginning becomes the destination of transfers upon encountering a file operation error and, since this is a SET statement,

```
;       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
;
;       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
;;     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
```

Figure 53a. Sequential File Input/Output Library.

```
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
@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
;;
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
;
```

Figure 53b. Sequential File I/O Library (Con't).

```
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
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
emsg:   db      cr,lf
        db      'disk full: &FID'
        db      '$'
        endif
```

Figure 53c. Sequential File I/O Library (Con't).

```
;;
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
;;
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
```

Figure 53d. Sequential File I/O Library (Con't).

```
;
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
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
;
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
;;     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
```

Figure 53e. Sequential File I/O Library (Con't).

may be changed in the user's program to "trap" error conditions rather than rebooting. The use of FILERR is apparent throughout the macro library.

The equates which follow define the usual BDOS entry points and functions, along with the diskette sector size (@SECT), and special non-graphic 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.

Referring again to Figure 53a, FILLNAM is a utility macro which is used in the construction of a file control block. In particular, it accepts a file name or file type along with a field size and builds a sequence of DB's which fill the name or type field with padded blanks. FILLDEF is again a utility macro similar to FILLNAM, but 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 file name (first 8 characters) or default file type (following 3 character field). Note that 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) upon subsequent invocations.

The last macro definition shown in Figure 53a is FILLNXT which is used to initialize two assembly time variables: @NXTB and @NXTD. @NXTB is used to count the accumulated size of buffers as they are automatically allocated in the FILE statement, while @NXTD is used to count files in the FILE macro for later reference in GET and PUT statements. They are included within a macro so that they will be 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. Note that FILLNXT then redefines itself as an empty macro so that subsequent FILE invocations do not reset the two counters.

A major utility macro, called FILLFCB, is shown in Figure 53b. The primary purpose of this macro is to construct a file control block in the CP/M standard format, where FID is the file identifier, DN is the disk name, FN is the file name, FT is the file type, BS is the buffer size, and BA is the buffer address, as described in the FILE statement above. Recall that some of these parameters may 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 either default name 1 or 2 is to be selected for the file. Note that the IRPC loop involving ?C will result 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 reader may wish to cross-reference the file control block format shown in the CP/M Interface Guide for exact formats.

The remainder of the FILLFCB macro, shown in the lower half of Figure 53b, 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's 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 which will subsequently

hold the buffer index to the next character to get or put in the file. All of these values will be used in the file operations which occur later.

The principal file access macro, called FILE, is shown in Figure 53c, and is used to set up the file control block, buffers, and access subroutines for a particular 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, is present to indicate 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 above. The FILE macro then generates either the label GET&FID or PUT&FID for input and output files, respectively, followed by a subroutine which GET's a single character or PUT's a single character for this file.

In general, 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 which follows the label PSUB in Figure 53d is used to initialize the file pointers to the proper positions 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" in the sense that 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, ERASE, DIRECT, RENAME, GET, and PUT macros are shown in Figure 53e. The FINIS macro, shown on the left, serves to empty the output buffers and close the file for output. Input files are skipped since no actions need take place to close an input file. The main purpose of the FINIS macro is to fill the remaining buffer segment (one sector size) with EOF's, then write 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 only for a single file given by the file identifier FID. In the case of the DIRECT macro, the non-zero flag is set if the file exists. No prechecks are made to see if the file exists before the ERASE operation takes place, although erasing a non-existant file is of no consequence. The DIRECT macro can, of course, be used to check if a file exists before the ERASE is executed if deemed necessary by the programmer.

The RENAME macro shown in Figure 53e (right) 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 has the effect of renaming the OLD file to the NEW file name. 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 16 bytes) into the second half of the old file name in the form required for a rename operation under CP/M (see the CP/M Interface Guide). The BDOS is then called to perform the rename function. Note again that there is no check to ensure the old file exists before the rename takes place.

The GET and PUT macros shown in Figure 53e 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 or 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, while the save/restore sequence is performed within the PUT&FID subroutine if the destination is a diskette file.

Figures 54a, 54b, and 54c show the full expansion of a segment of the case conversion program of Figure 52 (using the "+M" assembly parameter). Figure 54a shows 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 Figure 54b, along with the beginning of the GETSOURCE subroutine. Note that the conditional assembly ignores the portions of this FILE macro expansion which are related to output files while including the machine code for the input SOURCE file. In each case, the "&FID" labels result in names with the prefix or suffix "SOURCE" in order to associate the generated labels with this particular internal name. Figure 54c contains the end of the PUTSOURCE subroutine, followed by the machine code which initializes the file control block fields and buffer pointer. Upon completion of the FILE macro, the SOURCE file is ready for access. In particular, 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.

In order 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 diskette files.

The PRINT program, shown in Figure 55, is executed under the console command processor by typing

```
PRINT  filename
```

where "filename" is the name of a previously assembled program. PRINT assumes that there is a "PRN" file on the diskette, and possibly a "SYM" file on the same diskette drive. The PRN file is first printed, with a form feed at the top of each 56 line

```
                        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
    +                   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     H
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
    +                   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
```

Figure 54a. Sample FILE Expansion Segment.

```
    +                   IF      SOURCETYP<=2
    +                   FILLNXT
0000+#              @NXTB  SET     0
0006+#              @NXTD  SET     @LST+1
    +               FILLNXT        MACRO
    +                   ENDM
    +                   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
    +               ??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
```

Figure 54b. Sample FILE Expansion Segment (Con't).

```
    +           ??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
    +           ??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             XHLD    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
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
```

Figure 54c. Sample FILE Expansion Segment (Con't).

page. If the SYM file exists, it is also printed using the same formatting. If the files are sucessfully printed, they are both erased from the diskette.

Referring to Figure 55, the PRINT program begins by saving the console processor's 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 which specifies the internal name PRINT, and obtains the file name from the console command line. The file type, 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 diskette.

As shown on the left of Figure 55, 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 nor does it 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 proceeds to perform another page eject, and then opens the SYM file for access. It should be noted that the third FILE macro (Figure 55, left) accesses the SYM file using the internal name SYMBOL, but shares the buffer areas of the PRINT file. This is possible since the PRINT file has been erased at this point in the program and thus the buffers are available for use.

If the SYM file is present, the program loops between the SYCYLE (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 removed from the diskette. 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, thus completing execution of the PRINT program.

The next program, called MERGE, is somewhat more complicated. The purpose of the MERGE program is to accept two file names as input, taking the general command form

MERGE filename

where "filename" is the name of a master file, with assumed file type of MAS, as well as an update name with assumed file type UPD. The files consist of text files with varying length records, starting with a six character numeric "sequence number" followed by textual material, and terminated 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 purpose of the MERGE program is to read these two files and "shuffle" 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

```
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
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:
                        ;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
```

Figure 55. Program for Line Printer Page Formatting.

according to numeric order, and any update record which 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 extension MBK (master back-up), the original update file is renamed to the type UBK (update back-up), and the newly created file becomes the new MAS file. In this way, the operator can return to the backup files in case of error so that the source data is not destroyed.

The MERGE program is shown in Figures 56a, 56b, and 56c. Utility subroutines are listed first in Figure 56a, including the DIGIT subroutine which tests for valid decimal digits in sequence numbers. The IRPC which 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 (see the $+PRINT and $-PRINT inline parameters) to keep the listing short. In general, these two READ subroutines fill their respective sequence number buffers from the input source so that the merge operation can take place based upon 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 utility subroutines shown in Figure 56b include SEQERR, WRITESEQ, and COMPARE. The SEQERR subroutine reports an error condition when a non numeric character is detected in the sequence number field. Although the error reporting is somewhat spartan, sequence errors are easily found using the TYPE command on the master or update file. The WRITESEQ subroutine sends the buffered sequence number addressed by HL to the new file. WRITESEQ is called whenever the source for the next record has been determined. The COMPARE subroutine is used to determine 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 in Figure 56b 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 name TEMP and type $$$, but will be altered upon completion of the program to become the master file.

The merge operation proceeds in Figure 56b as follows. First the READU and READM subroutines are called to fill the sequence number buffers. The loop between MERGE and ENDMERGE in Figure 56c 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 in Figure 56c where master record is deleted. Alternatively, the COMPARE subroutine will cause control to transfer to the MASLOW label when

```
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
                ;
                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
                ;
                EOF&?F:         ;END OF FILE, SET SEQ# TO 0FFH
                        MVI     A,0FFH
                        STA     ?F&SEQ          ;SEQ# SET TO FF
                        RET
                        ENDM
                ;
```

Figure 56a. File Merge Program.

```
                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         WRIT0:  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     WRIT0           ;FOR ANOTHER CHAR
01B1 C9                 RET
                ;
                ;       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 0FFH
01BE FEFF               CPI     0FFH            ;END OF FILE
01C0 C8                 RZ                      ;BOTH ARE 0FFH
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 #
                ;
```

Figure 56b. File Merge Program (Con't).

```
                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 0FFH
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
                ;
                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
```

Figure 56c. File Merge Program (Con't).

the master sequence number is low. 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 (end of file detected 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 backup (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 is stopped.

Figure 57 shows an example of the files which are involved in a typical merge operation. In this application, the sequence numbers control the ordering of a list of names which is updated periodically. The NAMES.MAS file is the original master, which will be updated by merging the NAMES.UPD file, also shown in the figure. The merge operation is initiated by typing

```
MERGE  NAMES
```

and, upon completion, produces the new NAMES.MAS shown to the right in Figure 57.

The SEQIO library is typical of the interface one 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, one can construct more comprehesive libraries for random access based upon this library.

NAMES.MAS

```
000100 ABERCROMBIE, SIDNEY
000200 CARLSBAD, YOLANDA
000300 EGGBERT, EBENIZER
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, ULYSSIS
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, EBENIZER
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, ULYSSIS
001000 TWITSWEET, LADNER
001010 UMPLANDER, XAVIER
001090 VERANDA, VERONICA
001100 WILLOWANDER, PRATNEY
001110 XYLOPH, ERHARDT
001200 YUPPGANDER, MANNY
001210 ZEPLIPPS, EGGERWORTZ
```

Figure 57. Sample MERGE Disk Files.

10. ASSEMBLY PARAMETERS

Assembly parameters can be included when the assembly begins to control various assembler functions. In general, the macro assembler is initiated with the name of the source file, followed by the assembly parameters, indicated by a preceding dollar symbol ($). The parameters are indicated by single controls which denote particular functions. The letter or digit shown to the left below corresponds to the function shown to the right.

A controls the source disk for the .ASM file
H controls the destination of the .HEX machine code file
L controls the source disk for the .LIB files (see MACLIB)
M controls MACRO listings in the .PRN file
P controls the destination of the .PRN file containing the listing
Q controls the listing of LOCAL symbols
S controls the generation and destination of the .SYM file
1 controls pass 1 listing

Any or all of the above parameters can be included. In the case of the A, H, L, and S parameters, they are followed by the drive name to obtain or receive the data, where the drives are labelled 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 which is not recorded. The following is a valid assembly parameter list following the MAC command and source file name

```
$PB AA HB SX
```

which 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 either + or - symbols which enable or disable their respective functions. These functions are listed below

+L list the input lines read from the macro library (see MACLIB)
-L suppress listing of the macro library (default value)
+S append the .SYM to the end of the .PRN output
-S suppress the generation of the sorted symbol table
+M list all macro lines as they are processed during assembly
-M suppress all macro lines as they are read during assembly
*M list only "hex" generated by macro expansions
+Q list all LOCAL symbols in the symbol list
-Q suppress all LOCAL symbols in the symbol list
+1 produce a listing file on the first pass (for macro debugging)
-1 suppress listing on pass 1 (default)

The following is an example of a valid assembly parameter list which 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.

Note that the M parameter can be optionally preceded by the "*" symbol which causes the assembler to list only macro generations which produce machine code, and is used to suppress the listing of the instructions which are produced (i.e., all positions beyond the hex fields are not listed). Under normal operation, the macro assembler lists only generations which 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 (i.e., all symbols beginning with "??" are not listed), and macro lines which 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 always 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 which begin with the characters "??" in addition to the normally 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 ASCII form.

```
MAC sample $AB HA PB +Q +S +L *M
```

assembles the sample.ASM file from drive B, 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, and the instructions which correspond to generated macro lines are not included (although generated machine code is listed).

In addition to the parameters shown above, the programmer 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 to the left below corresponds to the action given to the right.

$-PRINT	stops the 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

Since MAC allows each line to be optionally prefixed by a line number, the "$" control can be included directly following this line number, if desired.

11. DEBUGGING MACROS

In completing the discussion of the macro assembler, it is worthwhile considering common debugging practices 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 evolve. Further, if errors occur in the macro generations after a small portion of the macro set has been improved, it is most likely the case that the error is being caused by the macros which were changed.

In the case of the Hornblower Highway System macro libraries, for example, iterative improvement was used to evolved the final macro library. In particular, only the simplest macros were first implemented, including the SETLITE, TIMER, and RETRY macros (see Section 10.1). 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 where individually written, added, and tested, resulting in the final macro library.

At each step, the programmer can use the various assembly parameters to control the debugging information. If the macro generations are not producing the proper machine code, it may 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, the programmer 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, the programmer can use the "+L" parameter when MAC is started to cause the libraries to be included in the listing as they are read.

As a final consideration, it may be necessary to enable the first pass listing of the assembly language using the "+1" parameter. In this case, MAC will list the program as it is being read on the first pass as well as the second pass. Note, however, that the listing will contain spurious error messages on this pass which may disappear on the second pass. The principal purpose of the first pass listing parameter is to allow the programmer 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 particular macro expands improperly, and the source of the error is not evident after examining various traces, it may 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.

12. SYMBOL STORAGE REQUIREMENTS

The maximum program size which 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 upon 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. In any case, the symbol table size is dynamically determined by MAC upon startup, and fills as symbols are encountered. In order 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 which 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 somewhat more complicated to compute. The general form is given by

$$M = L + 7 + H + T$$

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_i is the length of the i^{th} 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, instead of their multi-character 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. In fact, 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 (crlf's).

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 parameter header storage requirement of

$$H = P_1 + P_2 + 2 = 3 + 5 + 2 = 10 \text{ bytes}$$

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 REPT's, IRPC's, or IRP's, although temporary storage in the symbol table is used while the groups are actively iterating. In particular, 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 which is returned upon completion of the macro expansion.

In any case, a symbol table overflow message will result 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. Note that the "percentage" of symbol table utilization is always printed at the console at the end of the assembly. The form of the printout is

0hhH USE FACTOR

where hh is a hexadecimal value in the range 00 to FF, where 00 results from a near empty table, and FF is produced for a nearly full table. The value 080H, for example, is printed when the symbol table is half full. The programmer should keep note of the use factor as a particular program is developed in order to guage the relative amount of free space as the program is enhanced.

In many of the examples shown in this manual, macros include inline subroutines which are generated at the first invocation and called upon subsequent invocations (see the TYPEOUT macro in Figure 10, 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 when the macro is invoked the first time, and thus are not generated by the macro.

13. ERROR MESSAGES

When errors occur within the assembly language program, they are listed as single character flags in the leftmost position of the source listing. The line in error is also echoed at the console so that the source listing need not be examined to determine if errors are present. The single character error codes are:

B Balance error: macro doesn't terminate properly, or conditional assembly operation is ill-formed.

C Comma error: expression was encountered, but not delimited properly from the next item by a comma.

D Data error: element in a data statement (DB or DW) cannot be placed in the specified data area.

E Expression error: expression is ill-formed and cannot be computed at assembly time.

I Invalid character error: a non graphic character has been found in the line (not a carriage return, line feed, tab, or end of file); re-edit the file, delete the line with the I error, and retype the line.

L Label error: label cannot appear in this context (may be a duplicate label).

M Macro overflow error: internal macro expansion table overflow; may be due to too many nested invocations or infinite recursion.

N Not implemented error: features which will appear in future MAC versions (e.g., relocation) are recognized, but flagged in this version.

O Overflow error: expression is too complicated (i.e., too many pending operators), string is too long, or too many successive substitutions of a formal parameter by its actual value in a macro expansion. This error will also occur if the number of LOCAL labels exceeds 9999.

P Phase error: label does not have the same value on two subsequent passes through the program, or the order of macro definition differs between two successive passes; may be due to MACLIB which follows a mainline macro (if so, move the MACLIB to the top of the program).

R Register error: the value specified as a register is not compatible with the operation code.

S Statement error: the fields of this statement are ill-formed and cannot be processed properly; may be due to invalid characters or delimiters which are out of place.

V Value error: operand encountered in an expression is improperly formed; may be due to delimiter out of place or non-numeric operand.

Several error messages are printed at the console indicating terminal error conditions which abort the MAC execution. Whenever possible, the disk drive name, followed by the relevant file name is printed with the message.

NO SOURCE FILE PRESENT: The source program file (.ASM) following the MAC command cannot be found on the specified diskette. Use the DIR command in the CCP to locate the source file.

NO DIRECTORY SPACE: The diskette directory is full. Use the ERA command of the CCP to remove files which you do not need. There are often superfluous .HEX, .PRN, and .SYM files which can be removed.

SOURCE FILE NAME ERROR: The form of the source file name is invalid, or not specified. The command form must be:

MAC filename $assembly parameters

where the "filename" is the (up to eight character) primary name of the source file, with an assumed file type of ".ASM" (which is not specified).

SOURCE FILE READ ERROR: The source file cannot be read properly by the macro assembler. Use the CCP TYPE command to display the file contents at the console.

OUTPUT FILE WRITE ERROR: An output file cannot be written properly, probably due to a full diskette. As in the directory full error above, use the CCP commands to erase unnecessary files from the diskette.

CANNOT CLOSE FILE: An output file cannot be closed. The diskette may be write protected.

UNBALANCED MACRO LIBRARY: A MACRO definition was started within a macro library, but the end of file was found in the library before the balancing ENDM was encountered. Examine the macro library using the TYPE command of the CCP, or use the "+L" assembly parameter, to ensure that the library is properly balanced.

INVALID PARAMETER: An invalid assembly parameter was found in the input line. The assembly parameters are printed at the console up to the point of the error.

Appendix

8080 CPU INSTRUCTIONS IN OPERATION CODE SEQUENCE

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
00	NOP	2B	DCX H	56	MOV D,M
01	LXI B,D16	2C	INR L	57	MOV D,A
02	STAX B	2D	DCR L	58	MOV E,B
03	INX B	2E	MVI L,D8	59	MOV E,C
04	INR B	2F	CMA	5A	MOV E,D
05	DCR B	30	- - -	5B	MOV E,E
06	MVI B,D8	31	LXI SP,D16	5C	MOV E,H
07	RLC	32	STA Adr	5D	MOV E,L
08	- - -	33	INX SP	5E	MOV E,M
09	DAD B	34	INR M	5F	MOV E,A
0A	LDAX B	35	DCR M	60	MOV H,B
0B	DCX B	36	MVI M,D8	61	MOV H,C
0C	INR C	37	STC	62	MOV H,D
0D	DCR C	38	- - -	63	MOV H,E
0E	MVI C,D8	39	DAD SP	64	MOV H,H
0F	RRC	3A	LDA Adr	65	MOV H,L
10	- - -	3B	DCX SP	66	MOV H,M
11	LXI D,D16	3C	INR A	67	MOV H,A
12	STAX D	3D	DCR A	68	MOV L,B
13	INX D	3E	MVI A,D8	69	MOV L,C
14	INR D	3F	CMC	6A	MOV L,D
15	DCR D	40	MOV B,B	6B	MOV L,E
16	MVI D,D8	41	MOV B,C	6C	MOV L,H
17	RAL	42	MOV B,D	6D	MOV L,L
18	- - -	43	MOV B,E	6E	MOV L,M
19	DAD D	44	MOV B,H	6F	MOV L,A
1A	LDAX D	45	MOV B,L	70	MOV M,B
1B	DCX D	46	MOV B,M	71	MOV M,C
1C	INR E	47	MOV B,A	72	MOV M,D
1D	DCR E	48	MOV C,B	73	MOV M,E
1E	MVI E,D8	49	MOV C,C	74	MOV M,H
1F	RAR	4A	MOV C,D	75	MOV M,L
20	- - -	4B	MOV C,E	76	HLT
21	LXI H,D16	4C	MOV C,H	77	MOV M,A
22	SHLD Adr	4D	MOV C,L	78	MOV A,B
23	INX H	4E	MOV C,M	79	MOV A,C
24	INR H	4F	MOV C,A	7A	MOV A,D
25	DCR H	50	MOV D,B	7B	MOV A,E
26	MVI H,D8	51	MOV D,C	7C	MOV A,H
27	DAA	52	MOV D,D	7D	MOV A,L
28	- - -	53	MOV D,E	7E	MOV A,M
29	DAD H	54	MOV D,H	7F	MOV A,A
2A	LHLD Adr	55	MOV D,L	80	ADD B

OP CODE	MNEMONIC	OP CODE	MNEMONIC	OP CODE	MNEMONIC
81	ADD C	AC	XRA H	D7	RST 2
82	ADD D	AD	XRA L	D8	RC
83	ADD E	AE	XRA M	D9	- - -
84	ADD H	AF	XRA A	DA	JC Adr
85	ADD L	B0	ORA B	DB	IN D8
86	ADD M	B1	ORA C	DC	CC Adr
87	ADD A	B2	ORA D	DD	- - -
88	ADC B	B3	ORA E	DE	SBI D8
89	ADC C	B4	ORA H	DF	RST 3
8A	ADC D	B5	ORA L	E0	RPO
8B	ADC E	B6	ORA M	E1	POP H
8C	ADC H	B7	ORA A	E2	JPO Adr
8D	ADC L	B8	CMP B	E3	XTHL
8E	ADC M	B9	CMP C	E4	CPO Adr
8F	ADC A	BA	CMP D	E5	PUSH H
90	SUB B	BB	CMP E	E6	ANI D8
91	SUB C	BC	CMP H	E7	RST 4
92	SUB D	BD	CMP L	E8	RPE
93	SUB E	BE	CMP M	E9	PCHL
94	SUB H	BF	CMP A	EA	JPE Adr
95	SUB L	C0	RNZ	EB	XCHG
96	SUB M	C1	POP B	EC	CPE Adr
97	SUB A	C2	JNZ Adr	ED	- - -
98	SBB B	C3	JMP Adr	EE	XRI D8
99	SBB C	C4	CNZ Adr	EF	RST 5
9A	SBB D	C5	PUSH B	F0	RP
9B	SBB E	C6	ADI D8	F1	POP PSW
9C	SBB H	C7	RST 0	F2	JP Adr
9D	SBB L	C8	RZ	F3	DI
9E	SBB M	C9	RET Adr	F4	CP Adr
9F	SBB A	CA	JZ	F5	PUSH PSW
A0	ANA B	CB	- - -	F6	ORI D8
A1	ANA C	CC	CZ Adr	F7	RST 6
A2	ANA D	CD	CALL Adr	F8	RM
A3	ANA E	CE	ACI D8	F9	SPHL
A4	ANA H	CF	RST 1	FA	JM Adr
A5	ANA L	D0	RNC	FB	EI
A6	ANA M	D1	POP D	FC	CM Adr
A7	ANA A	D2	JNC Adr	FD	- - -
A8	XRA B	D3	OUT D8	FE	CPI D8
A9	XRA C	D4	CNC Adr	FF	RST 7
AA	XRA D	D5	PUSH D		
AB	XRA E	D6	SUI D8		

D8 = constant, or logical/arithmetic expression that evaluates to an 8 bit data quantity.

D16 = constant, or logical/arithmetic expression that evaluates to a 16 bit data quantity.

Adr = 16-bit address.