The macro assembler, SWAP—A general purpose interpretive processor

by M. E. BARTON

Bell Telephone Laboratories
Naperville, Illinois

INTRODUCTION

A new macro assembler, the **SW**itching **A**ssembly **P**rogram (SWAP), provides a variety of new features and avoids the restrictions which are generally found in such programs. Most assemblers were not designed to be either general enough or powerful enough to accomplish tasks other than produce object code. SWAP may be used for a wide variety of other problems such as interpretively processing a language quite foreign to the assembler.

SWAP has been developed at Bell Telephone Laboratories, Incorporated, to assemble programs for three very different telephone switching processors. (SWAP is written in the IBM 360 assembly language and runs on the 360 with at least 256K bytes of memory.) With such varied object machines and the need to have all source decks translatable from the previously used assembler languages to the SWAP language, it is no wonder that the SWAP design includes many features not found in other assemblers. The cumulative set of features provides a powerful interpretive processor that may be used for a wide variety of problems.

DESCRIPTION

The source language is free field. Statement labels begin in column one. Operation names and parameters are delimited by a single comma or one or more blanks. Comments are preceded by the sharp sign (#), and the logical end of line is indicated by the semicolon (;) or physical end of card. A method is provided for user interpretation of other than this standard syntax; SWAP is currently being used as a preliminary version of several compilers.

Inputs

The SWAP assembler may receive its original input from a card, disc, or tape data set. The SOURCE pseudo-operation allows the programmer to change the input source at any point within a program. It is also capable of receiving input lines directly from another program, normally a source editor.

Outputs

Because the input line format is free field, the assembly listing of the source lines may appear quite unreadable. Therefore, the normal procedure is to have the assembler align all the fields of the printed line. The positions of the fields are, of course, a programmer option. There are several classes of statements that may be printed or suppressed at the programmer's discretion. In keeping everything as general as possible, it is natural that any operation, pseudo-operation, or macro may be assigned to any combination of these classes of statements.

In addition to producing the object program, which varies with different applications, and the assembly listing just described, SWAP has the facility to save symbol, instruction, or macro definitions in the form of libraries which may be loaded and used to assemble other programs.

Macro expansions and the results of text substitution functions are another optional output. The programmer completely controls which lines are to be generated and the format of these lines. These lines may be printed separately from the object listing or placed on card, disc, or tape storage. This optional output may be used to provide input to other assemblers, and in this way SWAP can become a pseudo-compiler for almost any language. This output can also be used to produce preliminary program documents from comments which were originally placed in the source program deck.

Variables

There are numerous types of variable symbols, such as integers, floating point numbers, truth value, and character strings. The programmer may change or assign the type of any symbol as he wishes. For this purpose, the type of a symbol or operation is represented by a character. Each variable symbol may have up to 250 user-defined attributes which are data associated with each symbol. In addition, each symbol represents the top of a push-down list which allows the programmer to make a local use of any symbol.

A string variable would be defined by the TEXT pseudo-operation:

```
VOWELS TEXT 'AEIOU'
```

while a numeric value is assigned by SET:

```
LIMIT SET 10
```

The 'functional' notation is used extensively to represent not only the value of a symbol attribute, but also to represent array elements and predefined or user-defined arithmetic functions. In the following statement:

```
ALPHA(SA) SET BETA(SB)+10
```

the ALPHA attribute of symbol SA would be assigned a value ten greater than the BETA attribute of symbol SB.

An array of three dimensions would be declared by the statement:

```
ARRAY CUBE(-1:1,3,0:2)=4
```

In this example, the range of the first dimension runs from −1 through +1, while the second dimension is from +1 through +3, and the third is from 0 through 2. Each element would have the initial value 4, but the following statement could be used to assign another value to a particular element of the array:

```
CUBE(-1,2,0) SET 5
```

An attribute, array, or function reference may occur anywhere that a symbol may be used in an arithmetic expression.

Expressions

The following is a hierarchical list of the operators allowed in expressions:

**	or	↑	exponentiation
*	and	/	multiplication and division
unary−	and	unary¬	negation and complement
+	and	−	addition and subtraction
=, >, <, ¬=	or	≠	
=>	or	≥	the six relational operators
=<	or	≤	
&	and	¬	logical AND and AND of complement
|	and	!	logical OR and EXCLUSIVE OR

( ), [ ], and { } may be used in the usual manner to force evaluation in any order.

Four particular rules apply to the use of these operations:

1. Combined relations $A\rho B\rho C$ are evaluated the same as the expression $A\rho B\&B\rho C$ where ρ is any relational operator.
2. Character strings in comparisons are denoted as quoted strings.
3. The type of each operand is used to determine the method of evaluation. (For example, the complement of an integer is the 32-bit complement while the complement of a truth value is a 1-bit complement.)
4. If a TEXT symbol is encountered as an operand in an expression, it is called an indirect symbol, and its value is the result of evaluating the string as an expression.

Predefined Functions

Several built-in or predefined functions are provided to aid in evaluating some of the more common expressions. The following is a partial list of the available functions:

E(EXP)	Results in 2 raised to the EXP power.
MAX($EXP_1, \ldots, EXP_n$)	Returns the maximum of the expressions EXP_1 through EXP_n.

STYP(EXP, C)	Returns the value of EXP, but the type of the result is the character C as discussed in the *Variables* section.
SET(SYMB, EXP)	Returns the value of EXP and assigns that same value to the symbol SYMB. This differs from the SET pseudo-operation in that the symbol is defined during the evaluation of an expression.

Programmer-defined functions

To allow the programmer to define any number of new functions, the DFN pseudo-operation is provided. The general form of a function definition is written:

$$\text{DFN}\quad F(P_1, P_2, \ldots, P_n) = A_1{:}B_1,\ A_2{:}B_2,\ \ldots,\ A_n{:}B_n$$

where F is the function name, the *Ps* are dummy parameter names, and the *As* and *Bs* are any valid expressions. These expressions may contain the *Ps* and other variables as well as other function calls which may be recursive.

To evaluate the function, the *Bs* are evaluated left to right. The result is the value of the A corresponding to the first B that has a value of true (or nonzero). The colons may be read as the word "if." A simple example would be the function:

$$\text{DFN}\quad \text{POS}(X) = 1{:}X>0,\ 0{:}X\leq 0$$

which returns the value 1 if its argument is positive; otherwise, the result is zero. If the expression B_n is omitted, it is assumed to be true. Another example is the following definition of Ackermann's function:

$$\text{DFN}\ ACK(M, N) = N+1{:}M=0,\ ACK(M-1, 1){:}$$
$$N=0,\ ACK(M-1, ACK(M, N-1))$$

Two features are provided to allow an arbitrary number of arguments in the call of a function. The first is the ability to ask if an argument was implicitly omitted from the call. This feature is invoked by a question mark immediately following the dummy parameter name. If the argument was present, the result of the parameter-question mark is the value true; otherwise, the value is false. For example, the function defined by:

$$\text{DFN}\quad INC(X, Y) = X+Y{:}Y?,\ X+1$$

would yield the value 7 when called by $INC(2, 5)$ since Y is present, but the value of INC(3) is 4 since an argument value for Y was omitted.

The other feature which allows an arbitrary number of arguments is the ability to loop over a part of the defining expression, using successive argument values wherever the last dummy parameter name appears in the range of the loop. This feature is invoked by the appearance of an ellipsis (. . .) in the defining expression. The range of the loop is from the operator immediately preceding the ellipsis backward to the first occurrence of the same operator at the same level of parentheses. As an example, consider the following statement:

$$\text{DFN}\quad \text{SUM}(X, Y) = A + X^{**}(Y+C) + \cdots$$

The range of the loop is from the + following the right parenthesis backward to the + between the A and the X. The call SUM(4, 1, 2, 3) would yield the same result as the following expression:

$$A+4^{**}(1+C)+4^{**}(2+C)+4^{**}(3+C)$$

The loop may also extend over the expression between two commas as the next example shows. A recursive function to do the EXCLUSIVE OR of an indefinite number of arguments could be defined by:

$$\text{DFN}\quad \text{XOR}(A, B, C) = A\neg B \mid B\neg A{:}\neg C?,$$
$$\text{XOR}(\text{XOR}(A, B), C, \ldots)$$

Sequencing control

The pseudo-operations that allow the normal sequence of processing to be modified provide the real power of an assembler. In SWAP, the pseudo-operations that provide that control are JUMP and DO. JUMP forces the assembler to continue sequential processing with the indicated line, ignoring any intervening lines. The statement:

JUMP .LINE

will continue processing with the statement labeled: .LINE. The symbol .LINE is called a sequence symbol and is treated not as a normal symbol but only as the destination of a JUMP or DO. Sequence symbols are identified by the first character, which must be a period. A normal symbol may also be used as the destination of a JUMP or DO, if convenient. The destination of a JUMP may be either before or after the JUMP statement.

The JUMP is taken conditionally when an expression is used following the sequence symbol:

JUMP *.XX*, *INC*>10 # IS IT OVER LIMIT

The JUMP to $.XX$ will occur only if the value of the symbol INC is greater than ten.

The DO pseudo-operation is used to control an assembly time loop and may be written in one of three forms:

```
DO  .LOC, VAR=INIT, TEXP, INC      (i)
DO  .LOC, VAR=INIT, LIMIT, INC     (ii)
DO  .LOC, VAR=(LIST)               (iii)
```

Type (*i*) assigns the value of INIT to the variable symbol VAR. The truth value expression TEXP is then evaluated and, if the result is true, all the lines up to and including the line with .LOC in its location field are assembled. The value of INC (if INC is omitted, 1 is assumed) is then added to the value of VAR and the test is repeated using the incremented value of VAR.

Type (*ii*) is the same as type (*i*) except that the value of VAR is compared to the value of LIMIT; the loop is repeated if INC is positive and the value of VAR is less then or equal to the value of LIMIT. If INC is negative, the loop is repeated only while the value of VAR is greater than or equal to the value of LIMIT.

Type (*iii*) assigns to VAR the value of the first item in LIST. Succeeding values are used for each successive time around the loop until LIST is exhausted.

The following are examples of the use of DO:

Type (*i*) DO $.Y, M=1, M\leq 10 \& A(M)>0$
Type (*ii*) DO $.X, K=1, 100,$ K+1
Type (*iii*) DO $.Z, N=(1, 3, 4, 7, 11, 13, 17)$

Control of optional output

Selected results of macro and text substitution facilities may be used as an optional output. This is accomplished by the use of the EDIT psuedo-operation which may be used in a declarative, global, or range mode.

The declarative mode does not cause any output to be generated, but is used to declare the destination (printer, punch, or file) of the output and the method of handling long lines. It is also used to control the exceptions to the global output mode. For example, the statement:

```
PRINT  EDIT  OFF('ALL'),
       ON('REMARKS', NOTE, DOC),
                          CONT(72, 'X', '---')
```

would indicate that edited output is to be printed, and that any line that exceeds 72 characters is to be split into two print records with an X placed at the end of the first 72 characters and the remainder appended to the ---. If EDIT ON, the global form, were to be used with the above declarative, then only lines that contain NOTE or DOC in the operation field as well as all remark statements will be outputted.

The range form of EDIT allows a sequence of lines to be outputted regardless of their syntax. Lines outputted in this mode are then ignored by the remainder of the assembly processes.

Two examples of this form are EDIT .NEXT which causes the next line to be outputted, and EDIT .LINE which causes all lines up to, but not including, the line with the sequence symbol .LINE in its label field. See the Appendix for examples of the use of the EDIT pseudo-operation.

Macros

The macro facilities incorporated in SWAP make it one of the most flexible assemblers available. The macro facilities presented here are by no means exhaustive but only representative of the more commonly used features.

The general form of a macro definition is:

MACRO
prototype statement
macro text lines
MEND

The prototype statement contains the name of the macro definition as well as the dummy parameter names which are used in the definition. The macro text lines, a series of statements which make up the definition of the macro, will be reproduced whenever the macro is called.

Any operation, pseudo-operation, or macro may be redefined as a macro. Also, there are no restrictions as to which operations are used within a macro definition; this means that it is legitimate for macro definitions to be nested.

Macro operators and subarguments

Macro operators are provided to allow the programmer to obtain pertinent information about macro arguments and some of their common parts. A macro operator is indicated by its name character followed by a period and the dummy parameter name of the operand. For example, if a parameter named ARG has the value (A, B, C), then the number operator, N.ARG, would be replaced by the number of subarguments of ARG; in this example, N.ARG is replaced by 3.

Any subparameter of a macro argument may be accessed by subscripting the parameter name with the number of the desired subargument. Additional levels of subarguments are obtained with the use of multiple indexes. As an example, let the parameter named ARG assume the value $P(Q, R(S, T))$, then:

ARG(0)	is replaced by P
ARG(1)	is replaced by Q
ARG(2)	is replaced by $R(S, T)$
ARG(2, 0)	is replaced by R
ARG(2, 1)	is replaced by S

The macro operators may be used on the results of each other as well as on subparameters; for example, N.ARG (2) would be replaced by 2.

The following is an example of a simple macro to define a list of symbols:

```
          MACRO
          DEFINE LIST
          DO .LP, K=1, N .LIST
LIST(K,1) SET LIST(K, 2)
.LP       NULL # MARK END OF DO LOOP
          MEND
```

If the macro were called by the following line:
DEFINE ((SYMB, 5), (MATRIX (2), 7), (CC, 11))
it would expand to:

```
SYMB       SET  5
MATRIX(2)  SET  7
CC         SET 11
```

Macro functions

To provide more flexibility with the use of macros, several system parameters and macro functions have been made available. Macro functions are built-in functions that are replaced by a string of characters. This string, called the result, is determined by the particular function and its arguments. The arguments of a macro function may consist of macro parameters, other macro function calls, literal character strings, or symbolic variables. An example would be the DEC macro function, which has one argument, either a valid arithmetic or logical expression. The result is the decimal number equal to the value of the expression; the call DEC (7+8) would be replaced by 15.

Some of the major macro functions are:

1. IS(*expression, string*) is replaced by *string* if the value of *expression* is nonzero; otherwise, the result is the null string.
2. IFNOT(*string*) is replaced by *string* if the *expression* in the previously evaluated IS had a value of zero; otherwise, the result is null.
3. STR(exp_1, exp_2, *string*) is replaced by exp_2 characters starting with the exp_1 character of *string*.
4. MTXT(*tsym*) is replaced by the character string which is the value of the TEXT symbol *tsym*.
5. MTYP(*symb*) is replaced by the character that represents the type of the variable symbol *symb*.
6. MSUB(*string*) is replaced by the result of doing macro argument substitution on *string* a second time.
7. SYSLST(*exp*) is replaced by the *exp*th argument of the macro call.
8. MDO(*DO parameters; string*) is a horizontal DO loop where *string* is the range of the loop. Each time around, the loop produces the value of *string*, which is normally dependent on the DO variable symbol.

Keyword arguments

When the macro is called, keyword arguments are indicated by the parameter name followed by an equal sign and the argument string. An example would be the following calls of a MOVE macro:

```
MOVE  FROM=NEWDATA, TO=OLDDATA
                 or
MOVE  TO=OLDDATA, FROM=NEWDATA
```

Both calls will yield the same expansions as the expansion of the MOVE macro using normal arguments:

```
MOVE  NEWDATA, OLDDATA
```

Default arguments

Default strings are used whenever an argument is omitted from a macro call. The default string is assigned on the macro prototype line by an equal sign and the desired default string after the dummy parameter name. Although the notation is the same, default arguments are completely independent of the use of keyword arguments.

Marco pseudo-operations

The ARGS pseudo-operation provides a method of declaring an auxiliary parameter list which supplements the parameter list declared on the prototype statement. These parameters may also be assigned default values.

The parameters defined on an ARGS line may be used anywhere a normal parameter may be used. The parameter values may be reset by the use of keyword arguments.

It is also possible for the programmer to reset his named macro argument values anywhere within a macro by using the MSET pseudo-operation. For example:

```
PARM MSET DEC(PARM)
```

would change the value of PARM to its decimal value.

The following is an example of the use of the ARGS pseudo-operation:

```
    MACRO
    FUN NUMBER
    ARGS WORD=(ONE, TWO, THREE)
#     NUMBER=WORD (NUMBER)
    MEND
```

When the macro is called by FUN 1+1, the following comment would be generated:

```
# 1+1=TWO
```

but the call FUN 1+1, WORD=(EIN, ZWEI, DREI) would generate:

```
# 1+1=ZWEI
```

Text manipulating facilities

Some of the more exotic features provided by SWAP are the character string pseudo-operations and the dollar macro call.

HUNT and SCAN pseudo-operations

The HUNT pseudo-operation allows the programmer to scan a string of characters for any break character in a second string. It will then define two TEXT symbols consisting of the portions of the string before and after the break character. For example, the statements:

```
BRKS  TEXT  '+-*/'
        .
        .
        .
      HUNT  .LOC, TOKEN, REMAIN,
                  'LSIZE *ENTS', BRKS
```

will result in the symbols TOKEN and REMAIN having the string values of 'LSIZE' and '*ENTS' respectively. If one of the characters in BRKS could not be found in the scanned string, then a JUMP to the statement labeled .LOC would occur.

The SCAN pseudo-operation provides the extensive pattern matching facilities of SNOBOL3[1] along with success or failure transfer of control. This pseudo-operation is written:

SCAN TSYM $P_1 \ldots P_n$ GOTO

where TSYM is a previously defined string valued variable. The SNOBOL3 notation is used to represent the pattern elements P_1 through P_n as well as the GOTO field. See the references for a more detailed presentation of these facilities.

Dollar functions

Dollar functions are very similar to macro functions in that the result of a dollar function call is a string of characters that replace the call. However, these functions may be used on input lines as well as in macros. The dollar functions provide the ability to call a one-line macro anywhere on a line by preceding the macro name with a dollar sign and following it with the argument list in parenthesis. For example, the macro:

```
                MACRO
                CHECK          A, B
IS(A<B, DEC(B-A) MORE)
                IFNOT (DEC(A-B) OVER)
                MEND
```

could be called by:

```
OP   X # $CHECK(X,7)
```

For $X=4$, the line would appear in the assembly listing as:

```
OP   X # 3 MORE
```

and when X has the value 9, the line would appear as:

OP X # 2 OVER

Special control

A special pseudo-operation has been provided to allow the programmer to ignore most of the SWAP syntax on input lines. The pseudo-operation is called UNIOP for universal operation, and it assigns the macro named in the variable field as the operation to be used for all succeeding lines. This means that regardless of what appears on a statement, that macro is called and may be used to decompose the line into meaningful SWAP statements. The following macro will make a simple test (i.e., the presence of an equal sign) to see if a line is a FORTRAN arithmetic statement and interpretively perform the assignment if it is; otherwise, it will call the macro named OTHER.

```
              MACRO
              ARITH
#             STRIP STATEMENT NUMBER
                       AND LOOK FOR EQUAL
                       SIGN
              HUNT .OTHER, SYMB, RMDR,
                       'STR(7, 64, SYSLIN)', '='
MTXT(SYMB)    SET STR(2, 62, MTXT(RMDR))
                       # PERFORM ASSIGNMENT
              JUMP  .OUT # TERMINATE
                       MACRO EXPANSION
.OTHER        OTHER 'SYSLIN' # NOT
                       ARITHMETIC STATEMENT
              MEND
```

The system macro parameter SYSLIN is replaced by the entire line of the macro call. The HUNT pseudo-operation will search for an equal sign and force a jump to the statement labeled .OTHER whenever the equal sign cannot be found. If UNIOP were initially set to the ARITH macro by the statement:

UNIOP   ARITH

then the line:

100 MTX(2, 3) = MTX(3, 2) + 1

would serve as a call to the ARITH macro which would then generate the following line:

MTX (2, 3)   SET   MTX (3, 2) + 1

Approximately 150 lines of SWAP macro definitions (see the Appendix) were used to build an interpreter of a FORTRAN like language. The following is a listing of a sample program and the printout that resulted from interpreting the program.

```
      DIMENSION KOUNT(10, 10)
C
700   FORMAT (3X, 10I4)
C
      DO     50     N=1, 10
      KOUNT(N, 1) =1
50    KOUNT(N, N) =1
C
      DO     100    N=3, 10
      DO     100    M=2, N-1
100   KOUNT(N, M) =KOUNT(N-1, M)
C                         +KOUNT(N-1, M-1)
      DO     200    N=1, 10
200   PRINT     700, (KOUNT(N, M), M=1, N)
C
      STOP
      END
```

```
 1
 1  1
 1  2   1
 1  3   3   1
 1  4   6   4    1
 1  5  10  10    5    1
 1  6  15  20   15    6   1
 1  7  21  35   35   21   7   1
 1  8  28  56   70   56  28   8  1
 1  9  36  84  126  126  84  36  9  1
```

CONCLUSION

The general design and implementation of the SWAP macro assembler has led to three things:

1. The job of writing a program in assembler language has been made easier. This is saving many man hours of programmer effort over the life of a project.
2. The development of intermediate level languages using macros has been made easier. This is aiding in the design of a true higher level language by clarifying the requirements of the new language.
3. The interpretive processing capabilities of the SWAP assembler have been used to solve a wide variety of problems. This is significantly reducing

the number of programs needed and, more importantly, reducing the programmer effort required to produce a given program.

ACKNOWLEDGMENTS

The author wishes to acknowledge the contribution of Messrs. R. E. Archer, A. J. Emrick, N. M. Haller, and E. Walton of Bell Telephone Laboratories, Incorporated, to the design and implementation of SWAP. The author would also like to thank Mr. D. E. Eastwood for his many suggestions and "philosophical arguments."

REFERENCES

1 D J FARBER R E GRISWOLD I P POLONSKY
SNOBOL, a string manipulation language
JACM Vol II No 1 pp 21-30 January 1964

2 D J FARBER R E GRISWOLD I P POLONSKY
The SNOBOL3 programming language
BSTJ Vol XLV No 6 pp 897-901 July 1966

3 M E BARTON N M HALLER G W RICKER
Service programs
BSTJ Vol 48 No 8 pp 2866-2880 October 1969

APPENDIX

A SWAP Program to Interpretively Process a FORTRAN Like Language.

```
SYSPRINT EDIT  OFF(EDIT,'ALL'),ON('MACROS')
FTYPES%  TEXT  'IX'             # FORMAT ITEM TYPES
BRKS%    TEXT  '/,H'"()'        #FORMAT BREAK CHARACTERS
FTYPTB%  TRPAT (X(7D),'Q'),('(','P'),(')','C'),(',','C'),('/',
               'S'),(X(7F),'Q'),(255) # TRANSLATE BREAKS TO
                                ALPHABETICS
SQZ%     TRPAT (' ',0),(255)  # DELETES ALL BLANKS
EQ%      TEXT  '='              #EQUAL SIGN IS BREAK CHAR
DIMENSION OPSET ARRAY
STOP     OPSET END1
CONTINUE OPSET NULL
#
       MACRO # ALL UNDEFINED OPS ARE ASSUMED TO BE EQUATIONS
         NONOP
 HUNT .OUT V% E% 'MTR('STR(7,99,SYSLIN)',SQZ%)' EQ%   ## SQUEEZ
 OUT BLANKS
IS ('MTYP(O.MTXT(V%))'='U', DFN MTXT(V%) MTXT(E%))IFNOT(MTXT(V%)
 SET1 STR(2,99,MTXT(E%)))
       MEND
#
       MACRO
         GOTO  LOC,VAL=1
       JUMP    LOC(VAL)    ## ALSO TAKES CARE OF COMPUTED GOTOS
       MEND
#
       MACRO
         IF    COND,EQ,GT
TMP% TEXT 'MTR('COND',SQZ%)'
       SCAN    TMP%  *(E%)*  *LT%*   ## GET EXPRESSION
       JUMP    MTXT(LT%)    E%<0
       JUMP    EQ  E%=0
       JUMP    GT  E%>0
       MEND
#
       MACRO
         PRINT FMT
       DO    .X  K%=2,N.SYSLST    ## CHECK FOR ITERATIVE LISTS
IS('STR(1,1,SYSLST(K%))'='(', ITEM%)IFNOT(ITM%:DEC(K%) TEXT)
 'SYSLST(K%)'
.X     NULL
FMT    OUT_      MDO(K%=2,N.SYSLST; MTXT(ITM%:DEC(K%)) )
       MEND
```

```
        MACRO
FMT       OUT_
K% SET 1;J% SET 0 ;JJ% SET 0
.LP     EDIT     .NEXT    ## GENERATE A LINE OF PRINTOUT
MSUB(MTXT(FMT:_:DEC(K%)))
 JUMP .LP,SET(K%,K%+1)≤FMT:_L   ## HAS FORMAT BEEN EXHAUSTED
        JUMP     .OUT,J%≥N.SYSLST|J%≤JJ%   ## WHEN PRINT LIST
 EXHAUSTED OR NOTHING BEING DONE
JJ%     SET    J%
.RLP    EDIT     .NEXT     ## BACK UP TO LAST LEFT PAREN
MSUB(STR(FMT:_K,500,MTXT(FMT:_:DEC(FMT:_R))))
        JUMP .RLP  SET(K%,FMT:_R+1)>FMT:_L&JJ%<J%<N.SYSLST
        JUMP     .LP,J%<N.SYSLST
        MEND
#
        MACRO
          ITEM% IT              ## PROCESS ITERATIVE PRINT LIST
        HUNT     .LST,VAR%,REM%,'S.Q.IT',EQ%
TMP     MSET     MTXT(VAR%)
VS      MSET     TMP(N.TMP)    ##ISOLATE LOOP INDEX
        MACRO
        FRMNDX VS=I.DEC(VS)
VLST%   TEXT    'R.TMP(1).TMP(N.TMP-1)'
        MEND     FRMNDX
        FRMNDX     ## REPLACE INDEX BY ITS VALUE
ITM%:DEC(K%)  TEXT 'MDO(VS:MTXT(REM%);MSUB(MTXT(VLST%)) )'
        JUMP    .OUT
.LST    NULL
ITM%:DEC(K%)  TEXT IT    ## IT WAS JUST AN EXPRESSION
        MEND
#
        MACRO
FMT       FORMAT LST
        EDIT     SAVE,OFF            ## STOP PRINTING LINES
        MEND     FORT_PROG    ## SUSPEND  PROGRAM DEFINITION
REM%    TEXT     'LST'
A% SET 0;%LINES SET 1;FMT:_R SET 1  ;FMT:_K SET 1
FMT     BRK_OUT                 ## BUILD FORMAT DEFINITION
FMT:_L SET    %LINES
FMT:_:DEC(%LINES)  TEXT  'MDO(K%=1,A%;MTXT(ITM%:DEC(K%)))'
FORT_PROG  EXTEND       ## RESUME SOURCE PROGRAM DEFINITION
        EDIT     RESTORE      ## RESUME PRINTING LISTING
        MEND
#
        MACRO
FMT       BRK_OUT
.LP     HUNT     .OUT,TRM%,REM%,'STR(2,99,MTXT(REM%))',BRKS%
FMT     BRK_:MTR(REM%,FTYPTB%,1)   ## GO ON TRANSLATED BREAK
        JUMP     .LP
        MEND
```

```
        MACRO
          BRK_C                      ## COMMA OR RIGHT PAREN
        HUNT     .OUT,DUP%,TYP%,'MTXT(TRM%)',FTYPES%
        FTYP_:MTR(TYP%,FTYPTB%,1)
        MEND
#
        MACRO
FMT       BRK_P                      ## LEFT PAREN
FMT:_R  SET     %LINES-1   ## SAVE POSITION FOR AUTO REPEAT
FMT:_K  SET     1:MDO(K%=1,A%;+K.MTXT(ITM%:DEC(K%)))
        SCAN   REM%  *(SAVE%)*   *SV2%*     /F(.OUT)
BLMT%   SET     MAX(1,TRM%)    ## DUPLICATION FACTOR
        DO       .BK,B%=1,BLMT%
REM%    TEXT     'MTXT(SAVE%)'
.BK     BRK_OUT
REM%    TEXT     ',MTXT(SV2%)'
        MEND
#
        MACRO
FMT       BRK_S                      ## SLASH
        BRK_C
FMT:_:DEC(%LINES)   TEXT 'MDO(K%=1,A%;MTXT(ITM%:DEC(K%)))'
A%   SET   0 ;%LINES   SET   %LINES+1
        MEND
#
        MACRO
          BRK_Q                      ## QUOTED STRING
ITM%:DEC(SET(A%,A%+1))  TEXT   'Q.MTXT(REM%)'
REM%    TEXT     'STR(K.Q.MTXT(REM%)+2,99,MTXT(REM%))'
        MEND
#
        MACRO
          BRK_H                      ## HOLERITH STRING
ITM%:DEC(SET(A%,A%+1))  TEXT 'STR(2,TRM%,MTXT(REM%))'
REM%    TEXT     'STR(TRM%+1,99,MTXT(REM%))'
        MEND
#
        MACRO
          FTYP_I                     ## INTEGER
LN      MSET     STR(2,10,MTXT(TYP%))
DP      MSET    DEC(MAX(1,DUP%))
ITM%:DEC(SET(A%,A%+1))  TEXT   ':I.MDO(%N=1,MIN(DP,I.N.I.SYSLST-
I.DEC(J%));I.DEC(I.SYSLST(SET(J%,J%+1)),LN,' '))'
        MEND
#
        MACRO
          FTYP_X                     ## BLANKS
ITM%:DEC(SET(A%,A%+1))  TEXT 'MDO(N%=1,MAX(1,DUP%); )'
        MEND
```

```
        MACRO
          END
SYSPRINT EDIT OFF              ## TERMINATE SOURCE LISTING
        MEND    FORT_PROG          ## END OF SOURCE PROGRAM
        FORT_PROG            ## NOW EXECUTE SOURCE PROGRAM
        END1                     ## TERMINATE RUN
        MEND
#
FORMAT   OPBITS ON(ACTIVE)     # ALLOW THESE OPS TO EXPAND
                                DURING MACRO DEFINITION
END      OPBITS ON(ACTIVE)
END      OPBITS OFF(CONT)      # NO CONTINUATION ALLOWED FOR END
                                MACRO
EDIT     OPBITS ON(ACTIVE)
         EDIT  ON(FORMAT,END)
#
        MACRO          # MAKE ENTIRE PROGRAM A MACRO DEFINITION
          FORT_PROG
SYSPRINT EDIT  .NEXT           ## EJECT TO NEW PAGE
1
PRINT    EDIT  ON              ## PRODUCE SOURCE LISTING
```