

MAKRO

An extraordinary disk-based macro cross-assembler
for the development of large programs on
Z80 or 8080 machines.

Copyright 1980
Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107
(213) 793-5748

MAKRO

Macro Assembler

MAKRO overlays the CCP portion of CP/M. MAKRO samples location 6 to determine the start of FBASE and then lays claim to the entire space available for transient programs.

The CP/M call to MAKRO is of the form

```
MAKRO FILE.XYZ
```

where FILE is the name of the input source file assumed to be type .ASM, and X, Y and Z are assembly options:

X source file unit
Y object file unit } = A,B,C, or D
Z is the assembly option.

MAKRO generates an object file type.HEX on the designated unit.

The assembly option Z controls generation of the OBJECT file and assembly listing. The three least significant bits independently control assembler options.

Bit 0 controls the extent of the assembly. If Bit 0 = 0, the assembler skips pass 2, and neither an object file nor pass 2 diagnostics are available. This option is used to make a quick check of the source file.

Bit 1 controls the assembly listing. If Bit 1 = 0, only assembly diagnostics are generated.

Bit 2 controls the generation of the object file. If Bit 2 = 0, no object file is created.

Assembly is normally performed with one of the three pass options:

- 1 No object file, pass 1 and 2 diagnostics only.
- 5 Object file, pass 1 and 2 diagnostics only.
- 7 Object file, full assembly listing.

LABELS within MAKRO can be up to 10 characters in length and may contain no special characters.

Pressing Control-C when entering file names to MAKRO returns control to the DOS.

A .PRN file may be generated by following the assembly option with a drive specification for the .PRN file:

```
MACRO SOURCE.XYZ B:
```

will generate a .PRN file on drive B.

INTRODUCTION

MAKRO is a powerful disk-based macro assembler for the development of large programs whose source files may exceed available memory. Both the source and object files of MAKRO reside on disk, freeing all available memory for macro storage and the construction of symbol tables. MAKRO is an extraordinarily powerful development tool incorporating many features not commonly available. The assembler is a working tool which has evolved under the demands generated by its use.

Program development with MAKRO is a two-step process: the source file is created, modified and saved on disk using the text editor EDIT; MAKRO reads the source file and creates the corresponding object file.

MAKRO INPUT/OUTPUT

MAKRO is a two-pass assembler, reading the source file first to construct a symbol table, then generating the object file on the second pass.

Source code for MAKRO consists of the four fields: Label, Operation, Operand, and Comments.

- (1) A line starting with a semi-colon is interpreted as a comment.
- (2) Entries in the label field must be terminated by a colon. The label identifier starts with the first non-blank character and ends with the colon. The colon requirement applies to SET and EQU operations, and macro definitions.
- (3) If a label is present, the operation field begins with the first non-blank character after the colon.
- (4) If no colon (hence no label) is detected, the operation field begins with the first non-blank character.
- (5) A comment field must be preceded by a semi-colon. Trailing comments preceded by a double semi-colon ;; are tabbed to the right of the operand field. Comments are not allowed on source lines containing a macro call.
- (6) Source lines must be terminated by carriage return/line feed.

The MAKRO user must identify the origin of the object code by an ORG operation at the start of his source code. Failure to do so will result in the code being assembled at location 0.

The list output of MAKRO displays the program counter, object code, and a well-formatted source display. Horizontal tab sets align the label, operation and operand fields for all source lines. An alphabetized symbol table is presented at the conclusion of pass 2 of the assembly.

MAKRO utilizes all available memory after the load address. Program constants and assembler symbol tables reside in memory immediately after MAKRO. Macro text is stored at highest available memory. The region between is used for macro processing operations.

ASSEMBLER OPERATION

Entries present in the label field are maintained in a symbol table. These entries are assigned a value equal to the program counter at the time of assembly, except that for the SET and EOU pseudo-operations, the variable defined by the label field is assigned the value of the operand field. Entries created in the symbol table by the macro definition refer to the storage location assigned to the text of the macro body. The variables defined by the label field can be used in the operand field of other instructions either as data constants or locations.

The operation field is separated from the label field by the colon. If no label field is present, the operation field may begin anywhere on the line. Entries in the operation field must consist of either a valid Z80 instruction, one of the several pseudo-operations, or a previously defined macro.

The operand field, separated by a blank from the operation field, consists of an arithmetic expression containing one or more program variables, constants, or the special characters \$, @ or %, connected by valid operators. Evaluation of the operand field is performed using 16-bit integer arithmetic. Operations requiring multiple operands (e.g., MOV A,B or BIT 3,IX,4) expect the operands to be separated by a comma. Parameters passed in a macro call are separated by commas and terminated by a carriage return.

The special operand \$ refers to the program counter at the start of the instruction being assembled. (NOTE: some assemblers interpret \$ as the start of the next instruction.) The program variable \$ can be used as any other program variable except that its value changes constantly throughout assembly. The location counter \$ allows the user to employ program-relative computations.

MAKRO recognizes two other special operands. The @, when used as an operand, refers to the repetition counter index. The %, as an operand, refers to the number of actual parameters in the current macro call.

Assembler constants may be decimal, hexadecimal, octal, or binary. Valid hexadecimal constants must begin with a decimal digit, possibly 0, and be terminated by the suffix 'H.' Binary constants are terminated by 'B' and may contain only the digits 0 and 1. Octal constants are terminated by 'O' and may contain only the digits 0 - 7.

Arithmetic expressions involving string operands must not begin with the string.
Example:

80H + 'A'	is valid
'A' + 80H	is invalid

PSEUDO OPERATIONS

ASSEMBLER

PSEUDO OPERATIONS

expr = arithmetic expression

ORG expr	Define program counter to nnnn.
DS expr	Reserve n bytes of storage. The first and last bytes of the reserved storage area are modified. An unmodified reserved area can be created by ORG \$+SIZE.
DW expr	16-bit datum definition.
DB expr	8-bit data or ASCII character string definition. The operand may be an ASCII character string enclosed in single quotation marks. Examples: <pre style="margin-left: 40px;">DB 5,0DH,'FILE'</pre> <pre style="margin-left: 40px;">DB 'ASCII STRING',0DH</pre>
EQU	The operand defined by the label field is set equal to the expression defined by the operand field. This operation is performed in pass 1 of the assembler and the variable definition is fixed by the last such definition encountered in pass 1.
SET	The operand defined by the label is set equal to the expression defined by the operand field. This operation is performed in both pass 1 and pass 2 and the replacement is effected upon every encounter.
* IF expr	expr is evaluated. If the result is zero the scanner skips to the next ENDIF, END, or end-of-file before resuming assembly. If the expression evaluates to any non-zero value, assembly proceeds. Operation is performed in both passes. Read IF as "SKIP IF ZERO."
* NIF expr	expr is evaluated. If the result is not zero the scanner skips to the next ENDIF, END, or end-of-file before resuming assembly. Equivalent to NOT IF. Read NIF as "SKIP IF NOT ZERO."
* ENDIF	Identifies the end of a conditional assembly block.
END expr	Terminates assembly. expr is an optional execution address to which the hex loader will branch after completion of the load: <i>lack of END statement will cause READLAF ERROR</i>
* <u>Neither the IF nor NIF blocks preceding the ENDIF may contain comments containing</u> * <u>the END or ENDI character sequences.</u>	

ASSEMBLERPSEUDO OPERATIONS

expr = arithmetic expression

USE operand

Allows program assembly to proceed with multiple location counters. The operation is skipped if the operand has not previously been defined; however, the definition can appear after the reference, to be used by pass 2. The USE operation is best explained by example:

```
AORG: SET 0A000H
BORG: SET 0B000H
      USE AORG;           SET code origin to AORG
      [ CODE AT 0A000H ]
      USE BORG;           SET value of AORG to PC
                          SET PC to BORG
      [ CODE AT 0B000H ]
      USE AORG;           Resume code at end of previous
                          block which started at A000.
      [ CODE ]
      USE BORG;           Resume code at END of block which
                          started at B000.
```

The USE instruction can be used to insert program data at the end of instruction code:

```
AFTR: SET LAST;         Not known on pass 1.
      ORG Start;       Somewhere.
      [ CODE ]
RESUM: SET $;           Remember where we are.
      USE AFTR
STRING: DB 'CHARACTERS'
      USE RESUM;       Resume in-line coding.
      [ CODE ]
      USE AFTR
      [ MORE DATA ]
      USE RESUM;       Continue
LAST: SET $
      END
```

MACRO**Signifies macro definition.**

ASSEMBLERPSEUDO OPERATIONS

expr = arithmetic expression

MACND

Signifies end of macro definition

GOTO label	Directs assembler to skip forward to label before resuming assembly. If label is reached via a GOTO branch, the symbol will not be entered into the symbol table. If label is reached via a normal assembly sequence it is treated as an ordinary statement label. GOTO is used in conjunction with conditional assembly to effect complex assembly sequences. GOTO allows forward references only. An invalid label terminates the assembly pass.
IFGZ expr;label	If expr evaluates to zero, the assembler branches forward to label; otherwise assembly continues.
IFGNZ expr;label	If expr evaluates to non-zero, the assembler branches forward to label; otherwise assembly continues. Labels reached by IFGZ and IFGNZ branches are not entered into the symbol table. Note that label must be separated by a semi-colon from the end of expr.
REPT expr	Repeat block. The value of expr determines the number of times the repeat block is executed.
REPND	Defines the end of a repeat block. The portion of source code bracketed by REPT/REPND is assembled repeatedly.
USR expr	Assembly-time branch to user routine. MAKRO branches to the address given by the value of expr. The user routine may utilize all registers. MAKRO may be re-entered by a return RET. Upon entry to the user routine, the zero flag is set for pass 1 of the assembly, and the DE registers contain the address, within MAKRO, at which assembly must resume. This pseudo-operation provides the means for controlling output.
IFEQ STR1,STR2;LABEL	Branch to LABEL if character string STR1 is identical to STR2.
IFNE STR1,STR2;LABEL	Branch to LABEL if character string STR1 is not identical to STR2.

ASSEMBLERPSEUDO OPERATIONS

expr = arithmetic expression

IFNEG expr;LABEL	Branch to LABEL if expr results in a negative value.
IFDEF SYMBL;DEFND	Branch to DEFND if SYMBL has been entered in the symbol table.
LIST	Turns on full assembly listing, restoring any pass options.
NOLST	Turns off full assembly listing, retaining diagnostic and error messages.
COMPS STR1,STR2;LABEL	Branch to LABEL if character string 2 is greater than character string 1.
LINK FILENAME	Merges disk file FILENAME into the current assembly. The LINK pseudo-operation enables the assembly to include previously developed program modules.
INPUT	MAKRO allows the user to define program variables at assembly time. The INPUT pseudo-operation accepts an expression from the console input, evaluates that expression, and assigns the computed value to the variable defined by the label field.
XPAND	Display macro expansion (default case).
NOEXP	Suppress macro expansion.
APUSH expr	Places the value of expr on the internal assembly stack.
LABEL:APOP	Similar to SET pseudo-op except that value of LABEL is recovered from assembly stack. APUSH and APOP are primarily used within nested control macros as in FOR/NEXT loops. Such nesting requires that the starting address of FOR loops be recovered in reverse sequence by the following NEXT macros.
FORM	Causes page eject (via form feed). <i>(TITLE should precede FORM)</i>
TITLE 'PAGE HEADING'	Causes corresponding heading to appear on subsequent pages of the assembly listing. If the TITLE field is empty, MAKRO will prompt the user during pass 2 for the page heading. The prompt option is exercised by terminating the TITLE pseudo-op with a carriage return.
SETQ expr	Sets internal label-generating assembly variable to value of expr. A question mark appearing in the label field is expanded as the character string representing the hex value defined by SETQ. This operation was implemented to allow communication between macros.

ASSEMBLER ERRORS/DIAGNOSTICS

Assembler error and diagnostic messages consist of single character identifiers which flag some irregularity discovered during either pass 1 or pass 2 of the assembly.

- P Phase error: the value of the label has changed between the two assembly passes.
- L Label error: missing operation field or invalid destination label.
- U Undefined program variable.
- V Value error: the evaluated operand is not consistent with the operation.
- S Syntax error.
- O Opcode error.
- M Missing label field.
- A Argument error.
- R Register error.
- D Duplicate label.

MAKRO CONDITIONAL ASSEMBLY

The conditional assembly features of MAKRO include

COMPS	String comparison
IFEQ	Character string equality
IFNE	Character string inequality
IFNEG	Branch on negative
IFDEF	Branch if defined symbol
IF	Skip if zero
NIF	Skip if not zero
ENDIF	Termination of conditional block
IFGZ	Branch to label if zero
IFGNZ	Branch to label if not zero
GOTO	Unconditional branch

These pseudo-operations enable the programmer to direct the assembly by performing assembly time computations. In the simplest application, conditional assembly allows a program to be written with a number of options, such as various input/output modes, with the desired array of options selected by program switches. A single source code module can thus be used for a variety of applications. More powerful application of conditional operations directs the assembly according to results generated during the assembly process. An example of this application is given in the discussion of macro processing.

The conditional assembly operations effect their branching upon the results of evaluating an arithmetic expression. The expression begins with the first non-blank character after the operation field and ends with a carriage return or semi-colon. The label directed branches IFGZ and IFGNZ include a destination field following the expression. A semi-colon must separate the destination from the expression. The destination field is terminated by a blank or carriage return. Branching is performed in a forward direction only, the assembler skipping over source code until the destination label or end-of-file is detected.

Treatment of the destination label in label-directed branches requires discussion. The general form is

```
Branch expr; There
else : here
[ CODE ]
There:
```

If the branch condition is not satisfied, assembly proceeds in sequence with else, in which case the destination label (There) may be reached in the course of assembly. In this, the fall-through case, the destination label is treated as an ordinary statement label and is entered into the symbol table. However, if the branch condition is satisfied, the label is reached via a skip, and normal assembly proceeds with the first character following the colon at the destination. The destination label is not seen by the assembler.


The IF/ENDIF and NIF/ENDIF assembly blocks bracket portions of code which are conditionally assembled or disregarded. The IF block is disregarded if the corresponding expression evaluates to zero. The NIF block is disregarded if the expression evaluates to not-zero. Mnemonically, these conditions refer to the skip rather than the assembly.

Nested IF/NIF blocks cannot generally be assembled correctly. Consider blocks nested as

```
a  IF          expr1
b  IF          expr2
c  ENDIF       hopefully for the inner
   [ CODE ]    some code in here
d  ENDIF       hopefully for the outer
```

Assembly proceeds as follows:

expr1 is evaluated, the assembler skipping to the first ENDIF (c) if expr1 is zero. If expr1 is not zero, expr2 is evaluated, the assembler reaching the ENDIF (c) regardless of the results. It is seen that CODE is assembled regardless of the contents of either expression. The second ENDIF (d) is superfluous, and is ignored. There may be applications of such behavior, but the operation seems more likely to be a source of confusion. Complicated conditional branching is more easily and clearly generated by the label-directed operations.

 A cautionary flag must be raised regarding conditional assembly. Phase changes of assembly variables (change in value between the two assembly passes) can result in a totally invalid assembly. If such phase changes cause the course of the assembler through the source code to differ for pass 1 and pass 2, the resulting assembly is almost certain to fail. You must remember that any and all branches performed in pass 1 must be repeated in pass 2.

The character string tests, IFNE and IFEQ, perform a character-by-character test of the first two parameter strings, conditionally effecting the branch upon the outcome of the comparison. The forms of these operations are:

```
IFEQ STR1,STR2;LABEL
IFNE STR1,STR2;LABEL
```

String 1 begins with the first non-blank character after the operation code and extends to the character preceding the comma. String 2 includes the character following the comma through that preceding the semi-colon.

Remember that the destination field must be preceded by a semi-colon and that the destination label vanishes if the branch is true.

IFNEG expr;LABEL

expr is evaluated. If the result is negative (15-bit signed arithmetic) the assembler branches to LABEL. IFNEG, IFGZ and IFGNZ can be combined to effect any computational branch.

IFDEF SYMBOL;LABEL

The symbol table is searched for symbol. If the entry is found, assembly skips to LABEL. IFDEF is used to provide automatic type declaration.

COMPS STR1,STR2;LABEL

A character-by-character comparison is made between STR1 and STR2. If STR2 is greater than STR1, assembly branches to LABEL. The COMPS pseudo-op is used to test parameter type in a macro call.

MAKRO MACRO CAPABILITY

INTRODUCTION TO MACROS

A macro can be considered an assembly language super-instruction with which the user can invoke many elementary assembly language statements with a single macro call. Users familiar with FORTRAN utilize a macro in the FORTRAN statement function. BASIC programs using the DEF FN operation capitalize upon an economical feature similar to a macro. The PL/1 pre-processing pass is a macro phase.

Assembly language programming is distinguished from such high level languages on the basis of the translation from the programmer-oriented language to the machine-oriented object code. This translation is performed on an approximately one-to-one basis for assembly language programs -- one machine instruction for each assembly language instruction. Programs written in a high level language enjoy greater leverage in that a high level language statement may result in the generation of many elementary machine code instructions.

A macro assembler can be regarded as bridging the gap between rudimentary assembly and high level language programming. Indeed, several high level languages have been implemented upon an underlying macro structure. A high level language implemented by macros can furnish the efficiency of assembly language and the ease of high level programming. Via macros, the user can design his own open-ended high level language.

MACRO PROCESSING

Interpretation of a macro involves the three steps:

- macro definition
- macro call
- macro expansion

The macro definition is the means by which the programmer informs the assembler of the instruction sequence to be effected. Briefly, in the macro definition the programmer informs the assembler that "when I say this, I mean that." The macro definition associates a name (label) with the sequence of instructions. Subsequent to the definition, the macro name is used as an entry in the op-code field to invoke the entire instruction sequence. In order to provide more power and flexibility to the macro, beyond that which can be furnished by a text editor, the macro definition allows certain parameters (dummy) to be included in the definition. These dummy parameters appear in the operand field of the macro definition. The assembler recognizes the dummy parameters when they

appear in the sequence of instructions comprising the body of the macro.

The macro definition thus consists of the following:

```
NAME: MACRO          dummy parameter list
      [ MACRO BODY ]
      MACND           signals end of definition
```

The macro call consists of the macro name appearing in the operation (opcode) field of a subsequent instruction. Actual parameters, appearing in the operand field of the macro call, replace the dummy parameters of the macro definition.

In the macro expansion phase, the instruction sequence representing the body of the macro is delivered to the assembler. Dummy parameters appearing in the macro body are replaced, in sequence, by the actual parameters included in the call. With the single macro call, the user has invoked an entire instruction sequence.

MAKRO deals with the macro definition during pass 1 of the assembly. Source text, comprising the macro body, is transferred to a temporary buffer following the symbol table. The source text is scanned for occurrences of the dummy parameters which are replaced by the parameter sequence number. The compressed macro text is then stored uppermost in memory.

Macro expansion must be performed for both passes of the assembly. After recognizing a macro call, the body of the macro is expanded into the buffer area, with actual parameters replacing the parameter sequence values. Assembler input is directed to the expanded text (away from the mass storage device). Input from the mass storage device is resumed when the body of the macro is exhausted.

MAKRO IDIOSYNCRACIES

The treatment of macros by MAKRO differs somewhat from conventional technique. The differences, however, stem from careful consideration, and MAKRO processing is considerably more powerful than alternative methods. The primary departure from convention arises in the treatment of macro parameters. MAKRO delays the binding of parameter values until object code is generated (all parameters are call by name, not value). Dummy parameters appearing in the macro definition are treated as character strings which are recognized in the macro body regardless of their context. Thus, in the definition

```
MAX1: MACRO      String 1, String 2
      [ BODY ]
      MACND
```

any occurrence of String 1 in the macro body is regarded as a reference to the first dummy parameter. For example

```
MAX1: MACRO      THIS, THAT
DB 'THIS'        ;THIS or THAT
DW THAT
LXI H, THIS
MACND
```

is treated as reference to the dummy parameters as

```
DB '1'           ;1 or 2
DW 2
LXI H, 1
```

in which the digits represent the parameter sequence.

Actual parameters, in the macro call, are likewise treated without regard to context in the expansion phase. Character strings representing actual parameters directly replace the dummy sequence values. Thus the call

```
MAX1 ALFA, BETA
```

generates

```
DB 'ALFA'        ALFA or BETA
DW BETA
LXI H, ALFA
```

The revised and expanded body is then delivered to the assembler for interpretation.

PROCEDURAL AND SYNTACTICAL RULES

1. Dummy parameters must be at least two characters in length. All characters, including blanks, in both actual and dummy parameter strings, are considered significant.
2. Dummy and actual parameter strings begin with the first non-blank character in the operand field. Parameter strings are separated by a comma.
3. All labels generated within the macro body assume global status. The special character = appearing in the macro body is regarded as a reference to a four-digit hex number which is unique for each macro expansion. Labels generated for which global status is undesirable should be suffixed with the = character.

Thus, within the macro expansion,

LABEL:	assumes global status
L=:	is local to the current expansion

4. As a consequence of pass 1 treatment of the definition, a macro cannot be globally redefined.
5. No macro definition may appear within the body of another macro expansion.
6. Macro expansions may be nested up to ten deep, i.e., up to ten macro calls can be simultaneously active. (Refer to REPEAT BLOCK discussion).
7. Scanning for a macro call precedes the search through the op-code table. Thus a macro can be used to redefine a machine operation. For example, to trace jump operations the JMP instruction may be replaced by a macro as

```
JMP: MACRO ADDRESS
      PUSH PSW
      MVI A, 'J'
      CALL CHOUT
      CALL CHIN
      POP PSW
      DB 0C3H
      DW ADDRESS
      MACND
```

which causes the program to display 'J' and await keyboard input before effecting any JMP.

8. The number of actual parameters ordinarily agrees with the number of dummy parameters. Excess actual parameters are ignored. Insufficient actual parameters default to the null parameter.
9. The parameter separation character (default ',') in macro calls can be redefined at the time of macro definition. If the formal parameter list begins with a comma (,) the character immediately following is taken to be the parameter separation character for subsequent calls of that macro. The first formal parameter begins with the character following the separation character. This option is provided to allow syntactically more attractive macro usage.

10. The macro definition must precede any reference.
11. A null actual parameter, represented by two consecutive commas in the parameter string of the macro call, results in a null replacement string in the macro expansion. The first actual parameter is considered null if the calling parameter string begins with a comma.
12. The MACND pseudo-instruction may not be preceded by a label field.
13. MAKRO actual parameters, or portions thereof, enclosed in square brackets [], are treated as literal blocks and expanded without regard to any delimiters contained therein. Each such expansion strips off a matching pair of square brackets. The brackets must be balanced.

USING MACROS

Macro calls are typically used to alleviate tiresome sequences of instructions, such as in table generation or monitor function references. Thus

```
CHOUT: MACRO
        CALL OUTCH
        MACND
```

or

```
STATUS: MACRO PORT,STBIT
S=:     IN PORT
        ANI STBIT
        JZ S=
        MACND
```

illustrate the least imaginative exploitation of macro power. Computer literature is filled with awesome examples of the heights which can be reached by sophisticated macro use. See P.J. Brown, MACRO PROCESSORS, in which it is revealed that SNOBOL 4 is implemented by macros.

The following illustration of a high level language (BASIC) is presented in order to suggest more penetrating application of the macro:

TYPE DECLARATION

```
WORD: MACRO LABEL,VALUE
LABEL: DW VALUE
MACND
```

```
STRING: MACRO LABEL,DATA
```

```
LABEL: DB 'DATA'
```

```
NLABEL: EQU $+1-LABEL
MACND
```

If you want string length

```
LOOPVR: MACRO LOOP
```

Loop index variable

```
LOOPNM: DS 2
```

Loop start

Rep counter

```
MACND
```

PROGRAM LOOPING

```
FOR: MACRO LOOP,REPS
LXI H,REPS
SHLD LOOPNM
LOOPST: SET $
MACND
```

```
NEXT: MACRO LOOP
LHLD LOOPNM
DCX H
SHLD LOOPNM
MOV A,H
ORA L
JNZ LOOPST
MACND
```

ARITHMETIC OPERATIONS

```
ADDITION: MACRO LEFTARG,RTARG,ANSWER
LXI B,LEFTARG
LXI D,RTARG
LXI H,ANSWER
CALL FPADD
MACND
```

Macro expansion in conjunction with conditional assembly offers an especially powerful assembly combination. To illustrate, refer to the previously defined ADDITION macro. Now assume that we wished to address the destination (ANSWER) either directly as shown, or indirectly (LHLD instead of LXI). Further, assume that we wish to avoid the generation of the instruction entirely if the destination location is unchanged from a previous operation. Reflect upon the following complex:

```
ADDITION: MACRO LARG,RARG,ANS,FLAG
LXI B,LARG
LXI D,RARG
NIF HCON-ANS                               Check for valid H
GOTO ADDND
ENDIF
IF 1-FLAG                                   Flag is 0 for indirect
GOTO INDIR
ENDIF
LXI H,ANS                                   Direct
GOTO ADDND
INDIR:LHLD ANS                              Indirect
GOTO ADDND                                  Gobble label
ADDND: CALL FPADD
HCON: SET ANS
MACND
```

This macro was designed to illustrate many of the novel features of MAKRO. Some economy of code could have been effected by use of IFGZ and IFGNZ pseudo-operations. Note that no labels are generated by a call to this macro since the destinations INDIR and ADDND are invariably reached by a GOTO branch. Quite clearly the macro could be expanded to treat the left and right arguments as well. Complex macro usage greatly reduces the chance of coding error, since without macro expansion the chance of correctly entering a number of such sequences is minimal. A set of such complex macros need only be developed once and then merged into the current file. MAKRO, in conjunction with your macro file, becomes your high level language.

REPETITION CONTROL

MAKRO allows assembly time repetition (looping). A block of assembly code may be replicated up to 255 times by enclosing the block in REPT/REPND brackets. The form of the repeat block is

```
REPT expr
[ CODE ]
REPND
```

in which expr is evaluated, truncated to an 8-bit value, and used as a loop repetition factor. Repeat blocks may be nested, and may occur within a macro expansion. MAKRO maintains a control stack of length 80 bytes. The maximum depth of nesting is determined by the stack limit.

An active repeat block consumes 10 bytes of the control stack, and an active macro expansion consumes 8 bytes. Repeat blocks and macro expansions may be nested in any way so long as the total stack depth does not exceed 80 bytes.

In order to provide some flexibility to the repeat block, MAKRO recognizes two special operands:

- @ is a repeat loop index, counting up from zero, marking progression of the repeat block.
- % is a count of the number of active parameters in the most recent macro expansion.

MAKRO also allows looping over the actual parameters in a macro expansion. Such looping is governed by three special characters appearing in the macro body:

- +N Control-N Parameter flag (Press Control and N simultaneously)
- +S Control-S Start of macro loop
- +Q Control-Q End of macro loop

The start and end of the macro loop must be bracketed by +S/+Q; the loop is then repeated over all the actual parameters occurring in the macro call. Within such a loop, the elements of the parameter sequence are referenced by two +N's in sequence.

To illustrate the macro loop, assume we have a series of ASCII strings we wish to print, and that the sequence and number of these strings to be printed must vary within our program. Define the macro print all:

```
PNALL: MACRO
+S                               Start loop over all actual parameters
LXI H,+N+N
CALL PRINT
+Q                               End the loop
MACND
```

Now we use this macro as

```
PNALL S1,S2,S3
PNALL S6,S1,S9,S2,S7
```

The loop control automatically handles the counting and parameter referencing.

ASSEMBLY TIME INPUT

The INPUT pseudo-operation allows the user to define program variables at assembly time. Critical program variables, such as the assembly origin or I/O port numbers, may be entered as input variables, with their value determined by console input during pass 1 of the assembly.

As an example, assume that we have developed a program requiring input from a serial port; however, neither the port number or status mask can be standardized. We may therefore write the source program with these variables defined by input:

```
IPOINT:INPUT  
IMASK:INPUT
```

and the status check portion of the program would be

```
READY:IN IPOINT  
ANI IMASK  
JZ READY
```

The INPUT pseudo-operation is performed in pass 1 of the assembly. MAKRO displays the source line and awaits console input. The user may enter any valid expression which is terminated by a carriage return.

COMMUNICATION BETWEEN MACROS

The operations APUSH/APOP and SETQ allow communication between related macros. The function of these operations is exemplified by a conceptual DOIF macro.

As the name implies, the DOIF macro is to generate execution time instructions to selectively execute the following block of code. For cosmetic considerations, this macro will utilize '.' as the parameter separation character.

```
DOIF ,.ARG1.RELATION.ARG2
```

The macro is invoked as:

```
DOIF X.GT.Y
```

The macro must translate into a logical test of RELATION between the operands ARG1 and ARG2, and JUMP ahead if RELATION is false. While a backward reference can be effected by the SET pseudo-op, forward references cannot. (Why?)

The forward reference is implemented within the DOIF macro as

```
APUSH D=H  
JUMP IF FALSE TO D=
```

in which the = is uniquely expanded.

A subsequent IFEND macro generates the required label as

```
QVAL:APOP  
SETQ QVAL  
D?:
```

Test your understanding of the above by defining an ELSE macro to be inserted optionally between the DOIF and IFEND macros.

CHAINING SOURCE FILES

MAKRO allows a series of source files to be chained together via the LINK pseudo-operation. The LINK operation is performed at assembly time, producing an executable object module. The MAKRO LINK operation is performed at the source code level. The LINK pseudo-operation extends the assembly to include the named source file(s).

Suppose a main program is being developed which will require library modules FPPACK (a floating point package) and FPOUT (an input/output package). The main program should then include

```
LINK FPPACK  
LINK FPOUT
```

Assembly proceeds through the main program and continues through the link modules in the order given. The LINK pseudo-operation may appear anywhere in the source code, and LINK modules may themselves contain the LINK operation.

The LINK command, without a file name, acts as the INPUT pseudo-operation. The source line is displayed, prompting the definition of the link file at assembly time. Macro library files may be terminated by such a LINK command to chain the assembly to the current source file. In this case the macro library file should be specified as the input file.

The LINK file name must be terminated with a carriage return.

MAKRO EXPRESSION EVALUATION

ARITHMETIC EXPRESSIONS

Arithmetic expressions appearing in the operand field of MAKRO instructions are evaluated according to standard arithmetic rules. The following table defines the available arithmetic operations and the operator precedence.

<u>Operation</u>	<u>Precedence Value</u>	<u>Definition</u>
(16	Begin parenthetical expression
*	12	Multiplication
/	12	Division
\	12	Modulo, integer remainder
+	11	Addition
-	11	Subtraction
&	8	Logical AND
- or + (5E hex)	7	Logical OR
!	7	Logical EXCLUSIVE OR (XOR)
>	6	Right shift, zero fill
<	6	Left shift, zero fill
" (quote)		NOT, logical complement
)	0	End parenthetical expression

Expressions containing these operators are evaluated from left to right, execution of any operation delayed until all preceding operations of precedence value greater than or equal to the pending operation are performed.

The logical complement refers to the operand or parenthetical expression immediately following.

In the expressions

$$A > B, A < B$$

the left operand (A) is shifted in the indicated direction by B bit positions, with zero bits shifted in.

The modulo operator \ returns the integer remainder after division. Thus $A \setminus B$ yields

$$A - [A/B] * B$$

where the integer part of the bracketed term is taken. The modulo operator has precedence equal to *, /. The expression

22\3 * 5 yields 5 as
(22\3) * 5.

In any expression, the user may insert parentheses to force the intended computational sequence. In the previous expression, execution of the modulo can be delayed by

22\ (3*5) = 7

STRING HANDLING PRIMITIVE

Arithmetic operands and the first argument of the IFEQ and IFNE pseudo-operations may be subject to string segmentation. String segmentation is invoked if the first character of the operand is a left angle bracket '<'. The two characters immediately following the opening bracket are taken as the start/finish segmentation markers. The string argument is taken as the remaining characters up to but not including the right angle bracket '>'. The string handling primitive replaces the entire construct with the characters, if any, contained between the start/finish segmentation characters. Thus

<59123456789> yields 678
<()ARRAY(JI)> yields JI
<B(BARRAY(IJ))> yields ARRAY

The string primitive is also functional when recognized in the label field and macro parameter fields. Use of the segmentation primitive can be illustrated by a conceptual LOAD macro to place the value of the argument on an operand stack. The macro must take appropriate action when the argument is an array reference:

```
LOAD:MACRO ARG
    IFEQ <()ARG> ,;SCALAR          test for null index
    LXI H,<()ARG>                  else array, get index
    LXI D,<B(BARG)>
    DAD D
    GOTO QUIT
SCALAR:LXI H,ARG
    GOTO QUIT
QUIT:                               stack operand
    MACND
```