A MIX/MIXAL SYSTEM ON THE 360

by Frank Brice & Irving N. Rabinowitz

DCS Technical Report #18

Department of Computer Science
Hill Center for Mathematical Sciences
Livingston College
Rutgers, The State University of New Jersey
New Brunswick, New Jersey 08903

June 1972

The MIX Computer System

I.

A. Introduction

The MIX computer system is a machine-independent software system, written in FORTRAN IV, giving users the ability to assemble and execute programs written for the MIX computer, as defined in D. E. Knuth in *The Art of Computer Programming*. The system consists of four essentially independent major subsystems: the control system, the interpreter, the assembler, and the loader. The system runs as a single job under the operating system of the host machine, but the system has its own control statements to allow batching of MIX jobs within a single host machine job.

B. MCS: The MIX Control System

The MIX control system acts as an operating supervisor in running MIX jobs. All system input passes through the control system, and thus jobs are separated by control statements. In addition, control parameters are passed into the system via these control cards.

There are three types of statements in MCS: the MIXJOB statement, the EXEC statement, and the END statement. These MCS statements are distinguished from other system input by a control character punched in column one of each MCS card; the control character used in the current implementation is the equal sign, "=".

(1) The MIXJOB statement marks the beginning of a job and must physically be the first card of every job. This statement takes the following form:

= MIXJOB user identification and/or comments

The equal sign must appear in column one, as previously noted, but the rest of the statement has essentially free format. Embedded blanks are ignored, and any characters following the keyword MIXJOB are ignored.

(2) The EXEC statement passes information into the system about how a job is to be run. The standard form is as follows:

= EXEC ALG, 64, N

Again, imbedded blanks are ignored. The EXEC statement has three parameters: the PROC, or procedure parameter, the BYTE, or bytesize specification, and the DECK option. The three values shown above are the default values and may be omitted entirely (the EXEC statement itself, however, is still required). The parameters are separated by commas and must appear in the order shown.

The PROC parameter can take on four values, instructing the system how to process a job:

A	assemble only
ALG	assemble, load, and execute
LG	load and execute
G	go (this involves the MIX GO-button)

The first value invokes the assembler only; the second invokes the assembler, loads the object code into the MIX memory, and begins execution at the location specified in the assembly program. The third value invokes the absolute loader, and may only be used to load MIX object decks. Execution begins at the location specified in the object deck. The fourth value passes control immediately to the

MIX interpreter, which reads one card from the MIX card reader (the system input stream) into locations 0-15 and begins execution at location 0. This value does not provide for automatic loading.

The BYTE parameter can have the values 64 or 100, corresponding to the two easily implementable MIX machine configurations. (Note: the IBM/360 implementation allows assembly with either value, but execution is restricted to a bytesize of 64 only.) These values represent the binary and decimal versions of MIX.

The DECK parameter may be specified as Y or N, representing "DECK" or "NO DECK", respectively. This option applies only to assembly runs, and may be used to have the MIX object code produced during assembly punched onto cards in a form suitable for loading through the MIX loader. A word of caution is in order for users of the system's batching capabilities: card separators are not produced to separate decks punched by different jobs. Users must separate object decks themselves by checking for the "transfer" card at the end of each deck (see Section IV for a description of object decks and transfer cards). Another complication arises if fatal assembly errors occur: a transfer card is not produced and a card-by-card check must be made to separate decks. As a general rule, the deck option should not be used except under unusual circumstances, since the added costs of re-assembly are not prohibitive. (Note: The current Rutgers implementation directs all card output to

the printer, so users desiring punched output must also supply the necessary JCL to modify the MIX cataloged procedure.)

An alternative form of the EXEC statement is as follows:

```
= EXEC   PROC = ALG, BYTE = 64, DECK = N
```

When values are assigned by name in this manner, they may be given in any order or omitted (for default values) as desired. Assignment by location and assignment by name may also be mixed, provided the resulting statement is meaningful. For example, the statement

```
= EXEC   DECK = Y, 100
```

defaults the PROC parameter, requests a deck by name, and sets bytesize to 100 by location. The statement

```
= EXEC ,,Y
```

illustrates assignment by location only, with defaulting of the PROC and BYTE parameters.

(3) The third MCS statement is the END statement:

```
= END
```

This statement marks the end of a job and should follow all program and data cards of a MIX job.

Printed output for multiple jobs from the MIX system may be separated by a "system" page appearing at the beginning of each job. This page has the following header:

```
MIX 1009  COMPUTER SYSTEM, MODEL 100
```

This is followed by the MIXJOB and EXEC and images printed four lines down, giving the user identification supplied on the MIXJOB card. In addition, any MCS errors are noted on this system page.

MCS error messages are of two types. The first type is non-fatal and has the following form:

= MIXJOB ERROR --- {PROC / BYTE / DECK} OPTION ASSIGNED BY DEFAULT

If for any reason a given parameter cannot be read from an EXEC card, the above message will apear for that parameter. The second type of error is fatal and prints the following message:

= MIXJOB ERROR --- EXEC CARD MISSING; JOB DELETED

If the EXEC card of a job is missing or unidentifiable, the job is passed over and the next job is initiated. The same action occurs if the MIXJOB card is missing or unidentifiable, except that no message is printed. The system, in effect, never sees the job and thus leaves no trace of its having passed through the system; no system page is printed.

In summary, the deck setup to run a job under MCS is as follows:

= MIXJOB user identification

= EXEC

:

assembly or object program

:

data

:

= END

No separation is necessary between program and data. For assembly runs, the assembly END card serves this purpose; for users using the loader, the transfer card separates program from data.

For the Rutgers implementation on the IBM 360/67, the MIX deck should be enclosed by the following JCL:

```
// jobname          JOB  user identification
//                  EXEC MIX
// MIX  SYSIN       DD *
      :
  MIX  jobs
      :
//
```

II. The MIX Interpreter

A. Description

The MIX interpreter is a machine-independent program, written largely in basic FORTRAN with small sections in FORTRAN IV, which simulates the actions of the MIX computer as defined in Section 1.3.1 of D.E. Knuth's *The Art of Computer Programming*. It may be run as either a binary or a decimal MIX, but binary operation requires at least a 32-bit host machine word size (IBM 360, IBM 370), and decimal operation requires at least a 36-bit host machine size (GE 635, PDP-10, etc). MIX instructions are decoded and directly executed on a one-at-a-time basis in the same manner as actual computers execute their machine instructions.

Since the basic MIX machine is thoroughly described by Knuth, the purpose of this section is to describe extensions and optional features. The present input/output equipment is rather limited and consists only of the card reader (unit 16), the printer (unit 18), and the paper tape reader (unit 19). Both the card reader and the paper tape reader take their input from the standard input medium of the host machine (the card reader on the Rutgers 360/67 implementation). The printer output is routed to the standard output medium of the host machine (the line printer on the Rutgers 360/67). The IOC instruction (C=35) is inoperative.

Two options which are fully implemented are the indirect addressing mechanism and the floating point attachment. Both are fully described by Knuth and will not be further discussed here. Indirect addressing is described in exercise 3 of Section 2.2.2. The floating point attachment is described in Section 4.2.1, Part C.

A partially implemented option is the MIX negative memory, described in exercise 18 of Section 1.4.4. In the current implementation, the negative memory is present and may be used, but none of Knuth's conventions and none of the interrupt features have been implemented. MIX is normally run in normal state, in which negative memory is inaccessible. Once execution begins, there is no mechanism to switch from normal state to control state or vice versa. The only way to enter control state is prior to execution through the assembler or the loader. If the transfer address on the END card or the TRANS card is negative, MIX is placed in control state and execution begins at the specified negative location. MIX then remains in control state until it is halted, and the entire 7999 memory locations are available to the user. The one difference from normal operation is that jumps may not be made to negative locations, either from other negative locations or from positive locations.

The binary shifts and even/odd jump instructions described in Section 4.5.2 of Knuth have been implemented. In addition to the even/odd jumps for the A- and X- registers, even/odd jump instructions are available for all six of the index registers (J1E, J1O, J2E, J2O, etc.).

The final optional feature is a non-Knuth feature which may be used as a debugging aid. The operation TRACE (C=5, F=9) causes the following information to be printed at the completion of execuation of each instruction:

Line 1:

location counter, instruction just executed, A-register, X-register, J-register, MIX clock, comparison indicator

Line 2:

effective address, contents of effective address, registers 1 through 6, overflow indicator

Individual bytes are always given, except in the case of the address field of the instruction just executed and the J-register. The TRACE may be terminated by the Instruction ENDTR (C=5, F=10).

B. Errors

Errors occurring during execution of a MIX program cause a message to be printed along with a TRACE printout as described above. There are two types of errors: MIX machine errors and MIX program, or user, errors. If the former type occurs, the MIX systems programmer should be contacted immediately with full documentation of the error. The latter type are described below.

All of the MIX user errors are non-fatal except for three. In addition to these three, however, a program may be terminated due to its error count. Non-fatal errors are counted as they occur, and this count includes non-fatal assembly errors. Once this count exceeds 10, the program is terminated after the completion of the instruction which caused the final error. Note that this allows execution of programs containing an unlimited number of assembly errors provided no execution errors occur.

The fatal errors are described below:

1. MIX PROGRAM ERROR CRE20101, INTERVAL TIMER OVERFLOW; JOB TERMINATED

   Explanation: The MIX clock exceeded the maximum allowable run time (see part C, "Limitations")

2. MIX PROGRAM ERROR CRE20102, ILLEGAL OPCODE; JOB TERMINATED

   Explanation: The opcode of an instruction was not a valid opcode; i.e., it exceeded 63.

3. MIX PROGRAM ERROR CRE20103, ILLEGAL OR UNSUPPORTED OPERATION; JOB TERMINATED

   Explanation: The field specification for an instruction with an opcode of 5 did not specify a valid operation.

The non-fatal errors are as follows:

1. MIX PROGRAM ERROR CRE20104, INDEX REGISTER OVERFLOW IN LOAD INSTRUCTION

   Explanation: The operand of an index load exceeded two bytes; the register was loaded modulo the register size.

2. MIX PROGRAM ERROR CRE20201, ADDRESS OUT OF RANGE

   Explanation: The memory location referenced was not in the range -3999 to 3999; the address was reduced into the proper range.

3. MIX PROGRAM ERROR CRE20202, NEGATIVE ADDRESS ENCOUNTERED IN NORMAL STATE

   Explanation: A reference was made to negative memory without having the machine in control state; the address was transformed into a positive address.

4. MIX PROGRAM ERROR CRE20301, ILLEGAL INDEX FIELD IN INDIRECT ADDRESS REFERENCE

   Explanation: The index byte of the faulty instruction exceeded 63; it was reduced modulo 64.

5. MIX PROGRAM ERROR CRE20302, EFFECTIVE ADDRESS OVERFLOW

   Explanation: The effective address would not fit into two bytes; it was reduced modulo the square of the bytesize.

6. MIX PROGRAM ERROR CRE20401, L EXCEEDS R IN (L:R) FIELD SPECIFICATION; L SET TO ZERO

   Explanation: Self-explanatory.

7. MIX PROGRAM ERROR CRE20402, R EXCEEDS 5 in (L:R) FIELD SPECIFICATION; R SET TO FIVE.

   Explanation: Self-explanatory.

8. MIX PROGRAM ERROR CRE20901, ILLEGAL FIELD SPECIFICATION IN SHIFT INSTRUCTION; ZERO ASSUMED

   Explanation: The field byte of a shift instruction exceeded 7 and was set to zero.

9. MIX PROGRAM ERROR CRE20902, EFFECTIVE ADDRESS IN SHIFT INSTRUCTION IS NEGATIVE; ZERO ASSUMED

   Explanation: A shift of a negative number of bytes was requested and no shift was performed.

10. MIX PROGRAM ERROR CRE21001, ILLEGAL FIELD SPECIFICATION IN MOVE INSTRUCTION

    Explanation: The field byte of a move instruction exceeded 63; it was reduced modulo 64.

11. MIX PROGRAM ERROR CRE21301, FIELD SPECIFICATION OF JUMP INSTRUCTION EXCEEDS 9; ZERO ASSUMED

    Explanation: An illegal jump was requested; a straight JMP was performed.

12. MIX PROGRAM ERROR CRE21302, EFFECTIVE ADDRESS IN JUMP INSTRUCTION IS NEGATIVE; ABSOLUTE VALUE ASSUMED

    Explanation: Self-explanatory.

13. MIX PROGRAM ERROR CRE21303, EFFECTIVE ADDRESS OUT OF RANGE IN JUMP INSTRUCTION

Explanation: The target address of a requested jump was not in the range 0 to 3999; the address was reduced modulo 4000.

14. MIX PROGRAM ERROR CRE21304, FIELD SPECIFICATION IN REGISTER JUMP EXCEEDS 7; ZERO ASSUMED

    Explanation: An illegal register jump was requested; a jump if negative was performed.

15. MIX PROGRAM ERROR CRE21401, ILLEGAL FIELD SPECIFICATION IN ADDRESS TRANSFER INSTRUCTION; ZERO ASSUMED

    Explanation: The field byte of an INC, DEC, END or ENT instruction exceeded 3; INC was assumed.

16. MIX PROGRAM ERROR CRE21402, INDEX REGISTER OVERFLOW IN INC OR DEC INSTRUCTION

    Explanation: The result of an index INC or DEC instruction would not fit into two bytes; the result was reduced modulo the register size.

C. Limitations

Implementation restrictions in the area of optional equipment were covered in Part A: restricted input/output facilities and a very limited version of negative memory. The only other restriction is imposed by the interval timer, which allows each program 100,000 u, or 100,000 MIX time units, for execution. Timing is based on the instruction execution times given by Knuth. When the MIX system clock reaches this value, execution is interrupted and control is returned to the MIX control system.

III. MIX 1009 Assembler and Macro Processor, Version 3.1 (Released 11-10-71)

A. The MIXAL Assembler

The MIXAL assembler is a two-pass assembler which translates assembly instructions of the MIXAL language into machine code for either a binary or a decimal MIX computer. The assembler will process the MIXAL language exactly as it is defined by D. E. Knuth in *The Art of Computer Programming*, and in addition it allows a number of extensions to the language.

The first extension is a relaxation of the restrictions on future references. Future references may be used in any statement except an EQU or ORIG. This results in changes to rules 4b, 11b, 11c, and 12 in Knuth's definition of MIXAL in Section 1.3.2 of the above reference. Specifically, rule 4b is changed to "a symbol"; rules 11b and 11c are changed to "The ADDRESS should be a W-value which contains no future references;...", rule 12 is changed to include "A literal may not be used as part of an expression."

Another extension is free or fixed input format. Fixed, or Knuth, format is exactly as defined by Knuth, and this is the default format. Free format is governed by the following rules:

a) A LOC symbol must start in column 1;

b) the OPCODE (which may be up to ten characters in length) follows the first one or more blanks after the LOC symbol, but must begin before or at column 12;

c) the ADDRESS field follows the first one or more blanks after the OPCODE, but must begin before or at column 23;

d) comments may follow the first blank after the ADDRESS field, or must begin after column 23 if no ADDRESS field is present.

The pseudo-op used for changing formats is FORM. It may be used with three operands as follows:

FORM K	changes to Knuth format
FORM F	changes to free format
FORM	restores the previous format

Note that a program punched in free format must be preceded by a "FORM F" card which is punched in Knuth format. An additional word of caution is given for the use of the ALF pseudo-op in F-format: if leading blanks are desired in a word to be assembled by an ALF instruction, single quotation marks must surround the entire character string to inform the assembler that the blanks are a part of the word to be assembled. This is discussed further in the section on extensions to the ALF instruction.

The third extension includes three pseudo-ops which allow the programmer a certain degree of control over the format of the output listing. The first is the LIST pseudo-op, which may be used as follows:

LIST ON	lists all source and object lines
LIST OFF	suppresses printed output
LIST	restores the previous list option

The default value is LIST ON.

The SPC or SPACE (in F-format only) pseudo-op is used to cause lines to be skipped in the output listing. The ADDRESS field

is a W-value which causes V(W) lines to be skipped, where V(W) is the numerical value of the W-value. V(W) = 0 causes no lines to be skipped. V(W) < 0 causes an error message to be printed and no lines to be skipped, and if the ADDRESS field is blank, V(W) = 1 is assumed. However, if the page boundary is crossed, the listing continues at the top of the next page.

The EJCT or EJECT (in F-format only) pseudo-op causes a printer eject to the top of the next page. If it occurs at a normal page boundary, the EJCT is ignored. All three of these printer pseudo-ops are printed in the output listing unless a LIST OFF is in effect. They are printed after the appropriate actions have been performed; e.g., LIST OFF is not printed, LIST ON is printed, and EJCT is printed at the top of the new page.

An extended comment facility is another option in the assembler. In accordance with Knuth's definition, as asterisk in column 1 causes an input record to be treated as a comment. However, the asterisk need not appear in column 1; it need only be the first non-blank character. There is another type of comment called a REM comment. This type must have the letters REM as the first three non-blank characters on a card, with no imbedded blanks. These three letters are then deleted from the source record before printing. Note that this means that a symbol name REM may not appear in a program in the LOC field of an instruction.

Extended ALF and CON pseudo-ops allow a programmer to produce multiple object words from a single assembly instruction. This is accomplished in the ALF instruction by enclosing the character string to be assembled in single quotation marks. The assembler produces the

required number of object words in consecutive locations, adding trailing blanks if necessary to fill out the last machine word. Note that the presence of a quotation mark as the first character in an ALF operand always invokes this extension, so that if the programmer wants a quotation mark assembled as the first character of a string, he must follow the opening quotation mark by two others, which will be combined by the assembler into one. For example, to assemble the string 'TODAY IS NANCY'S BIRTHDAY', the following code should be used:

```
ALF '''TODAY IS NANCY''S BIRTHDAY'''
```

The character string in the operand may be of arbitrary length, as long is it fits on a single card.

The extended CON pseudo-op allows more than one W-value to be assembled by separating the W-values with semicolons. In addition, the same W-value may be assembled a number of times by using a repetition factor. When this feature is used, both the repetition factor and the number to be assembled must be W-values, and each one must be enclosed in parentheses. A repetition factor must be a positive integer; a zero or negative value causes an error message to be printed and a repetition value of one is assumed. A trailing semicolon in the operand of a CON instruction causes a zero to be assembled as the last word generated by the instruction.

An extension of the ORIG instruction is available with the BLOK, or BLOCK (in F-format only), pseudo-op. The instruction

```
BLOK N
```

has the same effect as the instruction

```
ORIG    * + N
```

It sets aside N words of storage by increasing the location counter by N, where N is a W-value containing no future references.

There are two pseudo-ops for extending the set of opcodes which the assembler will recognize. The first is the OPEQ pseudo-op, in which the symbol appearing in the LOC field is made equivalent to an opcode appearing in the ADDRESS field. For example, if the assembler is in F-format (to allow 10-character opcodes), the instruction

```
LOADA     OPEQ    LDA
```

would allow the opcode LOADA to be used to load the A-register in any assembly instruction following the OPEQ.

The OPDF, or OPDEF (in F-format only), instruction adds the symbol in the LOC field to the assembler's opcode table. The ADDRESS field of the OPDF instruction must be a W-value, the value of which defines the default values of the address, index, field, and opcode for the new instruction. These default values may be overridden by supplying new values when the new instruction is actually used. The OPDF instruction may be used to modify the default values of existing opcodes, but once changed, they can only be restored by the use of another OPDF instruction with the appropriate W-value. The machine opcode defined by the W-value in an OPDF instruction must be a valid MIX machine opcode; i.e., it must be an integer between 0 and 63, inclusive. For example, the instructions

```
BYTESIZE EQU       1 (4:4)
STOP     OPDF      2*BYTESIZE + 5
         STOP
```

define and use an opcode STOP which has the same effect as a HLT

instruction. The instructions

```
                FORM    F
STZADD1000      OPDF    1000 (0:2), 2(4:4), 33(5:5)
                STZADD1000
```

define and use an opcode STZADD1000 which stores zero into the address field of machine location 1000. This opcode could be used for location 2000 by using the instruction as follows:

```
STZADD1000      2000
```

An alternative method using indexing would be:

```
LDI             1000
STZADD1000      ,1
```

Other pseudo-ops which the assembler recognizes are MACRO (in F-format only), MEND, SETA, SETC, ANOP, AIF, BAIF, and FAIF, but these are used primarily in connection with the macro processor and will be discussed in detail in Part B of this section.

The pseudo-ops TRACE and ENDTR (both require F-format) are special instructions that generate machine words which cause starting and stopping (respectively) a trace of the execution of a program. This is a non-Knuth extension of the MIX machine itself, and is further covered in Section IV.

The various extensions to the MIX computer which Knuth defines in various parts of his series of books are recognized by the assembler. Specifically, the instructions AND, XOR, OR, and INT, as well as all of the floating point instructions FADD, FSUB, FMUL, FDIV, FLOT, and FCMP are included in the assembler's instruction set. In addition, the binary shift instructions and the jump-even and jump-odd instructions are recognized. Their definition and use is covered in Section IV.

The assembler checks all machine load addresses before transmitting a machine instruction to the loader or to the card punch. However, it does recognize the existence of negative memory in the MIX machine as defined by Knuth, and words are loaded into negative MIX memory. In other words, any instruction with a load address greater than -4000 and less than +4000 will be loaded without causing an error condition. In addition, if the transfer address on the END card is negative, MIX will be switched to control state and execution will begin at that negative address.

Output from the assembler consists of a printed listing of all source and object records in a side-by-side format (unless a LIST OFF is in effect) and an optional object deck on standard 80-column punched cards. All source lines are numbered on the listing, including any source lines generated by the macro processor. Pages are numbered and the bytesize for each assembly is printed on page 1. Diagnostic messages are printed at the end of the assembly listing, and they include warning and error messages. Terminal errors are printed when encountered and assembly is immediately terminated. Errors are discussed in detail in Part C of this section.

A request for optional card output produces a complete load module assembled for loading in absolute machine locations. The format is that definedd by Knuth in Section 1.3.1, problem 26. Fatal errors during assembly cause an object deck to be marked as non-executable.

B. The Macro Processor

This section is devoted to the definition of the macro language recognized by the macro section of the assembler. The macro language

is defined solely by the specifications of this section and is independent of any of Knuth's specifications.

A macro definition consists of four parts:

HEADER

PROTOTYPE

BODY

TRAILER

The header consists of a single card containing only the pseudo-op MACRO (F-format is required for using macros). The prototype has the form

```
$LOC      NAME      $ARG1,$ARG2,...,$ARGN
```

The NAME is the name of the macro and is governed by the usual rules for MIXAL symbols. It appears in the OPCODE field of the card. The optional $LOC symbol is an argument of the macro, and it may be regarded as $ARG0 if used. The remaining arguments appear in the ADDRESS field of the card and must be separated by commas. All argument names must be of the form $SYMBOL, where SYMBOL is a MIXAL symbol of up to nine characters. The macro trailer is the last card of the macro definition and consists of the macro name in the LOC field and the pseudo-op MEND. The macro name is required in the trailer. All macros must be defined before they are used.

The assembler allows the use of set symbols in connection with macros. These symbols must be defined prior to use. There are two types of set symbols: A-symbols and C-symbols. Both types have names of the form $SYMBOL, exactly the same as argument names. A-symbols are numerical symbols and have W-values as their values. C-symbols are character symbols and have character strings of up to ten charac-

ters as their values. Both types may be defined and re-defined any number of times during a program by using the pseudo-ops SETA and SETC, respectively. The character string in the ADDRESS field of a SETC statement must be enclosed in single quotation marks and is governed by the same rules as strings in extended ALF instructions. A-symbols may appear anywhere in an assembly program while C-symbols are restricted to macros or macro-related instructions (such as AIF).

A macro is called by using the macro name in the OPCODE field of a source record. The actual arguments should appear in the same relative positions as the corresponding formal arguments in the macro prototype. The actual arguments are the literal strings to be substituted for the formal arguments in the macro. Besides argument substitution all A-symbols and C-symbols appearing in the macro definition will have their current values substituted prior to any evaluation in the assembler.

Concatenation of strings and macro symbols may be done in a macro definition by merely concatenating the strings and symbols in the source record. In case of ambiguity, a period is used to separate the concatenated parts. For example, to concatenate the string STRING with the argument or set symbol $SYMBOL, one would write STRING$SYMBOL for concatenation on the right, and $SYMBOL.STRING for concatenation on the left. If $SYMBOL is an A-symbol, its W-value is converted to a string of digits representing a decimal integer, with leading zeroes suppressed, for substitution at the time of a call to the macro.

A macro call counter exists in the form of a system symbol called $SYSNDX. At the start of an assembly, its value is zero, and

each time a macro is called it is increased by one. It may be used to create unique names within nested or recursive macro calls. It is considered to be an A-symbol, and thus its value is a W-value.

Subarguments may be used in macros by the notation $SYM 1 < SYM 2 >, where $SYM1 is the name of a formal argument and SYM 2 is either an A-symbol or a W-value. This expression denotes the SYM 2^{th} subargument of the list which was presented as argument $SYM 1. When a macro call is made, a single argument consisting of a number of subarguments is enclosed in parentheses with the subarguments separated by commas. For example, if the argument $A is given as (X,Y,Z) in a particular macro call, and if $N is defined in a SETA statement to have the value 3, then the following substitutions could take place:

X,Y,Z,	for	$A
Y	for	$A < 2 >
Z	for	$A < $N >

In general, parentheses may be used to group parts of an actual argument which includes commas. The entire argument or any one of its parts may then be referenced, with the outermost level of parentheses being stripped off before substitution.

Conditional macro generation may be done through the use of the BAIF (backward assembly if), FAIF (forward assembly if), or AIF pseudo-ops, all of which are functionally equivalent. An AIF instruction is of the form

```
@LOC    AIF    (VALUE 1 = VALUE 2)  @ POINT
```

The LOC field must either contain a program point (to be discussed later) or be blank. The ADDRESS field specifies an equality condition

to be met for ajump to occur to the program point @ POINT. The parentheses around the condition statement are required. A VALUE may be:

a) the value of a set symbol, an argument, or a subargument;

b) a set symbol or a standard symbol;

c) a W-value;

d) a quoted string.

The condition is true and a jump is made if VALUE 1 is the same as VALUE 2, which implies that both are strings or both are W-values. Mixed modes cause an error condition, with the AIF condition considered false. An AIF instruction whose condition is false has no effect on the program being assembled.

A word of caution is in order for users of the AIF statement. All AIF's are processed in the assembler, after substitution by the macro processor. Any symbols appearing in the condition part of the AIF statement at this point must be known to the assembler if the statement is to work correctly, since the assembler performs a second level of substitution during evaluation of the value parts. For example consider the following:

```
ABC       EQU       1
$A        SETA      ABC
$B        SETC      'ABC'
$C        SETA      1
$D        SETC      '1'
          MACRO
          TEST
@N        AIF       ($A = $B)@N
@M        AIF       ($C = $D) @N
TEST      MEND
```

A call to this macro would result in the following code being sent to the assembler:

```
@N      AIF     (1 = ABC) @M
@M      AIF     (1 = 1) @N
```

Both conditions are true, with the symbol ABC being a standard MIXAL symbol defined by the EQU instruction. If one wanted to compare $B with the string 'ABC' in an AIF instruction, the following code would be used.

```
$A      SETC    '$B'
$B      SETC    'ABC'
        MACRO
        TEST
@M      AIF     ($A = 'ABC') @M
TEST    MEND
```

This would send the following to the assembler:

```
@M      AIF     ($B = 'ABC')@M
```

The assembler would then evaluate $B and compare its value, ABC, with the literal string ABC. This is the only way a C-symbol may be used in the assembler, and even when used this way, the AIF statement using the C-symbol must result from expansion of a macro.

The program points mentioned above take the form @SYMBOL, where SYMBOL is a MIXAL symbol of up to nine characters. A program point may appear in the location field of an instruction to identify it for a jump by an AIF instruction. Program points whould only be used within macros, since they are not valid location symbols and are not entered in the assembler's symbol table. A line may be marked with a program point even if it is already labeled with a standard

location symbol by preceding it with the instruction ANOP labelled with the program point in its location field. ANOP is an assembler "no-op" which has no effect on the assembly process. Its only purpose is to label a line already labelled, and the only type of label permitted is a program point.

The final feature of the macro processor is related to arguments containing subarguments. If $SYMBOL is an argument containing n subarguments, the symbol $SYMBOL < 0 > takes the value n, which is a W-value.

The MIXAL macro processor is an optional feature available to MIXAL users; however, it is offered strictly on an "as is" basis. Comments about problems encountered with the macro processor are invited, but support for this part of the MIX system cannot be guaranteed. No warranty of any kind is given that the MIXAL macro processor will meet any of the above specifications.

C. Errors

Error handling in the assembler and macro processor is done on three levels. The first type of error is non-fatal and is listed as a "warning" in the error messages of an assembly listing. Any number of these are permitted without effect on the assembly process, since some type of corrective action can be taken in every case. However, a total count of these errors is sent to the MIX interpreter, which uses this count in its own error handling system (see Section II, Part B).

The second type of error is the user fatal error, for which no reasonable corrective action can be taken. Assembly continues but the code generated for the line containing the error is unlikely to

execute properly, and the load module generated is marked as non-executable.

The third type of error is the system fatal error. This occurs when the system discovers that it has made some type of error, or when any of the system tables become filled. When this type of error is encountered, the assembly process is terminated upon completion of assembly of the current statement. A termination message is issued on the assembly listing and control is returned to the MIX operating supervisor. If a user encounters this type of error, and if it is a table overflow, he should try to reduce his use of system resources or try to assemble in two or more parts. If the error is a system error, or if the user has a special need for expanded system tables, he should contact the MIX system programmer, who will take the appropriate action.

Error checking in the assembler is fairly complete, but not everything is checked. In areas where the syntax is not checked, however, the error is usually detected at some other point in the assembly. Deficiencies found by users are welcomed. Error checking in the macro processor is rather sparse. If the macro processor encounters something it does not recognize, it will usually ignore it and continue, without printing a diagnostic message. Problems encountered by users will be considered, but corrections cannot be guaranteed (see Part B).

A listing of MIXAL error messages follows, with a brief explanatory comment for each one. Warning messages for non-fatal errors are as follows:

1. WARNING: ILLEGAL USE OF 'AIF' PSEUDO-OP IN LINE XXX --- STATEMENT IGNORED

   Explanation: A BAIF, PAIF, or AIF was used outside of a macro.

2. WARNING: ILLEGAL OPERAND IN 'FORM' PSEUDO-OP IN LINE XXX --- FREE FORMAT ASSUMED

   Explanation: The operand of a FORM instruction was not F, K, or blank.

3. WARNING: FORM STACK FULL AT LINE XXX --- FIRST HALF OF STACK DELETED

   Explanation: The stack used to store previous FORM commands was filled, so the oldest half of the stack was discarded. Form restoration may only be used for the most recent half of the FORM commands; for the actual size of the stack, see Part D.

4. WARNING: ILLEGAL OPERAND IN 'LIST' PSEUDO-OP IN LINE XXX --- 'ON' ASSUMED

   Explanation: The operand of a LIST instruction was not ON, OFF, or blank.

5. WARNING: LIST STACK FULL AT LINE XXX --- FIRST HALF OF STACK DELETED

   Explanation: See number 3.

6. WARNING: NEGATIVE OPERAND IN 'SPACE' PSEUDO-OP IN LINE XXX --- ZERO ASSUMED

   Explanation: The user asked to skip a negative number of line in a SPC instruction; the statement is ignored.

7. WARNING: SYMBOL EXCEEDS PERMISSIBLE LENGTH IN LINE XXX --- SYMBOL TRUNCATED

   Explanation: A symbol more than ten characters long was encountered; only the first ten were kept.

8. WARNING: ILLEGAL MACHINE OPCODE SPECIFIED IN 'OPFD' PSEUDO-OP IN LINE XXX --- NOP INSERTED

   Explanation: The opcode byte of the W-value is an OPDF instruction exceeded 63; the value of the byte was set to zero.

9. WARNING: OPCODE MISSING IN LINE XXX --- NOP ASSUMED

   Explanation: The opcode was either missing or unidentifiable due to incorrect placement in the source statement.

10. WARNING: OPCODE EXCEEDS PERMISSIBLE LENGTH IN LINE XXX --- OPCODE TRUNCATED

    Explanation: This message applies to free format only; the opcode exceeded ten characters in length, only the first ten were kept.

11. WARNING: NONBLANK CHARACTER FOUND IN COLUMN 11 IN LINE XXX

    Explanation: This message applies to Knuth format only; the rule specifying that column 11 must be blank was violated; the character is ignored.

12. WARNING: NONBLANK CHARACTER FOUND IN COLUMN 16 IN LINE XXX

    Explanation: See number 11.

13. WARNING: TERMINAL '=' MISSING IN LITERAL CONSTANT IN LINE XXX

    Explanation: A blank or end-of-record was encountered before the literal terminated; the blank terminated the entire address field and the missing equal sign was effectively inserted at that point.

14. WARNING: ILLEGAL CHARACTER FOLLOWS LITERAL IN LINE XXX --- ALL SUBSEQUENT CHARACTERS IGNORED

    Explanation: A literal may only be used as the A-part of an ADDRESS field, and thus must be followed by a comma, a left parenthesis, or a blank; this rule was violated and assembly of the instruction terminated at the illegal character.

15. WARNING: LITERAL CONSTANT EXCEEDS PERMISSIBLE LENGTH IN LINE XXX --- FIELD TRUNCATED

    Explanation: A literal exceeds nine characters (exclusive of equal signs); the entire literal is evaluated, but only the first nine characters are kept as the symbol identifier.

16. WARNING: LITERAL CONTAINING NO CHARACTERS ENCOUNTERED IN LINE XXX --- ZERO GENERATED

    Explanation: The string "= =" was encountered in the A-part of an ADDRESS field and was treated as though it were "=0=".

17. WARNING: ADDRESS FIELD OVERFLOW IN LINE XXX --- HIGH ORDER BYTES IGNORED

    Explanation: The computed address would not fit into two MIX bytes; the address was assembled modulo the square of the bytesize.

18. WARNING: UNEXPECTED DELIMITER IN COLUMN YY OF LINE XXX --- REMAINDER OF FIELD IGNORED

    Explanation: A syntax error was encountered in the source record; assembly of the statement was terminated at the point of error.

19. WARNING: INDEX FIELD OVERFLOW IN LINE XXX --- HIGH ORDER BYTES IGNORED

    Explanation: The computed index byte would not fit into one byte; the value was assembled modulo the bytesize.

20. WARNING: NEGATIVE INDEX VALUE FOUND IN LINE XXX --- SIGN HAS BEEN DROPPED

    Explanation: The computed index value was negative; its absolute value was assembled.

21. WARNING: RIGHT PARENTHESIS MISSING IN FIELD SPECIFICATION IN LINE XXX --- FIELD TERMINATED BY FIRST BLANK

Explanation: A blank or end-of-record was encountered before the F-part of an ADDRESS field terminated; the blank terminated the F-part and the missing parenthesis was effectively inserted at that point.

22. WARNING: FIELD SPECIFICATION OVERFLOW IN LINE XXX --- HIGH ORDER BYTES IGNORED

Explanation: See number 19.

23. WARNING: NEGATIVE FIELD VALUE FOUND IN LINE XXX --- SIGN HAS BEEN DROPPED

Explanation: See number 20.

24. WARNING: RIGHT PARENTHESIS MISSING IN FIELD SPECIFICATION IN LINE XXX --- FIELD TERMINATED BY ' ', '=', or ','

Explanation: A blank, equal sign, or comma was encountered in the field part of a W-value; the delimiter terminated the field part, and a right parenthesis was effectively inserted at this point.

25. WARNING: ILLEGAL DELIMITER FOLLOWS W-VALUE IN LINE XXX --- REMAINDER OF FIELD IGNORED

Explanation: Each part of a W-value must be terminated by a comma or a blank; the character encountered was not one of these, and assembly of the statement was terminated at this point.

26. WARNING: ILLEGAL FIELD SPECIFICATION COMPUTED IN W-VALUE IN LINE XXX --- (0:5) INSERTED

Explanation: The field specification of a W-value must be of the form (L:R), with $0 \leq L \leq R \leq 5$; this rule was violated and the default value (0:5) was assembled.

27. WARNING: ILLEGAL CHARACTER IN COLUMN YY OF 'SETC' STATEMENT IN LINE XXX --- NULL STRING ASSUMED

    Explanation: A set symbol or a quoted literal string was expected but not found; a null string was assumed.

28. WARNING: CHARACTER STRING EXCEEDS PERMISSIBLE LENGTH AT COLUMN YY IN LINE XXX --- STRING TRUNCATED

    Explanation: A literal character string exceeded ten characters; only the first ten were kept.

29. WARNING: UNDEFINED OR ILLEGAL SYMBOL IN OPERAND FIELD OF 'SETC' STATEMENT IN LINE XXX --- NULL STRING ASSUMED

    Explanation: The set symbol in the operand of a SETC statement was either undefined or of the wrong mode; i.e., an A-symbol instead of a C-symbol.

30. WARNING: CLOSING QUOTE MISSING IN CHARACTER STRING IN LINE XXX

    Explanation: A literal character string ran off the end of the source record; a closing quote was effectively inserted at the end of the record.

31. WARNING: ILLEGAL USE OF LOCAL SYMBOL IN LINE XXX --- SYMBOL IGNORED

    Explanation: A local symbol nH appeared in an ADDRESS field, or a local symbol nB or nF appeared in a LOC field.

32. WARNING: DUPLICATE SYMBOL IN LINE XXX --- SYMBOL IGNORED

    Explanation: A non-set symbol was defined more than once; the original definition was used.

33. WARNING: UNDEFINED SYMBOL IN LINE XXX --- SYMBOL DEFINED BY DEFAULT

Explanation: This condition does not actually violate the rules for MIXAL, but is called to the programmer's attention in case he is not aware of it; in accordance with the rules as defined by Knuth, undefined symbols are initialized to zero.

34. WARNING: ILLEGAL C-SYMBOL IN LINE XXX --- SYMBOL IGNORED

Explanation: An attempt has been made to define a C-symbol without including a '$' as the first character of the identifier.

35. WARNING: ILLEGAL OPCODE IN LINE XXX --- NOP INSERTED

Explanation: A non-standard opcode has been used without having been previously defined by an OPEQ or OPDF; a no-op was assembled.

36. WARNING: EXPRESSION TERMINATES WITH OPERATION IN LINE XXX --- OPERATOR IGNORED

Explanation: The left-to-right expression scanner encountered an end-of-expression before finding the final operand for the last operator.

37. WARNING: OVERFLOW IN LINE XXX

Explanation: The result of an arithmetic operation exceeded the MIX word size; the result was taken as $\pm(b^5-1)$, depending on the sign of the result, where b = the MIX bytesize.

38. WARNING: DIVISION BY ZERO IN LINE XXX

Explanation: Overflow occurred due to an attempted division by zero; the result was taken as $\pm(b^5-1)$.

39. WARNING: OVERFLOW DURING DATA CONVERSION AT LINE XXX

Explanation: Overflow occurred at some point during a conversion from a character string of digits (either a C-symbol value or a constant in the assembler's input stream) to a W-value; the W-value used was $\pm(b^5-1)$.

40. WARNING: ILLEGAL CONDITION SPECIFICATION FOR 'BAIF' OR 'FAIF' PSEUDO-OP IN LINE XXX --- STATEMENT IGNORED

    Explanation: A syntax error occurred in the condition specification of an AIF statement.

41. WARNING: UNDEFINED $-SYMBOL IN LINE XXX --- STATEMENT IGNORED

    Explanation: An undefined set symbol was encountered in an AIF statement condition specification.

42. WARNING: MODE CONFLICT IN LINE XXX --- STATEMENT IGNORED

    Explanation: Incompatible data types were compared in the condition specification of an AIF statement.

43. WARNING: ILLEGAL OR MISSING PROGRAM POINT IN LINE XXX --- STATEMENT IGNORED

    Explanation: The program point in an AIF statement was either missing or did not begin with an '@'.

44. WARNING: PROGRAM POINT EXCEEDS PERMISSIBLE LENGTH IN LINE XXX --- SYMBOL TRUNCATED

    Explanation: A program point symbol exceeded the ten character limit for MIX symbols; only the first ten were used.

45. WARNING: CLOSING QUOTE MISSING IN 'ALF' CHARACTER STRING IN LINE XXX

    Explanation: See number 30.

46. WARNING: RIGHT PARENTHESIS MISSING IN 'CON' OPERAND IN LINE XXX --- FIELD TERMINATED BY ' ' or ';'

    Explanation: An unpaired parenthesis caused scanning of a CON statement to terminate at the first blank or semicolon following the error.

47. WARNING: NONPOSITIVE REPETITION FACTOR IN 'CON' PSEUDO-OP IN LINE XXX --- '1' ASSUMED

Explanation: Self-explanatory.

48. WARNING: UNPAIRED '<' IN LINE XXX

Explanation: An unpaired bracket was encountered during macro expansion; expansion of the current line was terminated and expansion continued with the next line.

49. WARNING: ILLEGAL SUBARGUMENT SPECIFIED IN LINE XXX --- ZERO INSERTED

Explanation: A non-existent subargument was encountered during macro expansion; the subargument zero was used.

50. WARNING: LINE XXX EXCEEDS 80 CHARACTERS --- LINE TRUNCATED

Explanation: Macro expansion resulted in an assembly source line of more than eighty characters; only the first eighty were assembled.

51. WARNING: BRANCH TO NONEXISTENT STATEMENT IN LINE XXX --- BRANCH NOT PERFORMED

Explanation: A program point in an AIF statement with a "true" condition was not defined in the macro containing the AIF.

52. WARNING: UNPAIRED PARENTHESES IN LINE XXX

Explanation: Self-explanatory.

The following are the assembler's fatal error messages:

53. ASSEMBLER INPUT LIMIT EXCEEDED AT LINE XXX --- REMAINDER OF INPUT IGNORED

Explanation: The source record input limit was exceeded (this includes <u>all</u> source lines, including macro definitions and macro expansions).

54. ILLEGAL MACHINE ADDRESS SPECIFIED IN LINE XXX --- ASSEMBLY ABORTED

Explanation: An illegal MIX address was specified in an ORIG statement, a BLOK statement, or an END statement.

55. UNEXPECTED END-OF-FILE ON SYSIN AFTER LINE XXX --- TRANSFER CARD NOT GENERATED

Explanation: The END card was missing from the source program, causing the assembler to read past the end of the input file; object code is non-executable.

56. ILLEGAL CHARACTER FOUND IN SYMBOL IN LINE XXX --- ASSEMBLY ABORTED

Explanation: A non-alphameric character was encountered during expression evaluation where a symbol character was executed; evaluation terminated at the point of error.

57. ILLEGAL OPERATION SPECIFIED IN EXPRESSION IN LINE XXX --- ASSEMBLY ABORTED

Explanation: Characters other than +, -, *, /, +, and // were encountered during expression evaluation where an operator was expected; evaluation terminated at the point of error.

The following are terminal error messages:

58. ASSEMBLER ERROR AT LINE XXX --- JOB TERMINATED

Explanation: The assembler detected a system error and abandoned the job; contact your MIX representative with your complete job listing.

59. LITERAL TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

Explanation: The number of MIXAL literal constants exceeded the system limit.

60. LOCAL SYMBOL TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: The total number of local symbol definitions exceeded the system limit.

61. SYMBOL TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: Self-explanatory (this includes all MIXAL symbols, set symbols, local symbol definitions, and literal constants).

62. OPCODE TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: User-defined opcodes have filled the assembler's opcode table.

63. OPCODE EXTENSION TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: User-defined opcodes have caused the assembler's auxiliary opcode storage to be filled.

64. MACRO DEFINITION TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: The number of characters in macro definitions has caused system table overflow.

65. PROGRAM POINT TABLE FULL AT LINE XXX --- ASSEMBLY TERMINATED

    Explanation: The number of branch points within macro definitions has caused system table overflow.

D. System Limits

System limitations on user programs are the input limit and the limits imposed by table sizes. The source record input limit is 999 source records. The limit on literal constants is 100 distinct constants (multiple use of the same literal causes no demand for extra storage). The symbol table can accomodate 400 symbols, set symbols, local symbols, and literals. A separate limit of 200 is imposed on

local symbol definitions. The opcode table has space for 250 opcodes. Of these 197 are system defined, leaving 53 available spaces for user-defined entries. The limitations caused by the opcode extension table are difficult to predict in general. This table has a capacity of 250 characters and/or MIX bytes, and is used to store opcodes exceeding four characters in length and the values of opcodes defined with full MIX-word W-values. The macro definition table accomodates 2000 characters, with blanks removed and system characters substituted. The program point table allows up to 200 program points to be defined within the macros of a source program. The macro expansion stack also imposes limits on macros, but concrete limits depend on the individual programs and macros being processed.

IV. The MIX Loader

A. Description

The MIX loader provides facilities for loading a MIX object program directly into MIX memory for subsequent execution. The loader accepts object decks produced by the assembler, or it may be used by programmers writing MIX programs directly in machine language, provided they are punched in the card format of exercise 26, section 1.3.1 of Knuth. This format is summarized below:

Program Cards

Columns 1-5: ignored

Column 6 : number of consecutive words to be loaded from this card (1 through 7)

Columns 7-10: location of the first word (-3999 through 3999)

Columns 11-20: first word (a decimal number)

Columns 21-30: second word (if column 6 $\geq$ 2)

. . .

Columns 71-80: seventh work (if column 6=7)

Transfer Card

Columns 1-5: TRANS

Column 6: 0 (zero)

Columns 7-10: location to which control is to be transferred for initiation of execution (-3999 to 3999)

The object deck thus consists of enough program cards to contain the program plus one transfer card which is required and must be physically the last card in the deck. Negative numbers, either as locations in columns 7-10 or as loadable words in columns 11-80, are indicated by

"11-punch" overpunched over the least significant digit; i.e., in columns 10, 20,..., or 80. Note that there are two differences from Knuth's format: negative loading locations are allowed (see the section on negative memory in Section II), and the user need not worry about overwriting the loading routine, since the MIX loader loads the memory directly and is not itself being executed by MIX.

B. Errors

There are two possible error messages that may be used by the MIX loader. Both are fatal and cause the job to be abandoned. The first one is:

MIX LOADERERROR --- ILLEGAL RECORD AT LINE NNNN; JOB TERMINATED

This message is issued if, for any reason, the contents of an input card cannot be interpreted or one or more of the words cannot be loaded. The other message is self-explanatory and is as follows:

MIX LOADERERROR --- TRANSFER CARD MISSING; JOB TERMINATED

If problems are encountered in loading, make sure that the transfer card is present and is the last card of the deck; ensure that the cards follow the required format and that negative numbers contain the proper overpunch; and be certain that all locations are within the bounds of MIX addresses and that all words fit into five MIX bytes for whatever bytesize is being used.