COMPUTATION RESEARCH GROUP
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA

CGTM NO. 185
APRIL 1977

MORTRAN 2.0 SYSTEM PROGRAMMER'S GUIDE

INTRODUCTION

The MORTRAN2 processor is an extension of the processor called MORTRAN that was released for general public use in July of 1973. We now refer to the 1973 processor (and language) as MORTRAN1, and to the current processor (and language) as MORTRAN2.

```
*******************************************************************
*                                                                 *
*                            CAVEAT                               *
*                                                                 *
*                                                                 *
*          This document describes the MORTRAN2.0 processor  as   *
*  of   June   1975.    The   internal   operators,   method   of *
*  implementation,  and  logical flow apply to MORTRAN2.0 and not *
*  to any successor or predecessor of MORTRAN2.0.  We reserve the *
*  right to change any or all of the  above.   We  will, however, *
*  attempt  to  maintain  upward  compatibility with the LANGUAGE *
*  called MORTRAN, and with macros that use  ONLY  those features *
*  described in Mortran User's Guides.                            *
*                                                                 *
*******************************************************************
```

This document is a first draft and is not a complete description of the processor. Extensions and revisions now being made to the processor will be reflected in future revisions of this document. More detail will be added. Your comments, suggestions, and criticisms will be appreciated. Send them to

A. James Cook
Computation Research Group, Bin 88
Stanford Linear Accelerator Center
P.O. Box 4349
Stanford, California, 94305

THE BOOTSTRAP

Both the MORTRAN1 and MORTRAN2 processors were implemented by means of a 3-level bootstrap. Each level of the bootstrap is functionally equivalent to the other two in the sense that at each level of the bootstrap the following steps are performed:

1. Compile a FORTRAN program.
2. The compiled program reads a set of macros that define another programming language.
3. The program reads a program in the new language and converts the program in the new language to FORTRAN.

The 1st level program is a one page FORTRAN program.

- Its 'macros' are FORMAT statements
- The 'language' defined by these macros is the 2nd level language whose statements are delimited by semicolons.
- A "block" in the level 2 language consist of the text between two successive occurences of the pair of characters "<0". The pair "<0" is translated by the 1st level program as follows:

```
follows:     <0     becomes     10 CONTINUE  (1st occurence)
                                20 CONTINUE  (2nd occurence)
                                30 CONTINUE  (3nd occurence)
                                and so on
```

Thus <0 defines a label that is the start of a block that may contain at most 10 labels. Other label definitions are of the form <n where n is a decimal digit. For example <4 becomes

```
14 CONTINUE  (1st occurence)
24 CONTINUE  (2nd occurence)
34 CONTINUE  (3nd occurence)
and so on
```

The statements within a block are either ordinary FORTRAN statements that have been terminated by a semicolon, or a string of macro parameters, signaled by the presence of a special character in the first position of the statement. Assuming that the following occur in the 2nd block:

Level 2 statement	Generated FORTRAN
.1NEK6;	IF(A(1).NE.A(K)) GO TO 26
#IEQJ8;	IF(I .EQ. J) GO TO 28
5;	GO TO 25
+M;	M=M+1
-M;	M=M-1
=VS;	A(V)=A(S)
*TS	READ(5,II)(A(II),II=T,S)

The 2nd level program is converted to FORTRAN and becomes the processor for the 3rd level language. Statements in the 3rd level language are delimited by semicolons. The 3rd level macros

- are terminated by the currency symbol ($);
- have parameters that are from 1 to 9 characters long, denoted by a pound sign (#)
- have the pattern and replacement parts separated by an exclamation point (!)
- have at most 9 parameters.

In the pattern part of a level 3 macro the digit following the pound-sign determines the number of characters in the parameter. The digit following the pound-sign in the replacement part of a level 3 macro denotes the position of the parameter in the pattern part.

Some examples of level 3 language constructs:

```
Level 3                 Generated
construct               FORTRAN

.X                      MM(X)

'X                      MM(location of character X)

(...)statement;         IF(...) statement

(...)<...>              IF(.NOT.(...))GO TO alpha
                        .
                        .
                  alpha CONTINUE

(...)<...ELSE...>       IF(.NOT.(...))GO TO alpha
                        .
                        .
                        .
                        GO TO beta
                  alpha CONTINUE
                        .
                        .
                        .
                   beta CONTINUE

(.S=';)                 IF(MM(S).EQ.MM(location of character ;))
                         NOTE: characters are never compared
                               except for equality.

<*...*>                 a 'forever' loop

<_...XIT..._>           forever loop with EXIT (jump out)

X:                      statement label definition

(...)X;                 IF(...)GO TO (statement label X)

X;                      GO TO (statement label X)
```

(X=Y)	IF(X .EQ. Y)
(X~Y)	IF(X .NE. Y)
(X>Y)	IF(X .GT. Y)
(X<Y)	IF(X .LE. Y)

Note especially the last example. The "less than" symbol generates "less than or equal". This was done so that we could get a full set of six relational operators by using the last four constructs and by exchanging the variables where necessary.

A particularly sticky problem in understanding level 3 macros is that the .NOT.'s in the above examples are not really generated. The level 3 macros apply De Morgan's laws to avoid generating .NOT. because IBM FORTRAN IV (level H) generates embarassingly bad machine code for logical IF statements containing the .NOT.'s. This is also the motivation for the NOT macros in the MORTRAN2 standard macro set.

The level 3 program is the MORTRAN2 processor. We are currently maintaining the processor in the level 3 language. We are also writing MORTRAN2 in MORTRAN2 and will release it if it proves satisfactory.

THE MORTRAN2 PROCESSOR

The processor consists of:

- A MAIN program
- Four SUBROUTINE subprograms
- Three INTEGER FUNCTION subprograms
- One LOGICAL FUNCTION subprogram
- One BLOCKDATA subprogram

Their names and very brief functional descriptions follow:

SUBROUTINE NXTCRD reads a line (card) from the MORTRAN2 input unit, writes the line on the MORTRAN2 output file and determines whether the line was a control line. If the line was a control line, the appropriate action is taken, after which a new line is read. If the line was not a control line the character pointer (NC) is set and control is returned to the main program.

SUBROUTINE OUTLIN writes one or more lines from the output buffer to the FORTRAN file after converting certain integers (e.g. statement numbers and the length of strings) to external representation.

SUBROUTINE NCV converts an integer from internal representation to external representation.

SUBROUTINE MACTRC is a "trace" debugging program, that is contolled by control cards. It will print any or all of the following on the MORTRAN file:

- The pattern and replacements parts as they exist in storage at the time macro is defined.
- The pattern and actual arguments at the time the macro is recognized,
- The text into which the macro expands.

INTEGER FUNCTION N1 converts a single digit from external representation to internal representation.

INTEGER FUNCTION N2 converts a one or two digit number from external representation to an integer in internal representation and checks whether this internal integer is within the allowable range. If the number is not within the allowable range, an error message is issued, and a default value is assigned to N2. If the number is within the allowable range, its value is assigned to N2. In either case, a normal return is made.

INTEGER FUNCTION NONO writes most of the error messages issued by MORTRAN2 and returns an integer value which is used by the main program to adjust certain pointers. If the error was "fatal" a STOP 16 is issued, otherwise the error count is increased by one, a value of -1 is assigned to NONO, and a normal return is made.

LOGICAL FUNCTION CNV converts a single digit from external to internal representation. If the digit is in the allowable range the function value is set to TRUE, otherwise it is set to FALSE.

MORTRAN2 uses two labeled common blocks /MEM/ and /STATUS/. STATUS is short (19 storage locations) and contains flags and pointers. The labeled common block /MEM/ contains a single array MM in which most of the manipulation occurs.

The array MM should be dimensioned as large as possible. On this tape (file 8 or file 11) it is MM(28 000). This number may be reduced if necessary. The test program (file 5 or file 7) was successfully run with MM(8 000). Dimensioning MM this small however, severely restricts the number of macros and the number of alpha-numeric labels the user may write without overflowing the buffers. The following is the layout of the array MM:

```
        COMMON /MEM/ MM(size)

                                  MM array element
------------------------------    1
Input line buffer
------------------------------    80
Decimal digits
------------------------------    90
Alphabet
------------------------------    115
Special Characters
------------------------------    151
Unused
------------------------------    180
Input unit stack
------------------------------    200
Macro Marker Stack
------------------------------    250
Label Stack
------------------------------    300
First Character Table
------------------------------    400
Actual Parameter Buffer
------------------------------    size/8              (3 500)
Macro Storage Buffer
------------------------------    (size/8)*5          (17 500)
Output Buffer

Expansion Buffer
------------------------------    size                (28 000)
```

Figure 1.

The parenthesized numbers on the extreme right in figure 1 are the values assuming that size=28 000. Changing the value of "size" is not difficult. The blank between the 28 and the 000 (twenty-eight thousand) was included in order to make this task easier if it must be done to the FORTRAN source. Changing the fixed values (400 and below) should not be attempted unless the processor has been fairly well understood.

Throughout this description, the word "pointer" will mean an INTEGER variable which may be used as a subscript of the array MM.

The output buffer and the expansion buffer do not have a fixed boundary between them; the output buffer grows in a forward direction, the expansion buffer toward the output buffer as shown in figure 2.

```
            -------------------- fixed at (size/8)*5
 IO--->      Output buffer
                  |
  O--->           |
                  |
+                 V
                  |
+                 A
  S--->           |
                  |
 IE--->        Expansion Buffer     fixed at (size/8)*7
                  |
  U--->           |
                  |
+                 V
            -------------------- fixed at (size)
```

Figure 2.

IO is a (fixed) pointer; it always points to the beginning of the output buffer. O (the variable "O") points to the last character stored in the output buffer. IE is also fixed and points to initial location of the expansion buffer. U points to the last character in the expansion buffer, that is, to the last character that has been transfered from the input line buffer. S (which is initialized to IE) points to the first character in the expansion buffer. The character pointed at by S is the first character that is eligible for matching by macro patterns.

It may be useful to think of the 'S' end of the expansion buffer as a stack from which matched text is popped and onto which replacement text is pushed each time a macro is successfully matched. The pushes and pops occur at S. The entire expansion buffer from S through U may be considered as a FIFO queue with respect to the input and output buffers. The characters transfered from the input line buffer are enqueued at U, and characters to be output are dequeued at S.

Figure 3 is a block diagram of the main program. The general function of the blocks are as follows:

INPUT	transfers characters from the input line buffer to the expansion buffer and calls NXTCRD to fill the input buffer as necessary
MATCH	attemps to find a match between the contents of the expansion buffer and the pattern part of macros.
EXPAND	replaces that part of the expansion buffer that has matched a macro pattern with the corresponding replacement text.
GENERATE	formats and stores new macro definitions.
NOMATCH	moves characters that have failed to match from the expansion buffer to the output buffer
OUTPUT	uses subroutine OUTLIN to empty the output buffer by writing out the generated FORTRAN statements.

```
                      ----------
                      | INPUT  |
                      ----------
                        |     |
+                             A
                        |     |
+                       V
                      ----------
            ---------> | MATCH  |<----------
           |          ----------           |
           |   success  |     |failure     |
           |            |     |            |
+                       V     V
           |      ----------  -----------   |
            -----| EXPAND |  | NOMATCH  |---
                  ----------  -----------
                  |     |     |      |
+                       A            A
                  |     |     |      |
+                 V           V
                  ----------  -----------
                  |GENERATE | | OUTPUT   |
                  ----------  -----------
```

Figure 3.

During the transfer of characters from the Input line buffer to the expansion buffer (INPUT block in figure 3),

- Comments are deleted

- The apostrophes that delimit literals (character strings) are replaced by internal string delimiters.

- Sequences of two or more blanks that are not within literals are replaced by a single blank

- The nesting level is determined by the "bracket count." That is, the count is increased by one for each left bracket and is decreased by one for each right bracket. (The bracket charaters are usually < and >.)

MATCH block (macro pattern matching)

If, during the pattern matching process, either the expansion buffer becomes empty or the macro being matched requires more characters to complete the match, a branch back to the input block is made.

Macro patterns are matched against text in the expansion buffer character by character, proceeding from left to right. Formal parameters in macro patterns are denoted by special symbols (usually #) and are delimited by character strings that are non-null and non-blank. When a formal parameter symbol is encountered in the pattern part of a macro, the corresponding actual parameter is saved in the parameter buffer, the layout of which follows:

```
        ---------------------------------------- fixed at 400
IA--->  Pointer to 1st actual parameter ---
        -------------------------------     |
IS--->  Pointer to 2nd actual parameter ---|---
        -------------------------------    |   |
        Pointer to 3rd actual parameter    |   |
        -------------------------------    |   |
                   -                       |   |
                   -                       |   |
                   -                       |   |
        -------------------------------    |   |
        1st actual parameter          <----    |
                   -                           |
                   -                           |
                   -                           |
        -------------------------------        |
        2nd actual parameter          <--------
                   -
                   -
                   -
        -------------------------------
        3rd actual parameter
                   -
                   -
                   -
        -------------------------------
                   -
                   -
                   -
        ---------------------------------------- fixed at (size/8)
```

Figure 4.

The following constraints apply to strings being saved as actual parameters

- A blank may not be the first character
- A blank may not be the last character
- Parentheses must be balanced
- Brackets must be balanced
- It may not contain a semicolon (except in a quoted string)
- Quoted strings must be saved in their entirety.

Violation of any of the above constraints results in aborting the parameter-saving process. When a failure to match any macro occurs, a single character is sent to the output buffer and matching begins again with the succeeding character. An exception to this "single character" transfer occurs when "no-rescan text" is encountered in the expansion buffer. In this case the entire string (delimited by quotation marks) is transfered to the output buffer.

EXPAND block (expansion of macros)

When a successful match has occured, the macro is expanded as follows

- The matched text in the expansion buffer is deleted
- The replacement part of the macro is substituted for the deleted text in the expansion buffer proceeding from left to right, character by character. During this process, the occurence of the special symbol # followed by a number causes the actual parameter corresponding to the number to be inserted. Also, the occurence of the special operators (signaled by the special symbol @) may cause other text to be inserted in the expansion buffer. (See below for a list of of special operators.)

The recognition of macro definitions is a special case of pattern matching (whether standard or user-defined). In this special case the macro being matched is a meta-macro which is "wired in". Its pattern part is

%'#'='#'

and its replacement part is the special operator @MG which causes control to be transfered to the "MACRO GENERATION" block (see figure 3.)

GENERATE block (macro definition and storage)

The discussion of macro generation requires a preliminary discussion of how macros are stored. The First Character Table (see figure 1) is a device for increasing the efficiency of the processor. It has the following form:

```
          ----------------------------------------------fixed at 300
IC--->    character(1)  | pointer to macro list (1)
          character(2)  | pointer to macro list (2)
          character(3)  | pointer to macro list (3)
              -             -
              -             -
              -             -
          character(k)  | pointer to macro list (k)
          ----------------------------------------------fixed at 400
```

Figure 5.

Each time a macro is "defined" the first character of the pattern part of the macro is entered in this table if it is not already there. If the character is in the table, the macro is added to the corresponding list. When a character in the expansion buffer is being considered as a candidate for the first character of a macro, it is matched against the entries in this table. If the character is found in the table, the matching process begins with the last macro entered in the list having this first character. Since, for the average program, the vast majority of the characters in the program fail this table search, many useless attempts at matching are avoided. In addition, even when the table search is successful, only those macros in the list having that particular first character are considered in the matching process. The macros are linked so that the last macro stored is the first macro in the list to be considered for matching. That is, the macros are stored as a linked-list implementation of a stack. The form of a macro in storage is:

```
---------------------------------------------
Pointer to next macro
---------------------------------------------
Pointer to beginning of replacement part
---------------------------------------------
Pointer to end of replacement part
---------------------------------------------
Pointer to catenation part of macro
---------------------------------------------
Pattern
part
of the
macro
---------------------------------------------
Replacement
part
of the
macro
---------------------------------------------
```

Figure 6.

Macro generation must be considered as two separate cases:

+ CASE ONE (the usual case)

- Store the macro beginning with the first free storage location in the macro storage buffer
- Search the first character table
- Modify (or add) the pointer in the first character table
- Set the pointers in the macro just stored

When the macro is being stored,

- the symbols representing formal parameters (in the standard version #) are changed to internal parameter symbols
- Pairs of the following symbols are replaced by a single symbol (so that the literal symbol may be used.): @,#, and '. (at-sign, pound-sign, and apostrophe.)

+ CASE TWO (catenation onto the replacement part of a previously defined macro.)

Recall that in the discussion of pattern matching, reference was made to a "wired-in" meta macro. A second meta-macro, which is read in like any other, permits the catenation of text onto the replacement part of a previously defined macro. The pattern part of this second meta-macro is

%'#'=*'#'

and its replacement part is the special operator @MC which causes case two to be done during macro generation. We will call the second actual argument of this second meta-macro the "catenation part." The steps for case two are:

- Find the macro whose pattern part matches the pattern given. (If none, treat as case one.)
- Change the catenation pointer field in the macro (or the last catenation part of the macro) to point to the free storage space in the macro storage buffer.
- Store the catenation part of the new macro in the macro buffer
- Modify the pointers so that the catenation text is effectively catenated onto the replacement part of the original macro.

The following is a description of the internal operators refered to in the discussion of macro expansion. If @ is encountered in the replacement part of the macro being expanded then the following operations occur at the time the macro is being expanded:

```
@LG     Label generator is increased by 10.
        @b is put into stream, where b is the (binary) value of the
           label generator
@Ic     Internal counter operations
        c is 0  counter is set to zero.     Nothing is put in stream.
        c is =  The context should be @b@I= where b is a binary integer.
                the internal counter is set to that integer.
                Nothing is put into the stream.
        c is +  counter is increased by 1.  Nothing is put in stream.
        c is -  counter is decreased by 1.  Nothing is put in stream.
        c is C  @(value of counter) is put in stream.

@MT     macro trace called if %Ti (i not equal zero) has been set.
        Nothing put in stream.

@MSc    The character c is pushed onto macro marker stack
@MU     @(top of macro stack) inserted into stream. Stack is popped
@MRc    Top of macro stack is replaced by c.
@MG     Generate a macro from #1 and #2.  Nothing put in stream
@MC     Catenate #2 onto replacement of macro whose pattern
        matches #1.  Nothing put in stream.

@MM    Prints the first actual argument (#1) on the MORTRAN file (6).

@Cx     sets conditional generation
        x is N sets "nogen"
        x is G sets "gen"
        x is E Marks end of gen or nogen.

@b@LSi   b is pushed (inserted) in label stack at level i  (0=top).
         (b=binary integer.)          Nothing to stream.
@LUi     Level i of label stack is unstacked. Nothing to stream.
@LCij    @b is inserted into stream, where b is j+(the value of the
         label stack at level i).
```

If @ is encountered in the pattern part of the macro, then:

```
#@n    causes the parameter designator # to match exactly n characters
       (n>0 and n<10 ) instead of an arbitrary number of characters.

       The external representation of the character @ (obtained by
       writing @@ in the macro definition) is treated as any other
       non-special character.
```

Any @ character (not in a quoted string) that remains in the text sent to the output buffer is expected to be followed by a binary integer. The output block will delete the @ and replace the integer with the integer represented by five decimal digits followed by a blank. Leading zeros are replaced with blanks.

The following is a list of variables (mostly pointers) that are used in the MORTRAN2 main program. An indication of the general use of the variable is listed to the right of each. The names in parentheses are the blocks in the main program in which the variable is used. Refer to figure 3 to identify the blocks.

- "Argument" is used interchangably with "parameter"
- "Cursor" and "pointer" are used interchangably

A	Argument cursor (EXPAND,MATCH,GENERATE)
B	Pointer to last character in pattern part of the macro (MATCH)
	Replacement cursor (EXPAND)
C	Character cursor (INPUT)
D	Pattern cursor (MATCH)
E	Retry cursor (MATCH)
	Partial terminator (EXPAND)
F	Pointer to free space in macro buffer (GENERATE,CATENATE)
G	Number generator (EXPAND)
H	Bracket count for nesting level (INPUT)
I	Pattern cursor (CATENATE), Temporary elsewhere
J	Temporary
K	Temporary
L	Temporary
M	Link pointer for macros (MATCH)
N	Temporary
O	Pointer to last character in output buffer
P	Parenthesis and bracket count (MATCH)
	Quote toggle (GENERATE)
Q	Pointer to top of macro marker stack
R	Pointer to top of label stack
S	Pointer to character for start of match (MATCH)
	Pointer to last character inserted (EXPAND)
T	Pointer to table of argument pointers (MATCH,GENERATE)
U	Pointer to last character in expand buffer (INPUT,MATCH)
V	Temporary cursor starting at pointer S (MATCH)
W	Temporary cursor starting at pointer V (MATCH)
X	UNUSED
Y	UNUSED
Z	Quote toggle (MATCH)