Bell Laboratories
Murray Hill, New Jersey

Computing Science Technical Report #2

THE M6 MACRO PROCESSOR

Andrew D. Hall

April 12, 1972

THE M6 MACRO PROCESSOR

by

Andrew D. Hall
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

ABSTRACT

M6 is a general purpose macro processor which processes a continuous stream of input text by copying it character-by-character to an output text unchanged except for selected portions known as macro calls.

The processor is coded entirely in FORTRAN IV in a way that is intended to be highly portable. Some details of the implementation are also described.

THE M6 MACRO PROCESSOR

by

Andrew D. Hall, Jr.
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

1. Introduction

M6 is a macro processor designed by M. D. McIlroy and R. Morris of Bell Telephone Laboratories and combines ideas from many sources [1,2,3,4]. The version described here is a translation of an earlier experimental version written in MAD by R. Morris and has been written in FORTRAN IV in a way that is intended to be highly portable.*

M6 receives a continuous stream of input text from an external source and copies it character-by-character to an output text unchanged except for selected portions known as macro calls. If a macro call or a quoted string (6) never occurs in the input text, the processor does nothing at all to the text stream as it passes through.

The beginning and end of a macro call are signaled by opening and closing warning characters. In this implementation, the character sharp (#) is used for an opening warning character and either colon (:) or semicolon (;) may be used for a closing warning character. The call itself consists of a series of arguments separated by commas (,) as in

```
#ADD3,A,B,C:
```

Upon encountering a sharp in the input text, the processor suspends transmission of characters to the output text and instead begins collecting the arguments of the call. When the closing colon or semicolon is found, the argument after the initial sharp, or "argument 0", is taken to be the name of the macro being called and is looked up in a table of macro definitions to find the replacement text. The entire call, including warning characters, is deleted and the replacement text substituted in its place. The scan resumes at the beginning or end of the substituted text depending on which warning character terminated the call.

For instance, if the name PSYMBOL is in the table of macro definitions with replacement text A6 then the appearance of

* M6 has been compiled and executed on the GE-635, IBM 360/65, CDC 6600, Univac 1108, PDP-10 and SIGMA 7 without any changes in the source code except for input-output unit numbers.

#PSYMBOL: in the input stream would be replaced in its entirety by A6. If the input text contained

```
FORMAT (#PSYMBOL:)
```

then the output text would receive

```
FORMAT (A6)
```

Occurrences of nested macro calls not enclosed in string quotes (6) will be evaluated as they occur during argument collection. For example, in

```
#ADD3,#PSYMBOL:,B,C:
```

the call of PSYMBOL will be evaluated first so that the call to ADD3 becomes

```
#ADD3,A6,B,C:
```

2. <u>Macro Definition</u>

The macro processor has a number of <u>built-in</u> macro definitions (8), the most important of which is the macro, DEF, for defining other macros. This macro is used in the form

```
#DEF,arg1,arg2:
```

where arg1 is the name of a macro to be defined and arg2 is the replacement text to be associated with the name.

Aside from making an entry in the table of macro definitions, DEF has no effect, for its own replacement text is the null string. We can now define the macro PSYMBOL by the call

```
#DEF,PSYMBOL,A6:
```

If the input text is

```
#DEF,PSYMBOL,A6:FORMAT (#PSYMBOL:)
```

the output text is

```
FORMAT (A6)
```

If a call of DEF redefines a macro, the new definition supersedes the old. Also, if a macro has never been defined and is called, then the macro whose name is the null string is looked up and used. Initially this macro has null replacement text but it can, of course, be redefined.

3. Evaluation

We speak of the macro processor as evaluating its input. The way text is evaluated depends on whether it is part of a call, part of replacement text, or part of a string quotation. The following sections describe the process of evaluation in considerable detail.

4. Argument Substitution

In the replacement text of a macro, a dollar sign ($) followed immediately by a digit acts as a parameter. When a macro has been called, all occurrences of $0, $1, ..., $9 in the replacement text are each replaced by the corresponding argument in the call. Parameters for which no argument has been supplied are replaced by null strings. For example the input text

```
#DEF,ADD3,$1 = $2 + $3:#ADD3,X,Y,Z:
```

would yield

```
X = Y + Z
```

Since only the first ten argumnets may be referenced by parameters in replacement text, special conventions have been established for the collection of argument 9 which make it possible to write macros that have more than ten arguments (7).

5. Recursive Evaluation

The process of evaluating a macro call can be thought of as occurring in two steps:

First, the arguments of the call are substituted in the replacement text for occurrences of $0,$1,...,$9, regardless of any string quotes (6) which may be present in the replacement text.

Next, the text resulting from this substitution replaces the entire call, including the warning characters. If the original call was terminated by a colon, then scanning of the resulting input text resumes at the beginning of the substituted replacement text. If the original call was terminated by a semicolon then the scan resumes immediately after the substituted replacement text.

6. String Quotation

To permit warning characters to appear in text and be treated as ordinary characters, the macro processor recognizes a left-angle-bracket (<) and a right-angle-bracket (>) as string quotes. In a string enclosed in quotes, the characters sharp, comma, colon, semicolon are not recognized as special characters. Nested occurrences of string quotes must be balanced. When a quotation is evaluated the outermost pair is removed.

For example, the input text

```
<#A:>
```

is not a macro call, because the surrounding quotes exempt #A: from special status. Consequently the text evaluates to

```
#A:
```

The macro, QSYMBOL, can be defined to have the replacement text

```
A4,A2
```

by the following definition

```
#DEF,QSYMBOL,<A4,A2>:
```

7. Argument 9

Unlike arguments 0 through 8, argument 9 is collected without recognizing comma, sharp or string quotes as special characters. Thus, all arguments occurring after the ninth comma are collected as one string which then becomes argument 9.

For example, the macro, CONCAT, which simply concatenates up to 16 arguments, would be defined as follows:

```
#DEF,CONCAT,<$1$2$3$4$5$6$7$8#CONCAT9TO16,$9:>:
#DEF,CONCAT9TO16,<$1$2$3$4$5$6$7$8>:
```

8. Built-in Macro Definitions

In order to facilitate the writing of new macro definitions, a number of useful macros have been initially defined. Where an argument is interpreted as an integer, its value is found by taking the longest initial substring of digits (perhaps preceded by a sign) as a decimal number. If the initial substring of digits is null, the value is taken as 0.

The calls for the built-in macros are as follows:

#DEF,arg1,arg2:

A macro named arg1, with replacement text, arg2, is defined. The replacement text of DEF is null.

#COPY,arg1,arg2:

A macro named arg2 is defined with replacement text identical to that of the macro named arg1. The replacement text of COPY is null. For example

```
                    #COPY,ADD,+:
```

defines a macro + which has the same replacement text as ADD.

The macros DEF and COPY are used to define or rename macros. They work for built-in macros as well as macros previously defined by DEF and COPY.

#SEQ,arg1,arg2:
#SNE,arg1,arg2:

The replacement text of SEQ is 1 if arg1 is identical, character by character, to arg2. Otherwise the replacement text is 0.

The replacement text of SNE is 1 if arg1 is <u>not</u> identical to arg2. Otherwise the replacement text is 0.

For example,

```
              #SEQ,ABC,AB:#SNE,ABC,AB:
              #SNE,1,#SEQ,1,1::
```

would be replaced by

```
              01
              0
```

```
#GT,arg1,arg2:
#GE,arg1,arg2:
#LT,arg1,arg2:
#LE,arg1,arg2:
#EQ,arg1,arg2:
#NE,arg1,arg2:
```

The replacement text of GT, GE, LT, LE, EQ or NE is 1 if arg1 is respectively greater than, greater than or equal to, less than, less than or equal to, equal to, or not equal to arg2. The arguments, arg1 and arg2, are interpreted as integers. For example

```
#EQ,0,0:
```

and

```
#EQ,0:
```

would both be replaced by 1.

```
#IF,arg1,arg2,...,argn:
```

The arguments arg1,arg2,... are considered in pairs from left to right. If the left argument of a pair is the string 1, then the replacement text of IF becomes the right argument in the pair. If none of the left arguments in the pairs are 1, then the replacement text of IF is null.

For example,

```
#IF,0,arg2,1,arg4:
```

will be replaced by arg4.

```
#GO,arg1:
```

GO is a macro which allows conditional evaluation of the replacement text of a macro. If GO is evaluated as part of a replacement text and arg1 is equal to the string 1, the remainder of the replacement text is ignored. Otherwise GO has no effect. In either case, GO is replaced by the null string.

For example, if SPEECH is defined as follows:

```
#DEF,SPEECH,<NOW IS THE HOUR #GO,$1:FOR...>:
```

then

```
#SPEECH,1:
```

is replaced by

NOW IS THE HOUR

and

#SPEECH,2:

is replaced by

NOW IS THE HOUR FOR...

#GOBK,arg1:

GOBK is similar to GO, except that evaluation of the replacement text is restarted from the initial character.

#SIZE,arg1:

SIZE is replaced by the length of arg1 in characters.

#SUBSTR,arg1,arg2,arg3:

SUBSTR is normally replaced by the substring of arg1 beginning at character position arg2 and having length arg3. A negative arg3 is taken to be 0, and a null arg3 as arbitrarily large. In case of an improper substring, whose ends lie outside arg1, only its intersection with arg1 is taken.

#ADD,arg1,arg2:
#SUB,arg1,arg2:
#MPY,arg1,arg2:
#DIV,arg1,arg2:

ADD, SUB, MPY and DIV are replaced by the sum, difference, product and integer quotient of arg1 and arg2, respectively. The arguments, arg1 and arg2, are interpreted as integers. Overflow conditions are not checked and if arg2 is 0 in DIV, the result is the null string.

#EXP,arg1,arg2:

If arg2 is negative, the result is the null string. If arg2 is zero, the result is 1. Otherwise, the result is arg1 raised to the arg2-th power. Overflow conditions are not checked.

#DNL:

DNL reads the source stream through the occurrence of the next new-line character (10.2) and throws it away. DNL has null replacement text. DNL is used to delete unwanted new-lines from the source text. For example

```
#DEF,PSYMBOL,A6:
```

and

```
#DEF,PSYMBOL,#DNL:
A6:
```

are equivalent. If the #DNL: were not included in the latter call of DEF, then a new-line character would be included in the replacement text of PSYMBOL immediately preceding A6.

#SOURCE,arg1,arg2:

After the next new-line character is processed, the current input unit number will be "pushed down" and the input unit set to arg1. If arg2 is not null, the new unit will be rewound before use. The occurrence of an END macro will "pop" the the input unit to its previous value.

#END:

After the next new-line character is processed, the unit number will be "popped" to the value most recently saved by a SOURCE macro call. If the stack is empty when END is called, processing will be terminated.

#TRACE,arg1:

If arg1 is 1, trace mode is set on, otherwise off. When in trace mode, the level of each macro call and the first ten characters of each argument will be printed (10.1) as the macro calls are encountered during processing. The new-line character is printed as a blank.

9. Examples

The following examples illustrate some useful and interesting techniques.

Conditional replacement - IF

Suppose MIN is a macro to be called with two arguments both of which are integers and is to be replaced by the smaller of the two arguments. One way to write MIN is:

```
#DEF,MIN,<#IF,#LT,$1,$2:,<$1>,1,<$2>:>:
```

Redefinition

Quite often it is necessary to have a method for generating "created symbols" when using macros. For instance, when using macros to generate FORTRAN DO loops it is necessary to have a unique label every time a loop is generated. This can be accomplished as follows:

```
#DEF,CRSN,0:
#DEF,CRS,<#DEF,CRSN,#ADD,#CRSN:,1::#CRSN:>:
```

The initial value of CRSN is defined to be 0. Each time CRS is called, the definition of CRSN is incremented by 1 and the call of CRS replaced by the new value. At any time, the last symbol created can be obtained by calling CRSN.

GO

Suppose it is desired to write a macro, STARS, which has one integer argument. The call

```
#STARS,n:
```

is to be replaced by n asterisks ($n \geq 0$). STARS can be defined as follows:

```
#DEF,STARS,<#GO,#EQ,$1,0::*#STARS,#SUB,$1,1::>:
```

The use of GO in the replacement text of STARS will cause the replacement text which follows to be ignored when STARS is called with 0 as an argument. Thus the call

```
#STARS,2:
```

is evaluated in three steps, as follows:

```
                    *#STARS,1:
                    **#STARS,0:
                    **
```

String Quotes

The following two examples illustrate some of the effects of string quotes.

(a)

```
#DEF,X,$1:#X,Y:
```

and

```
#DEF,X,<$>1:#X,Y:
```

are both replaced by

```
Y
```

while

```
#DEF,X,<<$>1>:#X,Y:
```

is replaced by

```
$1
```

(b)

```
#DEF,A,#DEF,B,$1::-#A,GOSH:-#B,GEE:
```

is replaced by

```
--GEE
```

while

```
#DEF,A,<#DEF,B,$1:>:-#A,GOSH:-#B,GEE:
```

is replaced by

```
--GOSH
```

but

```
#DEF,A,<<#DEF,B,$1:>>:-#A,GOSH:-#B,GEE:
```

is replaced by

```
-#DEF,B,GOSH:-
```

10. Implementation Notes

10.1. Input-output Unit Assignments

The current implementation reads input text from logical unit 5 and writes output text on logical unit 43. Diagnostics, trace output and run statistics are written on logical unit 6.

If necessary, these unit assignments can be changed by modifying the appropriate DATA statements in the BLOCK DATA subprogram.

10.2. Treatment of Input Text

All reading of input text is handled by the logical function RDCHAR which reads records (card images) under an 80A1 format. Only the first 72 characters of an input record are considered significant so that the last 8 characters may contain sequence information. Trailing blanks are deleted and replaced by a "new-line" character.

10.3. Treatment of Output Text

M6 collects output characters until a new-line character occurs or a line exceeds 72 characters, at which time the line is padded with blanks to 72 characters and the sequence field of the last input line appended. The line is then written on the output file.

10.4. Implementation Parameters

The program contains 25 subroutines, totaling about 600 executable statements.

In the present implementation there is room for 250 distinct macro definitions of which 25 are already used for built-in definitions. Table entries of macros that have been redefined will be reused. The number of definitions permitted can be changed by adjusting the length of the COMMON regions named MLISTN, MLISTD, MLISTT, MLISTU and MLISTL and by adjusting the value of MFREE which is initialized in the BLOCK DATA subprogram.

About 12000 characters of string storage are available for macro names and corresponding replacement texts. Each character is stored as a FORTRAN INTEGER variable. This storage is also used for the temporary storage of argument strings during macro evaluation so that storage can be exhausted even though no new definitions are made. The amount of string storage available can be changed by adjusting the length of blank COMMON and by adjusting the value of LENGTH which is initialized in the BLOCK DATA subprogram.

The maximum recursive depth for macro calls is around 60 and depends on the number of arguments appearing in calls at each level. This maximum depth can be changed by adjusting the length of the COMMON region named PDLS and the value of the variable OPTR which is initialized in the BLOCK DATA subprogram.

The maximum depth of the stack of input units is 10 (see SOURCE and END in 8).

10.5. Diagnostic Messages

M6 can give diagnostic messages. Four of the diagnostics pertain to table limitations and are as follows:

```
STORAGE EXHAUSTED

PUSH DOWN LIST OVERFLOW

TOO MANY DEFINITIONS

INPUT STREAMS NESTED TOO DEEPLY
```

There are two diagnostics indicating an internal M6 error, as follows:

```
INCORRECT CALL TO LOG2

PROCESSOR ERROR
```

10.6. Improving Performance.

Measurement has shown that the major overhead in M6 is incurred in the execution of the subprograms STREQ, RDCHAR, WRCHAR and WRBUFF. The execution speed of M6 is approximately doubled by rewriting these routines in assembly language.

References

1. C. Strachey, A General Purpose Macrogenerator, Comput. J. 8, 3 (Oct. 1965) pp. 225-241.

2. C. N. Mooers and L. P. Deutsch, TRAC, A Text Handling Language, Proc. ACM 20th Nat. Conf. (1965), pp. 229-246.

3. IBM 7090/7094 IBSYS Operating System: Version 13, Macro Assembly Program (MAP) Language, Form C28-6392-3.

4. M. D. McIlroy, Macro Instruction Extensions of Compiler Languages, CACM 3 (1960) pp. 560-571.