**MASTER EN SCIENCES INFORMATIQUES**

# Proposition d'amélioration de l'implémentation du macro-processeur M6 - Annexes

Orlans, Michel

*Award date:*
1983

*Awarding institution:*
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE - DAME DE LA PAIX

NAMUR

Institut d'informatique

Année academique 1982 - 1983

# Proposition d'amélioration de l'implémentation du macro-processeur M6

## - annexes

Promoteur :

Professeur C. CHERTON

Mémoire présenté pour
l'obtention du diplôme de licence
et maîtrise en informatique par

Michel ORLANS

# ANNEXES

Spécifications de M6

Modifications locales

Spécifications de GSYS

Programme source

# Annexe 1: SPECIFICATIONS DE M6

## 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 call.

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

# THE M6 MACRO PROCESSOR

by

Andrew D. Hall
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.(1)

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 (section 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 encontering 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 beeing 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 then resumes at the beginning or end of the substituted text depending on which warning character terminated the call.

---

(1) M6 has been compiled and executed on the GE-635, IBM 360/65, CDC 6600, UNIVAC 1100, PDP 10 and SIGMA 7 without any changes in the source code except for input-output unit numbers.

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

Occurences of nested macro calls not enclosed in string quotes (section 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. Macro Definition

The macro processor has a number of built-in macro definitions (section 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 the macro to be defined and arg2 is the replacement text to be associated with the name.

Aside from making a new 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 to 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 occurences 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 yeld

```
X = Y + Z
```

Since only the first ten arguments 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 (section 7).

## 5. Recursive Evaluation

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

- First the arguments of the call are substituted in the replacement text for occurences of $0, $1,..., $9, regardless of any string quotes (section 6) which may be present in the replacement text.
- Next, the text resulting from this substitution replace 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, colon, comma, semicolon are not recognized as special characters. Nested occurences 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 definition

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

## 7. Argument 9

Unlike argument 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 is null, the value is taken to be 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 not 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 or equal to, less than, less 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 as no effect.In either case, GO is replaced by the null string.

For example, if SPEECH is defined as follow :

#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 lenght arg3.A negative arg3 is taken to be 0, and a null arg3 as arbitrarily large.In case of an improper substring, whose end lie outside arg1, only the 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.

#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 occurence of an END macro will "pop" the input unit number 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.

#DNL:

DNL reads the source stream through the occurence 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.

## 9. Examples

The following examples illustrate some useful and interesting techniques.

### Conditionnal 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 created symbol 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 >= 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.

example 1: `#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
```

example 2: `#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 countain 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 lengths of the COMMON regions named MLISTN, MLISTD,MLISTT,MLISTU,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 maximun 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 follow :

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 is assembly language.

## 11. References

1. C. Strachey, A General Purpose Macro Generator, 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 Extentions of Compiler Languages, CACM 3 (1960) pp. 560-571.

# Annexe 2: MODIFICATIONS LOCALES

## 1. NAME

m6 - general purpose macroprocessor

## 2. SYNOPSIS

m6 [ name ]

## 3. DESCRIPTION

M6 copies the standard input to the standard output, with substitutions for any macro calls that appear. When a file name argument is given, that file is read before the standard input.

The processor is as described in the reference with these exceptions:

- #def,arg1,arg2,arg3: causes arg1 to become a macro with defining text arg2 and (optional) built-in serial number arg3.

- #del,arg1: deletes the definition of macro arg1.

- #end: is not implemented.

- #list,arg1: sends the name of the macro designated by arg1 to the current destination without recognition of any warning characters; arg1 is 1 for the most recently defined macro, 2 for the next most recent, and so on. The name is taken to be empty when arg1 doesn't make sense.

- #warn,arg1,arg2: replaces the old warning character arg1 by the new warning character arg2.

- #quote,arg1: sends the definition text of macro arg1 to the current destination without recognition of any warning characters.

- #serial,arg1: delivers the built-in serial number associated with macro arg1.

- #source,arg1: is not implemented.

- #trace,arg1: with arg1 = '1' causes a reconstruction of each later call to be placed on the standard output with a call level number; other values of arg1 turn tracing off.

The built-in `warn' may be used to replace inconvenient warning characters.
The example below replaces `#' `:' `<' `>' by `[' `]' `{' `}'.

```
#warn,<#>,[:
[warn,<:>,]:
[warn,[substr,<<>>,1,1;,{]
[warn,[substr,{{>>,2,1;,}]
[now,{calls look like this}]
```

Every built-in function has a serial number, which specifies the action to be performed before the defining text is expanded. The serial numbers are: 1 gt, 2 eq, 3 ge, 4 lt, 5 ne, 6 le, 7 seq, 8 sne, 9 add, 10 sub, 11 mpy, 12 div, 13 exp, 20 if, 21 def, 22 copy, 23 warn, 24 size, 25 substr, 26 go, 27 gobk, 28 del, 29 dnl, 32 quote, 33 serial, 34 list, 35 trace. Serial number 0 specifies no built-in action.

## 4. SEE ALSO

A. D. Hall, M6 Reference Manual. Computer Science Technical Report #2, Bell Laboratories, 1969.

## 5. DIAGNOSTICS

Various table overflows and ``impossible'' conditions result in comment and dump. There are no diagnostics for poorly formed input.

## 6. AUTHOR

M. D. McIlroy

## 7. BUGS

Provision should be made to extend tables as needed, instead of wasting a big fixed core allocation. You get what the PDP11 gives you for arithmetic.

Annexe 3: SPECIFICATIONS DE GSYS

```
**********************************************************************
*                                                                    *
*                      M6 G-SYSTEM USER MANUAL                       *
*                                                                    *
**********************************************************************
```

This system offers the following macros to the user :

- G_UMANUAL to generate this user's manual,
- G_IMANUAL to generate explanations about its own implementation,
- G_SYS to generate the system itself, that is a file which is to be used under M6 to have the facilities described in this manual,
- C which can be used to include comments in the user's source file,
- SKIP used to facilitate user's source file formating,
- PROC to define user's procedures,
- UPDMRG to declare registers pairs which are pos sibly modified in a user's macro.

The system uses internal macros for its own work which names are all prefixed by 'g_'. It is under the user's responsibility not to use or define macros having such a name. Moreover, for local symbols defined by the user, the system generates in the output file symbols of the form 'An' where 'n' states for any decimal number. It is also the user's responsibility not to use such symbol in its own code.

The system is self suffisant in the sense it doesnt need any other system to work : just work it under M6.

## THE C MACRO.

The C macro is just to allow the user to write long comments in his source file which do not appear in the output. To use it, just write '<#C.<com>!>' in the input file, where 'com' states for the wanted comment. The only restriction to the content of 'com' is that angle brackets are to be balanced. In particular, the comment may contain as many return carriages as needed.

## THE SKIP MACRO.

The SKIP macro allows easy input file formating. It just skips all input characters up to and including the (n+1)th carriage return, where n is the value of the SKIP argument. In particular, SKIP without any argument is equivalent to built-in M6 macro 'dnl'.

## THE PROC MACRO.

The macro PROC allows the user to define fairly general procedures. To understand how it can be used, it is necessary to introduce the concepts of 'opened' and 'closed' code related to a procedure.

### 1. OPENED CODE.

The so called 'opened' code of a procedure is a code sequence which is generated at each call of that procedure. it corresponds to the usual concept of calling sequence. The difference is that, in this system, no general conventions have been made about a standart calling sequence : it is under the user's responsibility to define the calling sequence for each procedure he declares. This definition is given as part of the procedure definition and thus, as soon as a procedure has been defined, the user is not more concerned with the particularities of the calling sequence for that procedure : he just call it by its name, specifying args in a standart way in the call. This allows the user to tailor a calling sequence specific to each procedure, for a better use of machine capabilities. Moreover, that system offers the user the possibility to define so called 'open procedures', that is procedures which code is generated at each call.

## 2. CLOSED CODE.

Closed code is the code related to a procedure which is assembled only once and which is called for by the calling sequence. It replaces the procedure definition itself.

## 3. LOCALS.

The concept of local symbols is a wellknown notion we have not to explain here. The only difference with the usual sense is that locals may be defined also in the opened code of a procedure. In the definition of procedure codes, locals are to be referenced as '<#lcnm>!' where 'lcnm' is the local name. As allready stated, locals are replaced in the generated code by a system-generated symbol of the form 'An' where 'n' is a decimal number.

## 4. REGISTERS SAVING.

Because of 8085 hardware particularity, registers are saved by pairs : <a,flags>, <b,c>, <d,e> and <h,l>. As the saving mechanism saves registers pairs on the stack, it is meaningless to save the sp regiter. The saving is selective in that only the registers pairs which have been declared as to be saved in the call and which are possibly modified by the procedure execution are effectively saved. To achieve that, the first argument of a call is the list of register pair names the user wants to keep the values. The register pair names are as in the 8085 PUSH and POP instructions, that is 'a', 'b', 'd' and 'h'. The list has to be enclosed in angle brackets '<' and '>'. Moreover, in order to let the system to know about the registers pairs possibly modified by procedure execution, it is necessary to indicate in each procedure definition which are those pairs. To do that, the user must give as second argument to PROC the registers pairs names which are explicitely modified by machine instructions of the code. The system will automatically take into account the registers pairs possibly modified by any imbeded procedure calls.

## 5. PROCEDURE DEFINITION FORMAT.

To define a procedure, the user has to invoke PROC macro giving it up to 6 arguments as follows :

<#PROC.prnm.<mrg>.<olc>.<ocd>.<clc>.<ccd>!>

where

prnm = defined procedure name,
mrg = list of registers pairs names corresponding to all the registers possibly modified by machine instructions appearing in either the opened or closed code of the definition,
olc = the list of all locals used in the opened code,
ocd = the opened code itself, where locals are to be referenced as <'#lcnm>!' where 'lcnm' is the local name as it appears in olc,
clc = the list of all locals used in the closed code,
ccd = the closed code itself, where locals are to be referenced as <'#lcnm!>' where 'lcnm' is the local name as it appears in olc.

The PROC call has of course generally to be written on more than one line. The following rules are to be observed in order to generate code in a proper format :

- any return carriage between arg's have to be masked using 'SKIP' macro,
- codes are to be written in correct assembler format. The user is advised to write the codes enclosing each line between angle brackets in order to make the things clear,
- any user's macros can be used defining the procedures codes. However, in order to give the system the possibility to know about used registers pairs, those macros are to be defined using macro UPDMRG described later on. Moreover, those macros are to be defined before their use in any PROC argument,
- any user's procedure used in code definition of a procedure has to be defined previously. As a consequence, no recursive calls are allowed.

## 6. USER'S PROCEDURE CALL.

As a user procedure has been defined using PROC, it may be called according to the following format :

```
<       #prnm.<srg>.arg1.arg2. ... arg8!>
```

where prnm = invoked procedure name,
srg = the list fo registers pairs name corresponding to the registers the user wants to keep the values,
argI = the Ith argument of the procedure.

## 7. PARAMETERS.

In the opened code of a procedure, dummy arguments are to be represented as '$i' where 'i' is the order of the actual argument in the call PLUS ONE. Of course, no dummy argument may be present in the closed code of a procedure as it is the user's responsibility to define open code in order to insure the arg's transfer.

It is quite possible to use arguments in a call which are themselves macro call. If they are written with an ending expantion itself which is used as actual argument. If they are written with an ending '<;>', the actual argument is of course the non-expanded replacement text of the invoked macro. It is also possible to put the macro call in angle brackets in which case the actual argument is that macro call itself.

## THE UPDMRG MACRO.

**********

The UPDMRG macro allows the user to define his own macros in such a way that they may be used in opened and closed code definitions of procedures he wants to define without beeing obliged to remember which are the registers used by the code assembled by those macros.

To use it, just write :

<#UPDMRG.mrg!>

where mrg = the list of register pair names corresponding to registers possibly modified by code generated by the macro,

in the macro definition, with eventual preceeding and following carrige return are masked by using SKIP macro. That doesnt affect the code generated by that macro but, using it in a procedure code definition, it indicates to the system that indicated register pairs are possibly modified by the generated code, in such a way that the system has the possibility to save them if requiered.

# Annexe 4: LE PROGRAMME SOURCE

Le programme source est réparti dans dix fichiers. La suite de cette annexe comporte le texte de ces dix fichiers ainsi que des commentaires sur leur contenu.

## 1. M6.h

M6.H est le fichier contenant la description des données globales du système. Il reprend également la définition de quelques constantes . Cette description est insérée en tête des autres fichiers par un ordre "include" au pré-processeur du compilateur C .

Ces données sont ainsi considérées comme des variables externes par le compilateur lorsqu'il traite les autres fichiers .

Ces données globales sont :

- get , la table de réalisation des appels avec sa limite supérieure , gmax , sa limite inférieure , g0 , sa première placee libre , ge , et l'entrée courante , gf .

- put , la table de collection des appels avec des pointeurs identiques à ceux de la pile get , p0 , pmax , pe et pf .

- les indicateurs :

  lp : niveau d'output (où l'on écrit)
  lg : niveau d'input (où l'on lit)
  lq : niveau de quote (combien de "lquote" non équilibrés)
  l9 : niveau apparant d'appel dans le dixième argument (si l'on est dans un dixième argument et combien de sharp ont étés rencontrés)

- Les switches du système :

  rescan : 1 si le texte de remplacement de l'appel en cours de réalisation doit être réévalué , 0 sinon

  traceflag : 1 si les appels collectés doivent être tracés sur l'output standard , 0 sinon

- Les tables du système :

meta : pour stocker les caractères spéciaux
loaddef : pour stocker les données nécessaires au chargement des macros pré-définies
dummy : structure de définition d'une macro "bidon" qui , quand elle est évaluée , fait analyser le texte de l'argument numéro 0 de l'appel courant (par exemple , quand une macro pré-définie M6 possède un texte de remplacement , on s'arrange pour que ce texte devienne l'argument numéro 0 de l'appel courant puis on fait évaluer la macro "dummy")
getfree : structure de définition d'une macr "bidon" qui , quand elle est évaluée , fait libérer la pile get des textes des arguments de l'appel courant
one : contient les caractères '1' et 0 et est utilisée pour comparer un argument avec le caractère '1'
def : contient les 255 pointeurs vers les premières structures de définition de diverses sous-listes du système de gestion des macros
c : contient le caractère courant à évaluer

- Les constantes utilisées :

les diverses tailles des tables et des records utilisés
les caractères spéciaux
la taille de la table de pointeurs (def)

- Les records (types) utilisés :

(ldnext, ldswitch, ldident) décrit une macro pré-définie M6 dans la table loaddef

(fct, argum1, argum2, nextcps) decrit un élément de liste de définition avec l'adresse fct d'une procédure à exécuter , les deux arguments à donner à cette procédure et l'adresse du prochain record de la liste , nextcps

(next, dswitch, dtext, dtype, dlist, dident) décrit une structure de définition de macro avec l'adresse de la prochaîne structure dans la sous-liste , next , le switchw de la macro et son type ainsi que les adresses de son texte de définition , de son nom et de sa liste de définition

(prev, mframe, mchar, marg, ga0) décrit un objet manipulé sur la table get avec l'adresse de l'objet précédent dans la table , l'adresse de la structure de définition de la macro dont l'appel est courant , l'adresse du prochain caractère à évaluer , mchar , l'adresse dans la pile get du prochain caractère du texte d'un argument ou 0 , mchar , et l'adresse dans la pile get de l'argument numéro 0 de l'appel courant , ga0

(prev, pan, pa0) decrit un objet manipulé dans la table put avec l'adresse de l'objet précédent dans la pile put , prev, le numéro de l'argument que l'on est occupé à collecter , pan , etl'adresse dans la table put de l'argument numéro 0 de l'appel que l'on collecte

```
struct {
        int word;
};

struct {                    /* "put stack", currently gathering args */
        int prev;           /* previous frame ptr, self-relative */
        int pan;            /* argument number of arg being collected */
        int pa0;            /* self-relative ptr to arg0 */
};

struct {                    /* "get stack", currently expanding macros */
        int prev;           /* prev frame ptr, self-realtive */
        int mframe;         /* ptr to macro definition frame */
        char *mchar;        /* next char adress */
        int marg;           /* 0 or ptr to next arg char relative to gf */
        int ga0;            /* arg0 ptr self-rel */
};

struct {                    /* "definition stack" */
        int nextst;         /* next frame adress */
        int dswitch;        /* builtin func code, neg for dead definition */
        char *dtext;        /* definition text adress */
        int dtype;          /* type of the macro */
        int dlist;          /* adress of the first compilst structure or 0 */
        char *dident;       /* adress of first char of identifier naming def */
};

struct {                    /* compile structure ; for user-pre-defined macros */
        int fct;            /* pointer to a function to be executed */
        int argum1;         /* first actual parameter of the call */
        int argum2;         /* second actual parameter of the call */
        int nextcps;        /* adress of next compile structure or 0 */
};

struct {                    /* loaddef structure */
        int ldnext;         /* next frame pointer, self-relative */
        int ldswitch;       /* buildin definition code of m6-pre-defined macro */
        int ldident;        /* first character of macro name */
};

char metas[];               /* to store the special characters */
char loaddef[];             /* data to initialise definition table */
char dummy[];               /* dummy empty macro */
char getfree[];             /* all 0 */
char one[];
int def[];                  /* def table entry */
char c;                     /* current input character */
int rescan;
int traceflag;
```

```
int lp;                 /* arg collection level (out level) */
int lg;                 /* input level (get level) */
int lq;                 /* quote level */
int l9;                 /* apparent call level within arg 9 */

char *pf;               /* put stack frame ptr */
char *pe;               /* put stack end */
char *pmax;             /* top of put stack */
char p0[];              /* put stack */
char *gf;               /* get stack frame ptr */
char *ge;               /* get stack end */
char *gmax;             /* get stack limit */
char g0[];              /* get stack */

#define pend 6
#define dend 12
#define cend 8
#define lquote metas[0]
#define rquote metas[1]
#define sharp metas[2]
#define colon metas[3]
#define semi metas[4]
#define comma metas[5]
#define dollar metas[6]
#define NMETA 7
#define DSIZE 256       /* same value as in m67.c */
```

## 2. M67.c

ce fichier contient les initialisations du système qui peuvent être faites au moment du chargement du programme , c'est-à-dire :

- les tailles des tables
- les réservations de place pour ces tables
- les données nécessaires à l'initialisation du sous-système de gestion des macros (les noms et les switches des macros pré-définies)
- les valeurs initialisant les structures de définition des macros "bidon" dummy et getfree
- les valeurs initiales des switch traceflag et rescan

```
#define LDSTSIZE 350
#define PSIZE 2000
#define GSIZE 2000
#define DSIZE 256       /* same value as in m6.h */

int pmax PSIZE;
char p0[PSIZE];

int gmax GSIZE;
char g0[GSIZE];

int def[DSIZE];

char loaddef[LDSTSIZE] {
         6,0,  0,0, 0,0,                    /* empty macro */
        10,0, 35,0, 't','r','a','c','e',0,
        10,0, 23,0, 'w','a','r','n',0,0,
        10,0, 40,0, 'l','i','s','t',0,0,
        10,0, 22,0, 'c','o','p','y',0,0,
        10,0, 32,0, 'q','u','o','t','e',0,
        12,0, 33,0, 's','e','r','i','a','l',0,0,
        10,0, 36,0, 'a','p','n','d',0,0,
        10,0, 24,0, 's','i','z','e',0,0,
        12,0, 25,0, 's','u','b','s','t','r',0,0,
         8,0, 26,0, 'g','o',0,0,
        10,0, 27,0, 'g','o','b','k',0,0,
        10,0, 37,0, 'c','d','e','f',0,0,
         8,0, 28,0, 'd','e','l',0,
         8,0, 29,0, 'd','n','l',0,
         8,0,  7,0, 's','e','q',0,
         8,0, 38,0, 'k','e','y',0,
         8,0,  8,0, 's','n','e',0,
         8,0,  9,0, 'a','d','d',0,
         8,0, 10,0, 's','u','b',0,
         8,0, 11,0, 'm','p','y',0,
         8,0, 12,0, 'd','i','v',0,
         8,0, 13,0, 'e','x','p',0,
         8,0,  1,0, 'g','t',0,0,
         8,0,  2,0, 'e','q',0,0,
         8,0,  3,0, 'g','e',0,0,
         8,0,  4,0, 'l','t',0,0,
         8,0,  5,0, 'n','e',0,0,
         8,0,  6,0, 'l', 'e',0,0,
         8,0, 21,0, 'd','e','f',0,
         8,0, 20,0, 'i','f',0,0,
         0,0
};
```

```
int dummy[] { 0, 0, 0, 0, 0, 0, 0, 0 };

int getfree[] { 0, 0, 0, 0, 0, 0, 0, 0 };

char metas[] { '<', '>', '#', ':', ';', ',', '$' };

char one[] { '1', 0 };

int rescan 1;
int traceflag 0;
```

## 3. M61.c

m61.c contient le programme principal, main, les routines de diagnostic d'erreurs, diag et memfail, la routine d'initialisation des macros dummy et getfree, initdumget, et la routine d'initialisation du sous-système de gestion des macros, initdefst.

```
#

#include "m6.h"

main(argc,argv)
char **argv;
{
        extern fin;
        gf = ge = g0;
        pf = pe = p0;
        gmax =+ g0 - 10;
        pmax =+ p0 - 10;
        initdumget( );              /* to initialise dummy and getfree macro */
        initdefst( );               /* to initialise the definition table */
        if(argc>1) {
                fin = open(argv[1],0);
                control( );
                close(fin);
        }
        fin = dup(0);
        control( );
}


diag(m) {
        printf("%s0,m);
        iot( );
}       /* abort */

int iot 4;

initdefst( ) {
/*
 * to initialise the system definition table
 */
        register int i, j, k;
        for (i = 0 ; i < DSIZE ; i++) def[i] = 0;
                                    /* all def entries are set to 0 */
        i = loaddef;                /* enter m6-pre-defined macros */
        do {
                if((j = room(keyval(&(i->ldident)))) < 0) memfail("initdef");
                j->dtype  = 'd';
                j->dswitch = i->ldswitch;
                j->dlist = 0;
                if((j->dtext = alloc(1)) < 0) memfail("initdef1");
                *(j->dtext) = 0;
                k = space(&(i->ldident));
                if((j->dident = alloc(k))< 0) memfail("initdef2");
                move(&(i->ldident),j->dident,k);
                i =+ i->ldnext;
        } while (i->ldnext);
}
```

```
memfail(m) {
/*
 * diagnostic of memory lack
 */
printf("11s%s","No more available memory from system detected in ",m);
diag("");
}

initdumget( ) {
/*
 * init macro getfree and dummy
 */
        dummy->dtext = &"$0";
        getfree->dtext = &"";
}
```

## 4. M62.c

m62.c regroupe le "coeur" du système et ses principaux outils. Control est l'interpréteur M6, begincall la procédure prenant en compte un début d'appel, newarg celle qui signale un changement d'argument, endcall quant à elle marque la fin d'un appel tandis que setscan manipule les pointeurs de la pile get pour les maintenir à jour.

```
#

#include "m6.h"

control( ) {
        register int returnlg;          /* previous lg value */
        returnlg = lg -1;
        while(1) {
                get( );
                if(c==0 && lg>0 && lg > returnlg)  {
                        popget( );
                        rescan = 1;
                        if(lg == returnlg) return;
                }
                else {
                        if(!rescan) put( );
                        else if(lq>0) {
                                if(c==lquote) {
                                        lq++;
                                        put( );
                                }
                                else if(c==rquote || c==0)  {
                                        lq--;
                                        if(lq>0 || l9>0) put (rquote);
                                }
                                else put( );
                        }
                        else if(l9>0)
                                if(c==colon||c==0||c==semi)
                                        if(--l9<=0) endcall( );
                                        else put( );
                                else {
                                        if(c==sharp) l9++;
                                        else if(c==lquote) lq++;
                                        put( );
                                }
                        else {
                                if(c==sharp) begincall( );
                                else if(c==lquote) lq++;
                                else if(lp>0) {
                                        if(c==colon||c==0||c==semi) endcall( );
                                        else if(c==comma) newarg( );
                                        else put( );
                                }
                                else if(c==0) return;   /* lg=lp=lq=0 */
                                else put( );
                        }
                }
        }
}
```

```
endcall( ) {
        register char *pt, *r;
        rescan = c!=semi;
        newarg( );
        pushget( );
        pt = &pf->pa0;
        ge = move(pt,&gf->ga0,pe-pt);
        if(ge>gmax) diag("Call stack overflow from endcall");
        if(traceflag) dotrace( );
        r = finddef(0);
        setscan(r);
        popput( );
        lp--;
             if(r->dtype == 'c' ) compex( );
        else if(r->dtype == 'd' ) function(r->dswitch);
}

begincall( ) {
        lp++;
        pushput( );
        pe = pf + pend;
        pf->pan = pf->pa0 = 0;
}

newarg( ) {
        register char *pp;
        if(++pf->pan>=9) if(c==comma) l9++;
        *pe++ = 0;
        pe = (pe+1)&0177776;
        pp = &pf->pa0;
        while(pp->word!=0) pp =+ pp->word;
        pp->word = pe - pp;
        *pe++ = *pe++ = 0;
}

setscan(p)
char *p;
{
        gf->mframe = p;
        gf->mchar = p->dtext;
        gf->marg = 0;
}
```

## 5. M63.c

m63.c est le fichier qui contient les procédures d'accès aux macros findef, lookup, delete et remove ainsi que deux utilitaires, comp et move. Les quatres premières procédures sont celles qui sont spécifiées dans le chapitre deux tandis que comp compare caractère par caractère deux strings et que move déplace un certain nombre de caractères d'une adresse à une autre.

```
#

#include "m6.h"

char *
finddef(n)
int n;
{
/*
 * return adress of definition frame of macro named arg(n) or adress
 * of definition frame of empty macro
 */
        register int i;
        if((i = lookup(arg(n))) < 0) if((i = lookup("")) < 0)
                        diag("Undefined macro call; empty macro not found");
        return(i);
}

lookup(p)
char *p;
{
/*
 * search macro whose name is the same as string pointed by p
 * return -1 if not found
 */
        register int i;
        register char *q;
        q = p;
        i = def[keyval(q)];
        while(i) {
                if(comp(q,i->dident)) return(i);
                i = i->nextst;
        }
        return(-1);
}

comp(s,t)
char *s, *t;
{
        register char *x, *y;
        x = s;
        y = t;
        for(;*x++ == *y;y++)
                if(*y==0) return(1);
        return(0);
}
```

```
remove(p)
char *p;
{
/*
 * remove the definition frame at adress p
 */
        register int i, key;
        register char *q;
        q = p;
        key = keyval(q->dident);
        if(def[key] == q) def[key] = q->nextst;
        else {
                i = def[key];
                while(i->nextst != q) i = i->nextst;
                i->nextst = q->nextst;
        }
        /* subchain number key is reseted */
        if(i = q->dtext) free(i);
        if(i = q->dident) free(i);
        if(i = q->dlist) freecp(i);
        free(q);
}

char *
move(from,to,count)
char *from, *to;
{
        register char *x, *y;
        register int z;
        x = from;
        y = to;
        z = count;
        while(z--->0) *y++ = *x++;
        return(y);
}

delete(n)
int n;
{
/*
 * deletes the macro whose name is the same as argument n
 */
        register int i;
        i = lookup(arg(n));
        if (!(i < 0)) remove(i);
}
```

## 6. M64.c

Nous trouvons ici les procèdures qui manipulent les pointeurs des tables put et get quand celles-ci reçoivent ou perdent un appel de macro. Put et get sont les procèdures qui manipulent les flots de caractères argument et résultat, lisant le premier et écrivant le second. La procèdure trace est celle qui est associée à la macro pré-définie de même nom.

```
#

#include "m6.h"

pushget( ) {
        ge = (ge+1)&0177776;
        ge->word = gf-ge;
        gf = ge;
        ge =+2;
        ++lg;
}

dotrace( ) {
        char *arg( );
        register int i,j;
        printf("19d %c%s",lg,sharp,arg(0));
        for(j=9;j>0&&*arg(j)==0;j--);
        for(i=1;i<=j;i++)
                printf("%c%c%s%c",comma,lquote,arg(i),rquote);
        printf("%c0,c);
}

popget( ) {
        ge = gf;
        gf =+ gf->word;
        --lg;
        if(gf<g0) diag("Software error from popget");
}

pushput( ) {
        if(pe&1) {
                pf->prev =| 1;
                pe++;
        }
        pe->word = pf-pe;
        pf = pe;
        pe =+2;
}

popput( ) {
        pe = pf;
        pf =+ pf->word;
        if(pf->prev&1) {
                pe--;
                pf->prev =& 0177776;
        }
        if(pf<p0) diag("Software error from popput");
}
```

```
put( ) {
        if(lp>0) {
                *pe++ = c;
                if(pe>pmax) diag("Arg collection overflow from put");
        }
        else putchar(c);
}

get( ) {
        register int n;
        if(lg==0)
                c = getchar( );
        else while(1) {
                if(gf->marg!=0) {
                        if((c = gf[gf->marg++])==0) gf->marg = 0;
                        else return;
                }
                c = *(gf->mchar++);
                if(c!=dollar) return;
                n = *(gf->mchar) - '0';
                if(n<0 || n>9) return;
                ++gf->mchar;
                gf->marg = arg(n) - gf;
        }
}
```

## 7. M65.c

Define est la procédure associée à la macro pré-définie def qui entre dans le système une nouvelle macro avec son texte de définition.

Arg est une fonction dont le résultat est l'adresse de l'argument de l'appel courant qui a le numéro qui est l'argument de arg.

Function est un aiguillage qui donne le contrôle à la procédure associée à une macro pré-définie du système en se basant sur le switch de cette macro.

Lister réalise le service prédéfini annoncé au chapitre 4, à savoir l'impression de la liste de définition d'une macro pré-compilée.

```
#

#include "m6.h"

define( ) {
/*
 * introduce a new macro in the system
 */
        register int i, j;
        register char *k;
        k = arg(1);
        i = lookup(k);
        if(decbin(3) < 0) {
                if(!(i < 0)) remove(i);
                return(0);
        }
        if(i < 0) {
                if((i = room(keyval(k))) < 0) memfail("define");
                j = space(k);
                if((i->dident = alloc(j)) < 0) memfail("define1");
                move(k,i->dident,j);
        }
        else {
                if(j = i->dtext) free(j);
                if(j = i->dlist) freecp(j);
        }
        i->dtype = 'u';
        i->dswitch = decbin(3);
        i->dlist = 0;
        k = arg(2);
        j = space(k);
        if((i->dtext = alloc(j)) < 0) memfail("define2");
        move(k,i->dtext,j);
        return(i);
}

char *
arg(i) {
        register int n;
        register char *p;
        n = i;
        p = &gf->ga0;
        while(--n>=0 && p->word!=0) p =+ p->word;
        return(p->word!=0?p+2:p);
}
```

```
function(i) {
        register int n;
        n = i;
        if(n==0) return;
        else if(1 <= n && n <= 13) {
                        binop(n);
                        return;
        }
        else {
                switch(n) {
                case 20:
                        doif( );
                        return;
                case 21:
                        define( );
                        return;
                case 22:
                        copy( );
                        return;
                case 23:
                        meta( );
                        return;
                case 24:
                        size( );
                        return;
                case 25:
                        substr( );
                        return;
                case 26:
                case 27:
                        go(n);
                        return;
                case 28:
                        delete(1);
                        return;
                case 29:
                        dnl( );
                        return;
                case 32:
                        quote( );
                        return;
                case 33:
                        result(finddef(1)->dswitch);
                        return;
                case 35:
                        traceflag = comp(arg(1),one);
                        return;
                case 36:
                        append( );
                        return;
                case 37:
                        cdefine( );
                        return;
```

```
                case 38:
                        result(keyval(arg(1)));
                        return;
                case 40:
                        lister( );
                        return;
                }
        }
}

lister( ) {
/*
 * list the compile structure chain of user-pre-defined macro arg(1)
 */
        int begincall( );
        int closecall( );
        int changearg( );
        int empty( );
        int textout( );
        int argument( );
        register char *p, *q, *r;
        if((p = lookup(arg(1))) < 0) return;    /* macro not found */
        if(p->dtype != 'c') return;             /* not user-pre-defined */
        printf("24s%s0","  macro : ",p->dident);
        printf("%s0," liste des routines appelees avec les arguments ");
        printf("%s0,"================================================");
        p = p->dlist;
        while(p) {
                r = p->fct;
                     if(r == begincall) q = &"begincall";
                else if(r == closecall) q = &"closecall";
                else if(r == argument)  q = &"argument";
                else if(r == textout)   q = &"textout";
                else if(r == empty) q = &"empty";
                else if(r == changearg) q = &"changearg";
                else q = &"?????";
                printf("%s%s%d%s%d%s0,q,"(",p->argum1,",",p->argum2,")");
                p = p->nextcps;
        }
}
```

## 8. M66.c

Ici nous trouvons les diverses routines réalisant les services des macros pré-définies ainsi que la procédure copy, qui défini une nouvelle macro avec le texte d'une autre macro déjà présente dans le système.

Freecp restitue à UNIX la place occupée par la liste de définition d'une macro détruite.

Keyval est la fonction de hashing présentée au chapitre 2 tandis que room manipule les sous-listes de définition de macros pour y introduire une nouvelle structure.

Space calcule la longueur d'un string de caractères et append est spécifiée au chapitre 4.

```
#

#include "m6.h"


doif( ) {
        register int i;
        register int *p;
        char *arg( );
        i = 1;
        while(!comp(arg(i),one)) if((i =+ 2)>8) return;
        p = arg(i+1) - 2;
        ge = move(p,&gf->ga0,p->word+1);
        setscan(dummy);
}

bindec(i) {
        register int n;
        n = i;
        if(n == 0) return;
        bindec(n/10);
        *ge++ = (n%10) + '0';
}

result(i) {
        register int n;
        register char *p;
        n = i;
        setscan(dummy);
        ge = 2 + (p = &gf->ga0);
        if(n<0) {
                *ge++ = '-';
                n = -n;
        }
        if(n==0) *ge++ = '0';
        else bindec(n);
        *ge++ = 0;
        ge = (ge+1)&0177776;
        p->word = ge - p;
        *ge++ = *ge++ = 0;
}
```

```
binop(i) {
        int r;
        register int arg1, arg2, code;
        code = i;
        arg1 = decbin(1);
        arg2 = decbin(2);
        if(code < 7)    /* relationals */
                result((code & ((arg1<arg2)?4:(arg1==arg2)?2:1)) != 0);
        else if(code < 9)       /* seq=7 sne=8 */
                result((code==7)==comp(arg(1),arg(2)));
        else switch (code) {
        case 9:
                result(arg1+arg2);
                return;
        case 10:
                result(arg1-arg2);
                return;
        case 11:
                result(arg1*arg2);
                return;
        case 12:
                result(arg1/arg2);
                if(arg2==0) gf->ga0 = 0;
                return;
        case 13:        /* exp */
                r = 1;
                while(arg2-->0) r =* arg1;
                result(r);
                if(arg2<-1) gf->ga0 = 0;
        }
}

decbin(i) {
        register char *s;
        register char t;
        register int n;
        if(t = (*(s = arg(i))=='-')) s++;
        n = 0;
        while(*s>='0' && *s<='9') n = 10*n + *s++ - '0';
        return(t?-n:n);
}

dnl() {
        register char d;
        register int i;
        i = decbin(1);
        if(i < 0) return;
        do {
                d = getchar();
                if(d == 0) diag("Dnl out of standart input");
                if(d == '0') i--;
        } while(i >= 0);
}
```

```
quote( ) {
        register char *p;
        p = (finddef(1))->dtext;
        while(c = *p++) put( );
}

copy( ) {
/*
 * to define arg(2) with same text as arg(1)
 */
        register int i, j, k;
        k = arg(2);
        if((j = lookup(k)) < 0) {
                if((j = room(keyval(k))) < 0) memfail("copy");
                i = space(k);
                if((j->dident = alloc(i)) < 0) memfail("copy1");
                move(k,j->dident,i);
        }
        else {
                if(i = j->dtext) free(i);
                if(i = j->dlist) freecp(i);
        }
        i = finddef(1);
        j->dswitch = i->dswitch;
        j->dtype = i->dtype;
        k = space(i->dtext);
        if((j->dtext = alloc(k)) < 0) memfail("copy2");
        move(i->dtext,j->dtext,k);
        if((i = i->dlist) == 0) {
                j->dlist = 0;
                return;
        }
        if((j = (j->dlist = alloc(cend))) < 0) memfail("copy3");
        do {
                j->fct = i->fct;
                j->argum1 = i->argum1;
                j->argum2 = i->argum2;
                if((i = i->nextcps) != 0) {
                        if((j->nextcps = alloc(cend)) < 0) memfail("copy4");
                        j = j->nextcps;
                }
                else {
                        j->nextcps = 0;
                        break;
                }
        } while(1);
}
```

```
go(n) {
        if(comp(arg(1),one)) {
                popget( );
                if(lg > 0)
                        if(n == 27) setscan(gf->mframe);
                        else popget( );
        }
}

gogobk(n) {
/*
 * go or gobk macro call in a user-pre-defined macro
 */
        if(comp(arg(1),one)) {
                popget( );
                return(n);
        }
        return(0);
}

size( ) {
        register int i;
        register char *p;
        i = 0;
        p = arg(1);
        while(*p++ != 0) i++;
        result(i);
}

meta( ) {
        register char d, e;
        register int i;
        char *arg( );
        d = *arg(2);
        e = *arg(1);
        if(e == dollar) {       /* change dummy->dtext */
                *(dummy->dtext) = d;
        }
        if(d!=0) {
                for(i=0;i<NMETA;i++)
                        if(metas[i] == e) metas[i] = d;
        }
}
```

```
substr( ) {
        register char *s;
        char *t;
        register int arg2,arg3;
        char *arg( );
        s = arg(1);
        arg2 = decbin(2);
        arg3 = *arg(3) == 0 ? 32767 : decbin(3);
        if(arg2 < 1) {
                arg3 =+ arg2 - 1;
                arg2 = 1;
        }
        while(--arg2 > 0 && *s != 0) s++;
        setscan(dummy);
        ge = 2 + (t = &gf->ga0);
        while(arg3-- > 0) if((*ge++ = *s++) == 0) break;
        *ge++ = 0;
        ge = (ge + 1)&0177776;
        t->word = ge - t;
        *ge++ = *ge++ = 0;
        if(ge > gmax) diag("No more room for a substring");
}

append( ) {
/*
 * to append arg(2) to the definition text of arg(1)
 * if arg(1) is not found returns
 * if arg(1) is user-pre-defined, it becomes user-defined
 */
        register int i, j;
        register char *k;
        int m, n;
        if((i = lookup(arg(1))) < 0) return;
        if(i->dtype == 'c') {
                freecp(i->dlist);
                i->dlist = 0;
                i->dtype = 'u';
        }
        j = i->dtext;
        k = arg(2);
        m = space(j) - 1;
        n = space(arg(2));
        if((i->dtext = alloc(m + n)) < 0) memfail("append");
        move(j,i->dtext,m);
        move(k,i->dtext + m,n);
        free(j);
}
```

```
freecp(q)
int q;
{
        register int i;
        register int j;
        i = q;
        do {
                j = i->nextcps;
                free(i);
                i = j;
        } while(i);
}

space(p)
char *p;
{
/*
 * return number of bytes needed to store the string pointed by p
 * with his ending 0
 */
        register int i;
        register char *q;
        q = p;
        i = 0;
        while(*q++) i++;
        return(++i);
}

keyval(p)
char *p;
{
/*
 * makes a key from string p
 */
        register int key;
        register char *q;
        q = p;
        key = 0;
        signal(10,1);   /* system call to desable overflow */
        while (*q) key =+ *q++;
        key =& 0177;
        signal(10,0);   /* system call to enable overflow */
        return(key);
}
```

```
room(n)
int n;
{
/*
 * return adress of free definition frame linked to entry def[n]
 */
        register int i;
        if((i = alloc(dend)) < 0) memfail("room");
        i->nextst = def[n];
        def[n] = i;
        return(i);
}
```

## 9. M68.c

Nous avons regroupé ici les procédures spécifiées au chapitre 3 qui sont utilisée pour réaliser un appel de macro pré-compilée. Elles sont explicitées et spécifiées dans le texte.

```
#

#include "m6.h"

int bypasscall;          /* nonzero when a go or gobk macro must be
                          * taken into account while treatment of
                          * a user-pre-defined macro */
int pcps[4] { 0, 0, 0, 0 };
                         /* used when a gobk call must be treated */

compex( ) {
/*
 * routine called to treat the compile structures prepared by
 * cdefine in order to realise the user-pre-defined macro
 */
        register char *p;
        register int n;
        int (*adrfct)( );        /* pointer to a function */
        p = gf->mframe;          /* to have access to definition frame */
        if (!rescan) {           /* the text need not be rescanned */
                p = p->dtext;
                while ((c = *p++) != 0) {
                        while (c == dollar){
                                n = *p - '0';
                                if (n < 0 || n > 9) break;
                                p++;
                                argument(n,0);
                                c = *p++;
                        }
                        put( );
                }
        }
        else {                           /* text must be rescanned */
                bypasscall = 0;          /* not found a go or gobk */
                p = p->dlist;    /* find first compile structure */
                pcps->nextcps = p;
                        /* save first structure in case of gobk call */
                do {             /* treat each structure of definition list */
                        adrfct = p->fct;
                        (*adrfct)(p->argum1,p->argum2);
                        if(bypasscall == 26) break;
                        else if(bypasscall == 27) {
                                p = pcps;
                                bypasscall = 0;
                        }
                }while(p = p->nextcps);
        }
        setscan(getfree);
}
```

```
empty(p,q)
int p;
int q;
{
/*
 * routine that does nothing; called for empty user-pre-defined macro
 */
        ;
}

evalarg(p,q)
int p;
int q;
{
/*
 * routine to scan the argument number p of the current call
 */
        register char *base, *pp;
        base = arg(p);
        pushget( );
        setscan(dummy);
        ge = 2 + (pp = &gf->ga0);
        ge = move(base,ge,space(base));
        ge = (ge + 1)&0177776;
        pp->word = ge - pp;
        *ge++ = *ge++ = 0;
        control( );
}

argument(p,q)
int p;
int q;
{
/*
 * routine to output the argument number p of the current call
 */
        register char *base;
        base = arg(p);
        if (lp > 0) {
                while (*pe++ = *base++)
                        if (pe > pmax)
                                diag("Arg collection overflow from argument");
                pe--;                                   /* to leave the 0 */
        }
        else while (c = *base++) putchar(c);
}
```

```
textout(p,q)
int p;
int q;
{
/*
 * routine to output p characters of the definition text of a pre-compiled
 * macro ; first character of the string is the (q + 1) th. character
 * of the definition text
 */
        register char *base;
        register int r;
        r = p;
        base = (gf->mframe)->dtext + q;
        if (lp > 0) {
                while (r-- > 0) {
                        *pe++ = *base++;
                        if (pe > pmax)
                                diag("Arg collection overflow from textout");
                }
        }
        else while (r-- > 0) putchar(*base++);
}

closecall(p,q)
int p;
int q;
{
/*
 * semi, colon or 0 has been detected in pre-definition
 * and a special action may have to be taken in case of go or gobk call
 */
        register char *pt;
        register int n;
        rescan = p != semi;
        newarg( );
        pushget( );
        pt = &pf->pa0;
        ge = move(pt,&gf->ga0,pe-pt);
        if(ge>gmax) diag("Call stack overflow from endcall");
        if(traceflag) dotrace( );
        pt = finddef(0);
        setscan(pt);
        popput( );
        lp--;
             if(pt->dtype == 'c' ) compex( );
        else if(pt->dtype == 'd' ) {
                n = pt->dswitch;
                if(n == 26 || n == 27) bypasscall = gogobk(n);
                else function(n);
        }
        if(!bypasscall) control( );
}
```

```
changearg(p,q)
int p;
int q;
{
/*
 * comma has been detected in pre-definition
 */
        c = p;
        newarg( );
}
```

## 10. M69.c

Ce dernier fichier comprend les procèdures du sous système de prè-compilation ainsi que les outils de crèation et de manipulation des listes de dèfinition. Les descriptions et spècifications de celles-ci se trouvent dans le chapitre trois.

```
#

#include "m6.h"

char *text;             /* pointer to current character of definition text */
char *compilst;         /* pointer to the current compile structure */
int (*adrfct)( );       /* pointer to a function */
int intext;             /* 1 while compiling a string, otherwise 0 */
int text0;              /* to remenber the adress of the first character
                         * of the definition text */
int adrargst;           /* adress of last argst structure */
struct {
        int argprev;    /* adress of previous frame */
        int argvalue;   /* saved value of nbrarg */
#define argstsize 4
};
int nbrarg;             /* to remenber how many arguments have been
                         * collected so far */
int lc;                 /* to remenber how many sharp have been collected
                         * that have no colon, semi or 0 associated */
```

```
cdefine( ) {
/*
 * introduce the name of the new macro on pile def
 * same procedure as normal macros except type is 'c'
 */
        register int df;
        df = define( );
        if(df == 0) {   /* the switch is < 0 => not necessary to define */
                setscan(getfree);
                return;
        }
        df->dtype = 'c';
        initarg( );      /* initialize variables, first compile structure,
                          * first argst structure, first character */
        initlist(df);
        text = arg(2);
        text0 = text;
        intext = 0;
        lc = 0;
        nextchar( );     /* c is the current character of the definition text
                          * it is initialized by a call to nextchar( ) */
/*
 * pre-compilation of the definition text
 */
        if(c == 0) cpempty( );          /* definition text is empty or
                                         * countains only one or more
                                         * argument call(s) */
        else {                          /* definition text is not empty */
                while (c != 0) {
                        if (lq > 0) cplqtext( );
                   else if (l9 > 0) cpl9text( );
                   else {
                                if (c == sharp) cpbegincall( );
                           else if (c == lquote) lq++;
                           else if (lc > 0) {
                                        if (c == colon || c == 0 || c == semi)
                                                cpendcall( );
                                   else if (c == comma) cpnewarg( );
                                   else cpput( );
                           }
                           else cpput( );
                   }
                   nextchar( );
                }
        }
        closelist( );           /* to leave properly the definition phase */
        dispast( );             /* release first argstructure */
        setscan(getfree);
}
```

```
initlist(n)
int n;
{
/*
 * routine to link the chain of compile structures to
 * the definition text of the macro beeing compiled
 */
        if((compilst = alloc(cend)) < 0) memfail("initlist");
        compilst->fct = 0;
        compilst->argum1 = 0;
        compilst->argum2 = 0;
        compilst->nextcps = 0;
        n->dlist = compilst;
}

closelist( ) {
/*
 * routine to end the compile structures chain
 */
        int empty( );
        if (intext) closetext(text - 1);
        if(compilst->fct == 0) compilst->fct = empty;
                /* last thing done was output an empty text */
        compilst->nextcps = 0;
        compilst = 0;
}

opentext(p)
char *p;
{
/*
 * routine to start the compilation of a string of characters
 */
        int textout( );
        intext = 1;
        if (compilst->fct != 0) getcps( );
        compilst->fct = textout;
        compilst->argum2 = p - text0;
        compilst->argum1 = p;
}

closetext(p)
char *p;
{
/*
 * routine to stop the compilation of a string of characters
 */
        intext = 0;
        compilst->argum1 = p - compilst->argum1;
        if(compilst->argum1 == 0) {     /* the text is empty */
                compilst->fct = 0;
                compilst->argum2 = 0;
        }
}
```

```
nextchar( ) {
/*
 * routine to get the next character of the definition text;
 * if an argument is detected, produce a compile structure so that
 * it would be treated at execution time
 */
        int evalarg( );
        register int n, resumetext;
        resumetext = 0;
        c = *(text++);
        while (c == dollar) {
                n = *text - '0';
                if (n < 0 || n > 9) break;
                text++;
                if (intext) {
                        resumetext = 1;
                        closetext(text - 2);
                }
                if (compilst->fct != 0) getcps( );
                compilst->fct = evalarg;
                compilst->argum1 = n;
                c = *(text++);
        }
        if(resumetext) opentext(text - 1);
}

getcps( ) {
/*
 * routine to obtain a new compile structure and
 * to link this new structure with the list
 */
        register int i;
        if((i = alloc(cend)) < 0) memfail("getcps");
        compilst->nextcps = i;
        i->fct = 0;
        i->argum1 = 0;
        i->argum2 = 0;
        i->nextcps = 0;
        compilst = i;
}

initarg( ) {
/*
 * routine to initialise the argstructure chain
 */
        register int i;
        if((i = alloc(argstsize)) < 0) memfail("initarg");
        adrargst = i;
```