To: Distribution

From: Marshall Presser

Date: 09/19/79

Subject: A Multics Macro Processor

INTRODUCTION

The oral history of Multics includes some talk that there ought to be a macro processor as part of the standard product. While this facility already exists within a few installed products, e.g. editors, alm, runoff, etc., as well as one personal stand-alone general purpose macro processor (hereafter called Falksen's macro, described in both >udd>m>jaf>prog>macro.info, sys M and MTB 345 by J Falksen), there is no consensus as to what a standard macro processing facility should be. This document is a first attempt to:

(1) propose that a variant of Falksen's macro be imbedded in the PL/I compiler to satisfy the ADP development need described below,

(2) inform the Multics community of the more important issues,

(3) describe some existant macro processors.

BRIEF HISTORY

Macro processing seems to have arisen as a feature of assemblers when programmers desired a similar piece of code, often parameterized, to be executed frequently but without the overhead of a subroutine call or the tedium of reproducing the code on many punched cards. Soon clever people discovered that all sorts of wonderful features could be incorporated into macro assembly languages, including conditional macro time text replacement, iteration, recursion, local macro time variables, etc. Why limit these delights to assemoler programmers? IBM actually built them into its PL/I compiler, and the Multics PL/I %include facility is a simple, but useful macro processing tool. These days, no editor with pretensions is without something it calls macro processing ability. Similarly, many free standing macro processors exist.

some that seem to be language-dependent, and others that have the ability to generally preprocess text before it is to be subject to another translator.

WHO WANTS A PREPROCESSOR AND WHY?

There seems to be a wide variety of positions on the need for a macro processor. Some feeling exists that macro processing in a high-level language is in itself a "bad thing," and that its existence would cause an explosion of pseudo programming languages and invasion of idiosyncratic phrases into PL/I code, diminishing understandibility and maintainability. Certainly, there was a ban on the introduction of macro facilities in alm for a long while, although for a different reason.

Others merely wish for the availibility of named constants, but not in the sense of the present PL/I version of constants. This need could be met by pre-expansion of the source segment through an editor, replacing all the names by their constant values. It might be preferable to alter the language specification to meet this rather limited need.

There may be a commercial need to produce a macro processor to satisfy present or potential customer demand. In this case, suggestions are made to reproduce the facilities in the IBM preprocessor. Among the opponents of this is Dave Ward, who implemented a GCOS version and suggests that the IBM product is both inelegant and a bit messy to implement. Unless commerical considerations are rather pressing, mere compatibility is not terribly attractive.

There is a commonly voiced desire to be able to maintain a single source segment which could be conditionally compiled for development systems like ADP, DPS/E, etc, or for the standard product. Exactly which features are required is not clear, but a minimal set of useful features need contain no more than:

(1) single-level, rather than nested, macro definition ability,

(2) less sophisticated iteration and parameter passing mechanisms than found in alm, Falksen's macro, MIDAS, etc.

(3) conditional compilation depending on the existence of defined macros or their values as character strings,

(4) argument control from the command level.

The macro processing facility to meet these needs can easily be met with some revisions to Falksen's macro. Indeed, a subset of its facilities, with a few extensions, that I have already built in and used, seems to be sufficient for these needs. In

particular this includes macro definition, if-then-else structures, positional parameters referenced by number, macro libraries, tests for command line arguments, and a few other features.

IF WE WANT A USEFUL, EXTENDABLE PRODUCT IN THE NEAR FUTURE, THIS SEEMS THE BEST WAY TO GO.

ISSUES ARISING IN MACRO PROCESSING

1. Should the macro processor be free-standing or integrated into, say, the PL/I compiler?

As various macro processing facilities already exist elsewhere, for the in-house development need mentioned above, PL/I source segments seem to be the obvious candidate for "macro-ization." Furthermore, as everyone demands the consistency of line numbers, the integration of macro facilities into the PL/I compiler seems infinitely preferable to an independent pre-compilation expansion phase, although PL/I formatting programs which alter line numbers exist in this manner. Macro definition and expansion can occur in the lexing phase of the compiler, just as %include expansion now occurs. The macro processor should have a set of subroutine interfaces, somewhat like those of Falksen's macro, for these purposes. In this way minimal reworking of the compiler will be required and the facilities can be made available for other translators. It might also be useful to consider what, if any, alterations could be done to indenting programs to recognize macro expressions.

2. Should it be line-based, token-based, or character-based?

While most assembly language macro processing is based on lines or statements and much editing macro activity is based on character strings, it is reasonable to suggest that high-level language macro processing be based on tokens, in particular "identifier-like" tokens. (Should anyone really use a macro processor to redefine, say + to be -?) Furthermore, will not the expansion of either character strings imbedded in identifiers or arithmetic expressions create all kinds of nasty confusion? This tokenizing approach is also compatible with the present use of %include on the PL/I compiler. Strong objections to this idea should make themselves felt soon rather than later. There is an advantage to line-based (or statement-based) macros in that the listings can be made easier to interpret. If programmers place more than one complicated macro on a line, the relationship between line numbers in the unexpanded and expanded sources becomes problematic. The use of indent and format_pl1 could make for greater readability. See section 5 below for further discussion.

3. Should the triggering of macro expansion or definition require a special macro character?

Although it may seem a bit cumbersome to include a % to invoke the macro processor, certain benefits accrue. The macro processor can be made more efficient if the trigger character is used, as it would not require a lookup of every token to determine if macro expansion were needed. Furthermore, it eliminates some ambiguity and certainly makes the intention of the statement more clear. The trigger character is therefore useful in delineating non-standard PL/I constructs. On the other hand, consider the statement:

```
fred = control_function(norman,harold);
```

Programmers may wish to be able to write this statement and have it conditionally processed, so that in some circumstances, the right-hand-side is to be viewed as a function call or array reference, while in others, as a macro to be expanded. The requirement of a trigger character would provide some difficulty. If programmers mostly agree that the invocation of macros ought not to require a special trigger symbol, it then remains a problem of distinguishing between the macro call and the literal character string of the same name. This can be done by (1) requiring all macros to be triggered by the special symbol, (2) placing some literal brackets around tokens that must not be expanded, or (3) a macro pseudo-op which temporarily suspends expansion of certain macros. The last alternative would look the messiest, and the second would probably be the most convenient if sensible users tried to avoid using ambiguous names.

4. Should there be immediate expansion of macro definitions or should these be done at time of invocation?

The problem with immediate expansion is that the order of definition becomes critical. For example, the following sequence (where the define pseudo-op causes its first argument to be replaced by its second when encountered and undefine removes its argument from the list of expandible macros):

```
define(MEP,Marshall_Presser)
define(Ivan_the_Terrible,MEP)
undefine(MEP)
```

will expand Ivan_the_Terrible to MEP if macro definitions are expanded at call time and to Marshall_Presser if expansion is at define time. Suggestions welcomed as to which method is preferable, as this is not merely a question of implementation, but rather a fundamental design choice. The use of a trigger character greatly simplifies reading of the source segment, makes the programmers intentions clear, resolves questions of order dependance in macro definition, and may prevent ambiguities with

non-macro constructs in the source segment. Closely allied with this problem is that of the rescanning of macro expansions to see if they contain more macros to be expanded. If this is done then the sequence

```
define(fred,george)
define(george,fred)
```

will lead to chaos. If rescanning is either forbidden or required to be made explicit, then chances of infinite recursion are not likely. Alternatively, recursion and mutual recursion are very useful techniques and should be permitted, although they may prove more expensive than iteration when both are possible.

5. How should macro expansions appear in compiler listings?

In the case of named constants , for example, it may be useful to be able to see both the name and the numerical value to which it is defined at debug-time. Similarly the treatment of line numbers in compiler error listings can lead to confusion. This is a non-trivial question. Most macro assemblers provide the default that expansions do not appear in the listings. This strategy is not useful for debugging purposes in general. When using the PDP-11 BIOMAC package, a set of structured programming macros, it is not useful to see the expansion of these macros, but of user-defined macros. Shall we have an option that indicates that the listing is not to include macros found in system macro libraries? What are the implications of this strategy for the use of probe? Consider the case of the declaration macro, which can be viewed in some circumstances as a generalization of the "like" attribute. Most users will not want to see these expanded. Hence the need for a listing off/on toggle seems useful. Alternatively, each macro may have imbedded within its definition a pseudo-op indicating whether it is to be expanded, or possibly a control argument can be used. The real issue, however, is the selective listing of the macro expansion.

6. What Multics standards should be placed on the use of macro facilities?

7. Should arguments to macros be by position, keyword, both or neither?

What actions should occur if the number of arguments in the call is not the same as the number in the definition? MIDAS, for example, provides a wealth of parameter passing mechanisms, as well as distinguishing between unspecified arguments and explicitly null-specified arguments. While this may be a very useful function, it is implemented by a large set of special symbols to indicate the nature of the parameters being passed. Those not quite familiar with the system will find the unexpanded source very difficult to understand. Even with experience, the

MIDAS approach may very well be overkill. Experienced users might wish to comment on this. Keyword parameters make for much more readable code, although they change the format of a macro call to something different than that of a standard function/subroutine call. The similarity of appearance of macro calls to function calls may well prove to be a very handy feature. While reference to parameters by name rather than by position number, e.g. &1, &2, etc., produces more readable code, many macroprocessors require the latter kind of reference for ease of implementation.

8. What kind of macro time variables, if any, are required?

Falksen's macro provides scalars, arrays, lists, and queues, but only of character strings. These may be evaluated and manipulated arithmetically. Given this ability, is the creation of an arithmetic data type necessary, or merely useful? Falksen's macro also provides local, internal, and external variables as well as such macro constructs as the number of arguments passed to the call. What, if any, other facilities are advantageous?

9. What is a reasonable set of pseudo-operations and built-in functions?

Certain standards include the ability to undefine macros, to alter the special macro symbol, to conditionally compile depending on whether a macro has been defined or if the value of two strings are equal, to divert the input stream, to convert character strings to numbers and perform arithemtic on them or use them in comparisons, etc. Closely related are the string handling built-in fuctions, such as index, substr, verify, etc. While it would be desirable to have as much power as possible, what remains a useful working set?

10. What flow control mechanisms are most desired?

If-then-else and do-while loops seem to be common and useful tools, as well as the ability to set labels in macro definitions. This last feature is missing in Falksen's macro and might be found to be useful. The macro time label feature is distinct from the generation of labels. The alm ability to generate unique labels and refer to them within a macro may be a more useful feature of an assembly language macro processor than a high-level language macro processor.

11. What form of error recovery seems most desirable?

Simply having the macro processor halt upon the discovery of a simple syntax error seems a bit severe. A classification of error by severity level would appear to be more reasonable and compatible with PL/I syntax errors. However, macro processing

involves both "define-time" and "invoke-time" features, so that an illegal macro definition need not suspend processing, but a macro call to an unknown macro or use of an undefined variable can result in either only minor errors or complete non-comprehension, depending upon circumstances. A good approach, it seems to me, is that of the PL/I compiler which will continue processing as long as possible. This now occurs with faulty %include's, although it frequently leads to an avalanche of errors and associated messages.

12. What are the performance considerations?

Clearly the macro processor should not drastically increase compile time, especially for those who make no use of the facilities. If the macro trigger character is used, as for %include, %skip, and %page, the inclusion of macro facilities can be done so that only those requiring them need pay the price. Those who make moderate use of these facilities would not want to see an increase in compiling time of more than, say 10 to 15 percent. On the other hand, those who wish to use the macro facility to substantially extend PL/I in private directions might be encouraged to use a stand-alone macro pre-processor.

SOME MACRO PROCESSORS WORTH EXAMINING

What follows is a brief summary of the facilities of commonly used macro processors with a comment on their appropriateness, as I see it. For those who wish further information, a brief table appears in the appendix and a reference, where possible, is given to more comprehensive documentation.

1. alm (MPM-SHG):

Alm allows nested macro definition and invocation of macros within definitions. The imbedded macro need not be defined at the time of outer macro definition, but must be at the time of the invocation of the outer macro. This is characteristic of many macro processors. Definitions are triggered by the keyword "macro" but invocation requires no special character. There are facilities for unique label generation and reference, iteration by list, selection group control, a function yielding the number of arguments passed to the macro, and pseudo-ops for conditional assembly depending on whether an argument is an integer, if two arguments are equal, etc. Alm also allows conditional assembly on arguments passed at command level. There are no macro time variables. While there seem to be a wealth of good ideas in alm macro facilities, their flavour is not that of a high-level language and the alm macro structure may not be appropriate for the in-house use mentioned above.

2. Falksen's macro (>udd>m>jaf>prog>macro.info,sys M or MTB 345):

Nested macro definition has just been added. The inclusion of some other features, such as the ability to query command-line arguments and tests to determine the existence of the definition of macros with given names would seem to make this an attractive choice. First indications are that the integration into the lexing phase of the PL/I compiler should not prove too dificult. Extensive use is made of the trigger symbol and the white-space conventions are not PL/I-like and may require some restructuring, but it is easy to use , allows nested invocations, is recursive, provides useful control structures, provides easy readability, and has built-in debugging features. It does, however, prohibit the use of certain reserved words as macro variables.

3. IBM PL/I preprocessor (GC20-0009-4 or SC20-1609-1):

This preprocessor requires a separate pass before compilation in which preprocessor statements are converted before the compiler, per se, attacks the program. These can be independant phases, and I am not certain of the effects on listings. There exist character and fixed decimal variables which require explicit declaration, as well as preprocessor functions, which may have their own local variables. Flow of control is achieved by %DO groups and %IF...&THEN...%ELSE as well as %GOTO's. This implies preprocessor labels. Arguments are by position as well as keyword. Such preprocessor functions look very much like PL/I function procedures. Output of this product is not pretty. There are no explict provisions made for command line argument handling. Nested invocations are permitted, but I am not certain about nested definitions. Built-in preprocessor functions include substr, length, index, a unique decimal number generator, and a test indicating if parameters have explicitly been set on invocation. A reasonable restriction requires that a procedure definition can not span included files.

4. UNIX M3,M4,and M6 (UNIX Programmers Manual, Bell Labs):

There are provisions to define and undefine macros, but the rescanning of definitions at define time often yields unintended results. Useful features include the diversion of input streams, include files, change of quote symbol, etc. Increment, evaluate, length, substr, and index are the primary built-ins. Nested definition is impossible but nested invocation is permitted as well as invocation imbedded in definition. No special trigger symbol is used for invocation, however definition requires a trigger (in some versions). There is both a free standing form as well as facilities built into the C compiler. Curiously, the compiler requires that the first character to be the trigger character if the macro processor is to be turned on. The macro processor is easy to use for simple tasks and command line argument handling can easily be implemented, but the lack of a

trigger symbol is often found confusing and as the C compiler does not produce a listing, the use of a symbolic debugger virtually forces a stand-alone pass to be made. The automatic rescanning of all macro output for further macro invocation or definition may create problems for the sharing of macro libraries. There is little in the way of flow control and sophisticated use of the system is difficult.

5. macro facilites in Multics editors (emacs, teco, qedx etc.):

These need no comment here.

6. IBM 360/370 Macro Assemblers (GC28-6514 and GC33-4010):

Versions upto and including level F provided keyword arguments default arguments, list arguments, and label data types, but not nested macro definition. Unlike some macro processors, user defined names for parameters are specified, so that the tedious use of &1, &2, etc. for positional parametrs is avoided. There are both global and local variables of three types: binary, character, and arithmetic. Pseudo-ops for type of argument, length of argument, number of arguments in a list, etc, are available. There are recursive macro calls. However, macro definitions must be either in a macro library or at the beginning of the source and the orientation is very much line-based. Control flow is mostly achieved by if's and goto's. Later versions may have more powerful features with which I am not familiar. While some of the features, such as keyword parameters with defaults, are quite useful, the lack of structured data types and flow control suggest against this general style of macro processor.

7. MIDAS:

This is a powerful PDP-10 assembler developed at the MIT AI Lab. Unlike other macro processors, parameter passing is very well specified; there are 6 kinds of argument syntax as well as bind classes to determine what happens to parameters evaluating to the null string. Nested definition and invocation are available as well as various loop controls. Conditional assembly predicated on the value of arguments or definition of macros is also supported. There are no macro time variables. While a number of WOW features are present, it is unclear that many of them are really necessary. I have a copy of the documentation for those interested.

8. The UNIVAC Proposal:

There is a document circulating which proposes a rather extensive macro language of syntax similar to that of PL/I. It appears to be enormously powerful and may bear examination if such power is considered desirable. The proposal contains a good introduction

to macro processing as well as a reference guide to the proposed macro language. As described, it would exist in a preprocessor version and an implementation would not be trivial. An in-house version was once considered by G. Chang, but I can not find anything but some documentation describing it. I am the temporary guardian of a copy of the UNIVAC proprosal and those interested can borrow it.

WHAT NEXT?

There can be no clear choice of which macro processing activities to pursue unless there is a well-defined goal and that requires an assessment as to who will be the audience for this macro processing activity. Hence, some response is required before any serious macro processing design can begin.

Please send comments to:

MPresser.Multics,

or

Marshall Presser
Honeywell Information Systems
575 Tech Square
Cambridge, Mass. 02139

Or call:

(617) 492-9320
HVN 261-9320

In addition to comments sent through mail or messages, stop in for a chat sometime or recommend other macro processors or facilities thought to be useful. Sometime in the next few weeks those interested can gather for a design preview (sic) where a consensus can be forged.

APPENDIX

alm

1. Variables: None.
2. Parameters: Positional and numbered, but may consist of lists.
3. Pseudo-ops and built-ins: Unique character string generation and reference, argument count, length of an arguments (in chars), number of elements in an iteration set, etc.
4. Triggering: Definitions by keyword "macro" and terminated by "&end." Invocations and pseudo-ops require no special trigger. Control functions begin with &.
5. Flow control: Iteration by lists, either of the arguments, an argument which is itself a list, by a constructed list, or by selection of list elements. Conditional execution based upon the comparison of two strings, if a string represents an integer, or if a string has been passed as a control argument.
6. Command line interface: Strings can be set as command arguments and conditional processing can occur on these strings.
7. Debugging/listing: No debugging tools per se. Expansions placed in listing without line numbers. A stackable on/off toggle for listing.
8. Rescanning: None.
9. Nesting: Nested definition, invocation and imbedded invocation within definition.

Falksen's macro

1. Variables: character string variables of three scopes, like external static, internal static, and automatic. Scalars, arrays, lifo, fifo, and lists, i.e. sets.
2. Parameters: Positional and numbered. Can be lists, but list handling must be explicitly handled by programmer.
3. Psuedo ops and built-ins: Protected strings, parameter count, active functions, substr, length, quoting, unquoting, white space control, and library reference.
4. Triggering: All macro constructs preceeded by &. Some reserved words as well.
5. Flow control: If-then-else-fi and do-while-od. Return statement for premature exit. No labels or goto's.
6. Command line interface: Ability to send arguments to a macro through the command line. Also a subroutine interface in the usual Multics fashion.
7. Debugging/listing: Internal debugging aid, though output is a bit obscure. All macro phrases replaceed by expansions, no listing per se.
8. Rescanning: Not unless explicitly requested by a pseudo-op.
9. Nesting: Recently installed multi-level nested definition. Invocations permitted within definitions and invocations.

IBM PL/I preprocessor

1. Variables: Global and local of type fixed decimal and char(varying).
2. Parameters: Positional and specified by name. Of type fixed decimal or char.
3. Pseudo-ops and built-ins: Deactivation and activation of variables, substr, counter function, parmset(to indicate if a parameter has been set).
4. Triggering: Use of % as a trigger for virtually all preprocessor statements, but variable and procedure calls do not require a special character.
5. Flow control: DO groups and GOTO's (preprocessor labels begin with %). IF-THEN-ELSE control as well.
6. Command line interface: None(?)
7. Debugging/listing: No expliscit debugging aids. No listing controls on the preprocessor, as the preprocessor phase produces source for compiler phase. Preprocessor can be used without compiler.
8. Rescanning: All output of preprocessor rescanned unless explicitly prevented. %deactivate used to allow a variable to be used literally.
9. Nesting: No nested definition, but invocations permitted within invocations and definitions.

UNIX

1. Variables: None.
2. Parameters: Positional and numbered.
3. Pseudo-ops and built-ins: Change of quote symbol, undefine (a macro), conditional replacement predicated upon equality of strings or existence of macro with a prescribed name, and diversion of input stream. Increment, index, substr, and length.
4. Triggering: Varies with version, but usually no trigger character for invocation, but pseudo-ops may require one. Dollar sign used as the parameter number trigger.
5. Flow control: Primitive, see 3. above.
6. Command line interface: Nothing explicit, but conditional compilations produced by altering the search rules (for include files).
7. Debugging/listing: No debugging aids. In stand alone form, everything expanded and listed (by printing the output file). C compiler provides no listing, so often error message become obscure.
8. Rescanning: Everything rescanned unless explicitly prevented. Definitions rescanned at define time.
9. Nesting: No nested definition, but invocations with definitions and nested invocation permitted.

IBM 360/370 Assembler

1. Variables: Both global and local in scope, of three types, arithmetic, binary, and character.
2. Parameters: Both keyword and positional, in any order.

defaults for keyword parameters. All parameters named. Arguments can be lists and subscript notation to access list elements. Use of &syslist pseudo-op to access entire set of positional arguments as a list.
3. Pseudo-ops and built-ins: Generation of consecutive 4 digit numbers, substr, and functions to determine data type, length (in bytes) , length (in characters) or arguments and variables, etc.
4. Triggering: Definitions triggered by keyword "MACRO". Invocation requires no trigger, but some pseudo-ops do.
5. Flow control: Mostly by means of if and goto, but a &ACTR pseudo-op allows FORTRAN like do loops (increment down to zero from an initally user set number).
6. Command line interface: A read only pseudo var &SYSPARM results in a string set in a JCL statement. Can be used in conditional statements.
7. Debugging/listing: Debugging by explicit code only. Listing control to allow allow/prevent listing of defintions as well as of expoansions.
8. Rescanning: None.
9. Nesting: No nested definition. Invocations allowed within definitions, but not within invocations.

MIDAS

1. Variables: None, but use can be made of assembly time variables.
2. Parameters: Keyword and positional, specified by name. Defaults for keywords. Special handling for unspecified and nullspecified parameters available. Wealth of argument types, "wholeline", "balanced"!", "evaluated", etc.
3. Pseudo-ops and built-ins: Lots of assembler pseudo-ops and conditional assembly pseudo-ops.
4. Triggering: Definition triggered by keyword. Invocation require only name. Pseudo-ops often begin with a ".".
5. Flow control: Mostly by means of REPEAT and IRP loops. Ability to break from a loop as well as a goto.
6. Command line interface: .ITYMAC pseudo-op defines a nameless macro which reads arguments from the teletype.
7. Debugging/listing: No explicit debugging aids. Document unclear on listing.
8. Rescanning: Not done.
9. Nesting: Full nesting facilities.

UNIVAC

1. Variables: Of type decimal fixed and char (varying). Arrays of both types. Associatively addressed arrays.
2. Parameters: Positional, but reference in macro definitions is not like that of procedures. A macro picture (template) is constructed, e.g.

```
     <'SUB' EXPRESSION 'FROM' REFERENCE ';'>
```

will match string like SUB FRED + SAM **2 FROM HAROLD (23);. In

further macro constructs in the macro body FRED + SAM **2 will be substituted for EXPRESSION and HAROLD (23) for REFERENCE.
3. Pseudo-ops and built-ins: Language features used in template matching including : CHARACTER, IDENTIFIER, EXPRESSION, etc. Also index, length, substr, etc. Meta function to alter and extend the macro language.
4. Triggering: Macro definition head the text to be expanded. They trigger by keyword. No trigger character. Macro of two kinds: trigger and syntax. Former are triggered by match of of a macro picture in the expandable text. Latter are only recognized within other macros, never from the text. Limitation of trigger macros is that they must begin with a literal.
5. Flow control: Within trigger macros there are DO loops (var = exp to exp by exp), goto's and if-then-else as well as procedure calls. Procedures but not macros, have PL/I-like syntax.
6. Command line interface: None described.
7. Debugging/listing: Check and warn statements and statement prefixes. Former used dynamically to print debugging info on possible change of value of variable, execution of a statement, macro or procedure. Latter used whenever macro is invoked, partially, but incompletely matched. Former meant only at debug time, latter as error messages for improper usage and are meant to be permanent. As this is meant as a preprocessor, no explict listing control, rather printing of output file.
8. Rescanning: By default, during the processing of a trigger macro there is no rescanning. By default, after the processing of the macro, the returned string is rescanned. Both defaults can be overridden by the RETRY and PROTECT options respectively.
9. Nesting: Apparently not in definitions. Nested invocation possible, but invocation of trigger macros not permitted in the definition of other trigger macros.