[illegible] IPC

To: Distribution

From: J. Falksen

Date: August 15, 1977

Subject: GENERALIZED MACRO PROCESSOR

The Generalized Macro Processor expands a macro source. A macro source is a mixture of literal data and macro constructs. An expanded macro source contains the literal data as-is, and the macro contructs replaced by their corresponding strings, if any.

A macro definition begins with a definition header, ends with a definition trailer, and has a body which is a mixture of literal data and macro constructs. A macro definition may be part of a macro source, or may be in a segment by itself.

The Generalized Macro Processor includes these features:

- Variables
- Arrays
- Three storage classes of arrays/variables.
- Assignment
- Iteration
- Conditional execution
- Macro calling
- Active function calling

The language has been made very context-sensitive in order to allow, as much as possible, literal data to be entered as it is to be generated. All macro constructs begin with "&".

This language is used internally by RGL for generating PL/I source. Since it will be installed as part of RGL anyway, it might as well be documented and made available to anyone who can utilize it.

Comments may be mailed to:

James Falksen
Honeywell Information Systems Inc.
5115 North 27th Avenue, MS K-28
Phoenix, AZ 85016

Messages or mail may be sent on System M to:

Falksenj.Multics

## 1. Macro Definition

```
&macro xxx<NL>...&mend<NL>
```

A macro definition for macro "xxx" looks like this. The macro name, "xxx", may be up to 26 characters long. The first character must be alphabetic, the rest may be alphanumeric (to include "_"). <NL> is a required new-line.

### Macro Definition Location

Macros can be found in three locations.
1) Imbedded within a macro source.
2) In a loose segment, named xxx.macro.
3) In an archive, with a component name xxx.macro.

Imbedded macros are from the &macro thru the &mend<NL>. Otherwise, "&macro..."must be the first characters of the segment or component. "&mend<NL>" must be the last characters.

```
          &macro ck
                    if (code ^= 0)
                    then do;
                         call com_err_(code,"&name",&1);
                         return;
                    end;
          &mend
```

The result of calling this macro will be all characters after the header <NL> up to the &mend, with the constructs "&name" and "&1" replaced by their values.

## DATA DECLARATION

Macro data may be in three classes. LOCAL- which last only as long as the current macro call. INTERNAL- which are available in the macro anytime it is called within the same expansion. EXTERNAL- which are available to any macro called within the same expansion. Search order when looking for a data name is: local, internal, external.

All of the data types described below can be in any of these three classes. The examples will show only "&loc" forms. In any of them, "&int" or "&ext" may be substituted for the "&loc".

Data names may be up to 16 characters long. The first character must be alphabetic, the rest may be alphanumeric (to include "_").

There is no conflict between data names and macro names because of the form of reference. "&xxx()" means call macro "xxx". "&xxx{}" means reference array "xxx". "&xxx " means reference scalar "xxx". (Scalar names and array names do conflict, however.)

### 2. Macro Scalar Declaration

```
   &loc xxx&;
   &loc xxx=yyy&;
```

Macro scalars may be declared with initial values. When the declaration is encountered and the variable does not already exist, it is created and assigned the value "yyy". If it does exist, then the declaration is ignored. If "yyy" has the form "&(expr)" then expr is evaluated and used. Otherwise "yyy" is a string. In either case, macro constructs may be used to make up "yyy".

```
          &ext name=rgl_dump_&;
          &int cur_rep=&1.list&;
          &loc i&;
```

### 3. Macro Array Declaration

```
   &loc xxx{expr1:expr2}&;
   &loc xxx{expr1:expr2}=yyy&;
```

Macro arrays are just like macro scalars except that dimensionality is added. "expr1" specifies the lower bound. "expr2" specifies the upper bound. If the array already exists but has different dimensionality, it is an error. If an initial value is specified, it is assigned to each element of the array being created.

```
          &ext char{0:127}=0&;
```

4. Macro Array Varying Declaration

```
   &loc xxx{expr1:expr2}var&;
```

Arrays of varying extent are available. "expr1" specifies the minimum extent available. "expr2" specifies the maximum extent available; The actual extent available is dependant on the highest and lowest elements which were assigned to it.

```
          &int graphic{-&1:&1}var&;
```

5. Macro List Declaration

```
   &loc xxx{expr}list&;
```

A list is a set of unique elements. "expr" specifies the maximum number of elements the list is to hold. A list is assigned like a scalar, but referenced like an array. When an assignment is made to a list, the list is searched to see if the new element is there. It is added only if it is not. See EXAMPLES for ways to utilize this data type.

```
          &ext et_{50}list&;
```

6. Macro Stack Declaration

```
   &loc xxx{expr}fifo&;
   &loc xxx{expr}lifo&;
```

"expr" specifies the maximum number of elements the stack is to hold. A stack is assigned like a scalar, but it can be referenced as either a scalar or an array. An assignment to a stack causes a new element to be added to it. A scalar reference to a fifo stack causes the oldest element to be used and then deleted. A scalar reference to a lifo stack causes the newest element to be used and then deleted. An array reference to a stack causes the data to be used but not deleted. The subscript "0" refers to the top-of-stack element. Subscript "-1" refers to the next-to-top one, etc.

EXECUTABLE CONSTRUCTS

7. Assignment

```
&let xxx=yyy&;
&let xxx{expr}=yyy&;
&let xxx{expr1:expr2}=yyy&;
```

A value may be assigned to a variable, array element, or range of array elements. If a range is specified, the "yyy" is assigned to each element in the range. If "yyy" has the form "&(expr)" then "expr" is evaluated and used. Otherwise "yyy" is a string.

```
          &let var1=5280&;
          &let var2=&(&var1+1)&;
          &let var3=&var2+1&;
```

Note that the last one is not an arithmetic assignment. After doing these, var1 will contain "5280", var2 will contain "5281", and var3 will contain "5281+1".

8. Macro Call

```
&xxx(string1,string2, ... )
```

Macros can be called from within macros. "xxx" is the name of the macro. There can be NO white-space between the macro name and the "(". "string1" is parameter 1 of the called macro, etc. Leading white-space in each string is discarded. If a macro has no arguments, then the form will be "&xxx()". "," separates one parameter from the next. However "(" and ")" may be used to pass a list of things as one argument. "stringn" may contain macro constructs, but all ",", "(" or ")" resulting from a construct will be literal.

Up to 100 arguments may be passed to a macro. Each argument is limited to 500 characters.

```
          &xxx(abc,def)
calls macro "xxx" with parameters: "abc", "def".
          &xxx(&"abc,def&")
calls macro "xxx" with parameter "abc,def".
          &xxx((abc,def))
calls macro "xxx" with parameter "(abc,def)".
          &xxx(&2,&yyy())
calls with parameters: "parameter2", macro-yyy-expansion.
```

```
                  &let x=&strip(&name,.pl1)&;
```
This assigns the result of the macro "strip" to variable "x";

## 9. Iteration

```
   &do stuff1 &while yyy &; stuff2 &od
```

Macros may use iteration. Either "stuff1" or "stuff2" may be null; so either leading, trailing, or imbedded decision may be utilized. "stuff1" is executed. Then the logical value of "yyy" is determined. If it is false then "stuff2" is skipped and processing continues following the "&od". If it is true, then "stuff2" is executed, followed by "stuff1", and then another test.

If "yyy" is of the form "&(expr)" then "expr" is evaluated. If the result is 0, it is false; otherwise it is true. Or "yyy" can be of the form "stringRELstring". REL can be "=", "^=", "<","<=", ">", ">="; Or "yyy" can be of the form "string". Then "0", "F", "FALSE", or "NO" (ignoring case) is false; anything else is true.

(&* gives number of parameters passed, &{...} is a non-constant reference to a parameter.)

```
          &let vv=&*&;
          &do
             (&{&vv})&+
          &let vv=&(&vv-1)&;
          &while &(&vv>0)&;
          ,&od;
```
If 3 parameters were passed this will give
```
          (parameter3),(parameter2),(parameter1);
```

## 10. Conditional Processing

```
   &if yyy &then stuff1 &fi
   &if yyy &then stuff1 &else stuff2 &fi
```

Processing can be altered by testing for existance of conditions. The logical value of "yyy" is determined. If it is true, then "stuff1" is executed and "stuff2" is skipped. Otherwise, "stuff1" is skipped and "stuff2" is executed.

If "yyy" is of the form "&(expr)" then "expr" is evaluated. If the result is 0, it is false; otherwise it is true. Or "yyy" can be of the form "stringRELstring". REL can be "=", "^=", "<","<=", ">", ">=". Or "yyy" can be of the form "string". Then "0", "F", "FALSE", or "NO" (ignoring case) is false; anything else is true.

11. Comment

```
&comment ... &;
```

This allows you to place a comment in a macro. Nothing between the &comment and the &; is looked at.

12. Return

```
&return
```

This construct causes an immediate halt of processing of the current macro.

```
      &if &(&*=0)
      &then &error 2,No arguments, call ignored.&;  &return
      &fi
```

13. Arithmetic Expressions

Arithmetic expressions can occur in various contexts. A usual one is in the form "&(expr)". In evaluating an expression, it is first expanded (look for all &contructs) and then it is scanned and arithmetically executed. The only valid operands are fixed point numbers. The calculations are carried internally as decimal(59,9) and should suffice for most applications. The normal arithmetic operators are supported: "+", "-", "/" and "*". Parentheses, "(" and ")" may be used for grouping. Relational operators are also available. These have lower precidence than "+". The operators available are "=", "^=", ">", "<", "<=" and ">=". AND and OR are accomplished via "*" and "+".

```
          &if &((&2=0)+(&2=2)) &then ...&fi
```

In this case if parameter 2 is not equal to either "0" or "2", then the sum will equal zero which means false.

```
          &if &((&2=0) * (&3 <= 10)) &then ... &fi
```

In this case if either parameter 2 is not zero or parameter 3 is greater than 10 the product will be zero which means false.

14. Error Reporting

```
&error expr,string&;
```

A macro can discover that improper conditions exist. This construct allows the fact to be reported. "expr" is evaluated. The result must be in the range 0-4.

These values will give different forms of messages:

```
 0)       NOTE: Macro "xxx", line n.
                    string
 1)       WARNING Macro "xxx", line n.
                    string
 2)       ERROR SEVERITY 2 Macro "xxx", line n.
                    string
 3)       ERROR SEVERITY 3 Macro "xxx", line n.
                    string
 4)       ERROR SEVERITY 4 Macro "xxx", line n.
                    string
```

Severity 3 will stop the output segment from being created; however processing will continue. Severity 4 will cause immediate termination of the macro expansion.

```
          &error 2,Second parameter missing, "13" assumed&;
          &error 4,Table name not supplied.&;
```

## VALUE CONSTRUCTS

### 15. Simple Parameter Reference

```
   &n
   &nn
```

A macro may be called with parameters. In the macro when you want to reference the second parameter, you say "&2" or "&02". The number after the & is either one or two digits. Note that if the text character following is a digit and the parameter being referenced is less than 10, then the leading zero is necessary. Reference to a parameter which was not supplied results in a null string being used.

```
          Listing of &3.&1.
```

gives you

```
          Listing of parameter3.parameter1.
```

### 16. Multiple Parameter Reference

```
   &{expr}
   &{expr1:expr2}
   &{expr1:expr2,string}
```

Sometimes it is necessary to reference a parameter via a variable. Or you want to reference a series of the parameters. The first form gives you a parameter, the number being specified by "expr". The second form gives you a string which is made up of parameters "expr1" thru "expr2" connected by one space. The third form gives you a string which is made up of parameters "expr1" thru "expr2" connected by "string".

```
          &{2:4, , }
```

will give you

```
          parameter2 , parameter3 , parameter4
```

Any occurance of "}" in "string" must be protected. "string" is limited to 150 characters. Any macro constructs may be used in specifying "string".

17. Protected String

```
  &" ... &"
```

This construct protects a string from the macro processor. The &" is removed from each end and the delimited string is not scanned for any other macro constructs.

18. Parameter Count

```
   &*
```

This construct is replaced by the number of parameters passed to the macro.

```
          &{1:&*,}
```

gives you a string which is all parameters connected by a null string.

19. Literal &

```
   &&
```

This construct gives you an & in the output.

20. Macro Scalar Reference

```
   &xxx
```

This form makes a scalar reference to a data item. "xxx" is the name of the variable. The character immediately following "xxx" may not be "(" or "{".

```
                          &abc(xyz)
```

calls macro abc with parameter "xyz".

```
                         &abc&.(xyz)
```

outputs the contents of variable "abc" concatenated with the string "(xyz)".

This form of reference may be made to scalar data and stack data. Referencing stack data causes the referenced value to be removed from the stack. See Macro Stack Declaration for details.

21. Macro Array/List/Stack Reference

```
&xxx{expr}
&xxx{expr1:expr2}
&xxx{}
&xxx{expr1:expr2,string}
&xxx{,string}
```

This form makes array reference to a data item. "xxx" is the name of the array. There can be NO white-space between "xxx" and "{". This gives all the facility described under Multiple Parameter Reference, except that an array variable is referenced instead of the macro parameters. Also, if "expr1:expr2" is null, then reference is made to all elements in the data item.

This form of reference may be made to arrays, lists, and stacks. However, only a single element reference may be made to a stack. See Macro Stack Declaration for further details.

22. Active Functions

```
&[ string ]
```

This construct allows you to make use of active functions within macros. Macro consructs may be used to make up "string". "string" is limited to 500 characters. The value returned by the active function is limited to 500 characters. If active functions are being nested, only the outermost [ is preceeded by &.

```
          Today is &[date], and it is &[time].
          Report is due &[date "&quote &1&;"].
          File it in &[directory [wd]]>status.
```

23. Substr Function

```
&substr string,expr1 &;
&substr string,expr1,expr2 &;
&substr string,expr1:expr2 &;
```

Only part of a string need be used. Macro constructs may be used to produce "string". "string" is limited to 16384 characters. The first form gives you the part of "string" from character "expr1" to the end. If "expr1" is negative, then it is the character number from the end of the string. "expr1" cannot reference outside "string".

The second form gives you the part of "string" from character "expr1" for a total length of "expr2". If the number of characters left in "string" is less than "expr2", then the result is padded with spaces. If "expr2" is negative, the padding is to the left, otherwise it is to the right. If no padding is needed, then the sign of "expr2" is immaterial.

The third form gives you the part of "string" from character "expr1" thru "expr2". Both of these values must be within "string". In this case, if "expr2" is negative, it means to count from the end of "string".

```
If you do &let xxx=abcdefg&;
          then this:              yields this:
     &substr &xxx,2,3&;             bcd
     &substr &xxx,3&;               cdefg
     &substr &xxx,-3&;              efg
     &substr &xxx,3,8&;             dcefgbbb
     &substr &xxx,-3,8&;            efgbbbbb
     &substr &xxx,-3,-8&;           bbbbbefg
     &substr &xxx,3,5&;             cdefg
     &substr &xxx,3:5&;             cde
```

24. Length Function

```
   &length string&;
```

You can get the number of characters in string.
          &length &1&;
will tell you how long parameter 1 is.

25. Usage Function

```
   &usage string&;
```

This function allows you to document what macros went into the generation of a macro output. "string" is an ioa_ control string which describes the format of the output. It is given to ioa_ with 3 parameters: dname,ename,macname dname is the directory name of the segment which contained the macro. ename is the entry name of the segment which contained the macro. macname is the macro name (less .macro) of the macro. All white-space needed in the result must be specified in "string". "string" is used once for each macro used.
          &usage /* ^a>^a -- ^a */^/&;
This is one way you could display macro usage at the end of a PL/I source.

26. Quote Processing

```
   &quote string&;
   &unquote string&;
```

The macro processor is supposed to language-independant. But since it is operating in the Multics environment, I thought it best to be able to handle quoted strings properly. The first form will double any internal quote characters within "string". It does NOT surround "string" with quotes. "string" is limited to 16384 characters.

The second form removes quoting characters. Any doubled quotes within a quoted string will be replaced by one occurance. The quotes surrounding the string will be dropped.

```
          Processed on : &unquote &[date_time]&;
```

This kind of thing is necessary because date_time returns a quoted string.

27. Null Separator

```
   &.
   &+
```

This construct causes no output. The first form is used when there is ambiguity without it. Or when white-space skipping must be terminated.

```
          &3&.7
          &name&.suffix
```

These cases are just resolving the ambiguity. You do not want to reference "&37" or "&namesuffix".

```
          &if ... &thenƀƀƀƀƀxxƀƀƀ&fi
```

gives you an output of

```
          xxƀƀƀ
```

However,

```
          &if ... &then&.ƀƀƀƀƀxxƀƀƀ&fi
```

gives you

```
          ƀƀƀƀƀxxƀƀƀ
```

The second form is a separator which "uses up" all white-space following it. Suppose you do not want to have to say:

```
          &if ...&then&. xx&fi
```

to get the three characters of output you want. Instead you can say:

```
          &if ...
          &then
               &. xx&+
          &fi
```

28. Rescanning

```
   &scan string &;
```

The normal mode of processing is for the results of any macro construct to be considered as a protected string, that is, it is not rescanned. Sometimes this is not what is needed. This construct causes macro expansion to be done on "string" and then it is re-expanded. "string" is limited to 16384 characters. However, complete constructs must be included within "string". Suppose that the first parameter to a macro is "a,b,&[time],d".

```
                         &call(&1)
```

This will expand macro "call" with one parameter: "a,b,&[time],d".

```
                    &call(&scan &1&;)
```
Macro "call" will receive one parameter: "a,b,08:21,d".
```
                    &scan &&call(&1)&;
```
This will expand macro "call" with 4 parameters: "a", "b", "08:21", and "d";

## 29. Macro Library Reference

```
   &lib xxx1,xxx2, ... &;
```

A macro source can specify what libraries are to be searched for macro definitions. "xxxi" is the name of an archive (less the .archive) to be found via system search rules. These libraries are known through the rest of the expansion.
```
                 &lib drfdev_macros,simplex&;
```
This tells the macro processor that you want to look for macro definitions in drfdev_macros.archive and then in simplex.archive.

When a macro call is encountered, the macro is looked for in this sequence:
1) macro already used or found imbedded.
2) an initiated loose segment.
3) a loose segment in the working directory.
4) a component of an archive specified in the &lib statement.

## 30. Macro debugging

```
   &trace
```

As an aid in debugging macros, there is a tracing facility. If the second line of a macro definition consists of "&trace" then information will be printed as the macro executes. This is what a trace can look like:
```
     1    MACRO(2)  zilch
     2    ARG 1:  "abc"
     3    ARG 2:  "xyz"
     4    &loc x = 2&;
     5    arith     (2>0)
     6    log-101   (1)
     7    arith     (2-1)
     8    &let x = 1&;
     9    arith     (1>0)
     10   log-101   (1)
     11   .....&if 76:107 &then 1 &else 0 &fi
     12   arith     (1>0)
     13   log-101   (1)
     14   arith     (1-1)
     15   &let x = 0&;
     16   arith     (0>0)
     17   log-101   (0)
     18   .....&if 76:107 &then 0 &else 1 &fi
     19   arith     (0>0)
```

```
    20    log-101   (0)
    21     MEND(2)  zilch
```

Line 1 names the macro being expanded and tells the nesting depth (2). Lines 2-3 show the arguments begin passed to this macro. Line 4 shows the declaration of the local variable "x" and it being assigned an initial value. Lines 5, 7, 9, 12, 14, 16, 19 indicate an arithmetic expression which will be evaluated. The expression is printed enclosed in (); these are not part of the source.

Lines 6, 10, 13, 17, and 20 indicate a logical value which is being evaluated; it also is printed enclosed in (). "log-x0y" tells the environment under which the logical is being used; "x" represents TRUE possible; "y" represents FALSE possible. 101 and 100 mean that the result will be determined by the expression. 001 and 000 mean that the expression is being skipped over because a non-selected part of an IF is being skipped over.

Lines 8, and 15 show an assignment begin done.

Lines 11 and 18 shows the completion of an if-then-else-fi. 76:107 means that the &if begins at character 76 of the macro definition and the &fi ends at character 107 of the macro definition. The number following the "then" indicates whether this part is begin done or not; the number following the "else" indicates whether the part was done.

Line 21 indicates that the macro is finished, telling the name and nesting.

## 31. White-space

White-space means any of : HT, SP, NL, VT, FF.

White-space is always skipped over under these circumstances:
1) After these macro tokens:

```
          &;       &do      &while   &od      &error
          &usage   &scan    &substr  &length  &lib
          &quote   &unquote &if      &then    &else
          &fi      &+
```

2) Following "(" and "," in a macro call (at level 1).
3) Following the ")" in "&(expr)".
4) Following "=" in &let &loc &int and &ext.

## NOTES

Each macro construct has a very specific termination condition. When constructs are nested, the beginning and ending of each construct must be totally within any containing construct. There are 4 different types of constructs, as far as termination conditions are concerned.

First there are the types of construct which are terminated by by the even match of a delimiter pair:

```
   &" ... &"
   &[ ... ]
   &xxx(string1,string2, ... )
   &xxx{,string}
   &xxx{expr1:expr2,string}
   &xxx{expr1:expr2}
   &xxx{expr}
   &xxx{}
   &{expr1:expr2,string}
   &{expr1:expr2}
   &{expr}
```

Then there are the kinds which are self-delimiting:

```
   &&
   &*
   &+
   &.
   &n
   &nn
   &return
   &xxx
```

Also there are the kind which have a termination keyword:

```
   &do stuff1 &while yyy &; stuff2 &od
   &if xxx &then stuff1 &else stuff2 &fi
   &if xxx &then stuff1 &fi
   &macro xxx<NL>...&mend<NL>
```

And finally, &; terminates these constructs:

```
   &comment ... &;
   &error expr,string&;
   &length string&;
   &let xxx=yyy&;
   &let xxx{expr1:expr2}=yyy&;
   &let xxx{expr}=yyy&;
   &lib xxx1,xxx2, ... &;
   &loc xxx&;        (int or ext may replace loc)
   &loc xxx=yyy&;
   &loc xxx{expr1:expr2}&;
   &loc xxx{expr1:expr2}=yyy&;
   &loc xxx{expr1:expr2}var&;
   &loc xxx{expr}fifo&;
```

```
&loc xxx{expr}lifo&;
&loc xxx{expr}list&;
&quote string&;
&scan string &;
&substr string,expr1 &;
&substr string,expr1,expr2 &;
&substr string,expr1:expr2 &;
&unquote string&;
&usage string&;
&while yyy &;
```

The keywords to the macro processor are reserved. That is to say, they cannot be used as data names. It does not stop their use, however, as macro names. This is the list of current reserved words. It does contain some for future expansion.

| | | |
|---|---|---|
| arg | comment | do |
| else | empty | error |
| ext | fi | hbound |
| if | int | let |
| lbound | length | lib |
| member | loc | macro |
| mend | quote | return |
| scan | substr | unquote |
| usage | while | |

## EXAMPLES

Here is a macro which helps in referencing error_table_ names:

```
&macro et_
&int et_{50}list&;
&if &(&*=0)
&then
dcl  error_table_$&et_{, fixed bin(35)ext static;
dcl  error_table_$} fixed bin(35)ext static;
&return
&fi
&let et_=&1&;
error_table_$&1&mend
```

If you expand this segment:

```
if (code = &et_(badarg))
then code = &et_(notfound);
 ...
code = &et_(badarg);
 ...
&et_()
end;
```

You get this segment:

```
          if (code = error_table_$badarg)
          then code = error_table_$notfound;
           ...
          code = error_table_$badarg;
           ...
          dcl  error_table_$badarg fixed bin(35)ext static;
          dcl  error_table_$notfound fixed bin(35)ext static;

          end;
```

You call this macro each time you want to reference to an error_table_ value. It gives you the complete name string back, and puts the name into a list. The last call is made with no argument. Upon this condition, the list (which contains one occurance of each of the names referenced) is dumped out into a list of declarations.

## USAGE

There are 2 ways to use this macro processor: as a subroutine or as a command. The subroutine is written to be called by any procedure which wants a macro expanded. This could be a compiler, exec_com, etc. The command interface was added to allow direct use, also. Both are on System M in the directory
>udd>m>jaf>prog

Syntax: macro macroname {control_args}
        macro "&string"

Function: To expand a segment containing macro constructs. The segment macroname.macro expands into a segment named macroname.

Argument: macroname is an entryname of the segment to be expanded. It is found by macro search rules. The suffix macro is supplied if not present.

&string is any string beginning with a "&". This string is macro expanded and then printed.

Control arguments: One of the following.
-print, -pr: print the resulting expansion, instead of creating segment.
-long, -lg: print which macros were used after expansion is done.
-call XXX: if no error occurred in processing, execute XXX after expansion is complete.

Note: macro "search rules" in this case are:
1) initiated segments
2) working directory

```
call macro_(macname,outptr,outlen,arglp,argct,msg,code);
call macro_$expand(segname,outptr,outlen,arglp,argct,msg,code);

dcl macro_ entry(char(32)var,ptr,fixed bin(24),ptr,fixed bin,
                    char(200)var,fixed bin(35));
dcl macro_$expand entry(char(32)var,ptr,fixed bin(24),ptr,
                    fixed bin,char(200)var,fixed bin(35));
```

Function: To find a segment and expand it. The segment may be either a source containing macro constructs, or a macro definition.

Arguments:
macname: name of macro to expand. (IN)
segname: name of segment to expand (IN)
outptr: pointer to resulting expansion (IN)
outlen: index of last character used (IN/OUT)
arglp: pointer to argument structure (IN)
argct: number of arguments supplied (IN)
msg: text to support error code
code: standard system return code

Note:The argument structure has this form.

```
dcl 1 argl(argct) based(arglp),
    2 p ptr,        /* points to argument string */
    2 l fixed bin;  /* length of argument string */
```

External and internal macro variables, known macros, and macro library names are retained across calls to this routine. To cause these to be forgotten:

```
          call macro_$free;
```

Note: Macro libraries can be made available with this call:

```
 call macro_$library(libname);
 dcl macro_$library entry(char(32)var,fixed bin(35));
```

libname: is the name of an archive (without .archive), it will be found via system search rules.