To: J3
From: Malcolm Cohen
Subject: Macro processing facility
Date: 2019-February-14

Introduction

This document contains a draft of a possible "macro processing" subclause for the Fortran 202x standard.

Formal requirements and specifications will appear in a separate paper (not at this meeting).

There are a several issues that are not discussed in this paper, but are worthy of further consideration.

- Whether the processor should be required to be able to produce an "expanded" source code (possibly only expanding specified macros rather than all the macros in the code). There is widespread agreement that such a capability would be useful.
- As a *macro-expr* is a fully general constant expression, so that it can enquire about various processor and data type capabilities, the full macro facility really needs to be part of the processor. However, it would be reasonable for a software tool to implement a small but useful subset of those capabilities, and this would be useful to tide people over until the full version has been implemented by their compiler.
- It might be useful for a macro to be able to produce a diagnostic error message directly; if the macro can detect that there is a problem, this would allow a better error message to be produced than the standard compiler "Syntax error".
- Especially in the case of producing diagnostic messages, a macro operator to "stringify" a token or token sequence would be useful. (Even if we don't do macros but do some kind of template/parameterisedmodule instead, such a facility could be useful.)

6.5 Macro processing

6.5.1 Macro definition

A macro definition defines a macro. A defined macro shall only be referenced by a USE statement, IMPORT statement, or macro expansion statement. A defined macro shall not be redefined.

R601	*macro-definition*	**is**	*define-macro-stmt*
			[*macro-declaration-stmt*] ...
			macro-body-block
			end-macro-stmt
R602	*define-macro-stmt*	**is**	DEFINE MACRO [, *macro-attr-list*] :: *macro-name* ■
			■ [([*macro-dummy-arg-name-list*])]

C601 (R602) A *macro-dummy-arg-name* shall not appear more than once in a *macro-dummy-arg-name-list*.

R603	*macro-attr*	**is**	*access-spec*

The DEFINE MACRO statement begins the definition of the macro *macro-name*. Appearance of an *access-spec* in the DEFINE MACRO statement explicitly gives the macro the specified attribute (ref:"Accessibility attribute"). Each *macro-dummy-arg-name* is a macro dummy argument. A macro dummy argument is a macro local variable.

R604	*macro-declaration-stmt*	**is**	*macro-type-declaration-stmt*
		or	*macro-optional-decl-stmt*
		or	*macro-variable-decl-stmt*
R605	*macro-type-declaration-stmt*	**is**	MACRO *macro-type-spec* :: *macro-local-variable-name-list*
R606	*macro-optional-decl-stmt*	**is**	MACRO OPTIONAL :: *macro-dummy-arg-name-list*
R607	*macro-variable-decl-stmt*	**is**	MACRO VARIABLE :: *macro-local-variable-name-list*
R608	*macro-type-spec*	**is**	INTEGER [([KIND=] *macro-expr*)]

C602 (R605, R607) A *macro-local-variable-name* shall not be the same as the name of a macro dummy argument of the macro being defined.

C603 (R606) A *macro-dummy-arg-name* shall be the name of a macro dummy argument of the macro being defined.

C604 (R608) If *macro-expr* appears, when the macro is expanded *macro-expr* shall be of type integer, and have a non-negative value that specifies a representation method that exists on the processor.

A macro type declaration statement specifies that the named entities are macro local variables of the specified type. If the kind is not specified, they are of default kind. A macro variable declaration statement declares untyped macro local variables; the value of an untyped macro local variable is a token sequence, and its initial value is an empty sequence (no tokens). A macro local variable that is not a macro dummy argument shall appear in a macro type declaration statement or in a macro variable declaration statement.

R609	*macro-body-block*	**is**	[*macro-body-construct*] ...
R610	*macro-body-construct*	**is**	*macro-definition*
		or	*expand-stmt*
		or	*macro-body-stmt*

		or	*macro-do-construct*
		or	*macro-if-construct*
		or	*macro-int-assignment-stmt*
		or	*macro-tok-assignment-stmt*

C605 A statement in a macro definition that is not a *macro-body-construct* or *macro-definition* shall not appear on a line with any other statement.

R611	*macro-do-construct*	**is**	*macro-do-stmt* *macro-body-block* *macro-end-do-stmt*
R612	*macro-do-stmt*	**is**	MACRO DO *macro-do-variable-name* = *macro-do-limit* , ■ ■ *macro-do-limit* [, *macro-do-limit*]

C606 (R612) A *macro-do-variable-name* shall be a local variable of the macro being defined, and shall be of type integer.

R613	*macro-do-limit*	**is**	*macro-expr*

C607 (R613) A *macro-do-limit* shall expand to an expression of type integer.

R614	*macro-end-do-stmt*	**is**	MACRO END DO

A macro DO construct iterates the expansion of its enclosed macro body block at macro expansion time. The number of iterations is determined by the values of the expanded macro expressions in the MACRO DO statement.

R615	*macro-if-construct*	**is**	*macro-if-then-stmt* *macro-body-block* [*macro-else-if-stmt* *macro-body-block*] ... [*macro-else-stmt* *macro-body-block*] *macro-end-if-stmt*
R616	*macro-if-then-stmt*	**is**	MACRO IF (*macro-condition*) THEN
R617	*macro-else-if-stmt*	**is**	MACRO ELSE IF (*macro-condition*) THEN
R618	*macro-else-stmt*	**is**	MACRO ELSE
R619	*macro-end-if-stmt*	**is**	MACRO END IF
R620	*macro-condition*	**is**	*macro-expr*

C608 (R620) A macro condition shall expand to an expression of type logical.

A macro IF construct provides conditional expansion of its enclosed macro body blocks at macro expansion time. Whether the enclosed macro body blocks contribute to the macro expansion is determined by the logical value of the expanded macro expressions in the MACRO IF and MACRO ELSE IF statements.

R621	*macro-int-assignment-stmt*	**is**	MACRO *macro-integer-variable-name* = *macro-expr*

C609 (R621) *macro-integer-variable-name* shall be the name of a macro local variable of type integer.

R622	*macro-tok-assignment-stmt*	**is**	MACRO *macro-tok-variable-name* = *assignment-tok-sequence*

C610 (R622) *macro-tok-variable-name* shall be the name of an untyped macro local variable that is not a macro dummy argument.

R623 *assignment-tok-sequence* **is** [*result-token*] ... [&&]

R624 *macro-body-stmt* **is** *result-token* [*result-token*] ... [&&]

C611 (R624) If the first *result-token* is MACRO the second *result-token* shall not be a keyword or name.

C612 (R624) If the first *result-token* is DEFINE or END, the second *result-token* shall not be MACRO.

R625 *result-token* **is** *token* [%% *token*] ...

R626 *token* **is** any lexical token including labels, keywords, and semi-colon.

C613 && shall not appear in the last *macro-body-stmt* of a macro definition.

C614 When a macro is expanded, the last *macro-body-stmt* processed shall not end with &&.

R627 *end-macro-stmt* **is** END MACRO [*macro-name*]

C615 (R601) The *macro-name* in the END MACRO statement shall be the same as the *macro-name* in the DEFINE MACRO statement.

R628 *macro-expr* **is** *basic-token-sequence*

C616 (R628) A *macro-expr* shall expand to a scalar constant expression.

Macro expressions are used to control the behavior of the MACRO DO and MACRO IF constructs when a macro is being expanded. The type, type parameters, and value of a macro expression are determined when that macro expression is expanded.

6.5.2 Macro expansion

6.5.2.1 General

Macro expansion is the conceptual replacement of the EXPAND statement with the Fortran statements that it produces. The semantics of an EXPAND statement are those of the Fortran statements that it produces. It is recommended that a processor be capable of displaying the results of macro expansion. It is processor-dependent whether comments in a macro definition appear in the expansion. It is processor-dependent whether continuations and consecutive blanks that are not part of a token are preserved.

The process of macro expansion produces Fortran statements consisting of tokens. The combined length of the tokens for a single statement, plus inter-token spacing, shall not be greater than 1 000 000 characters.

NOTE 6.1

This length is the same as the minimum limit on statement length permitted by the standard.

Note that breaking tokens across continuation lines in macro definitions and in EXPAND statements does not affect macro expansion: it is as if they were joined together before replacement.

R629 *expand-stmt* **is** EXPAND *macro-name* [(*macro-actual-arg-list*)]

C617 (R629) *macro-name* shall be the name of a previously defined macro.

C618 (R629) The macro shall expand to a sequence of zero or more complete Fortran statements.

C619 (R629) The statements produced by a macro expansion shall conform to the syntax rules and constraints as if they replaced the EXPAND statement prior to program processing.

C620 (R629) The statements produced by a macro expansion shall not include a statement which ends the scoping unit containing the EXPAND statement.

C621 (R629) If a macro expansion produces a statement which begins a new scoping unit, it shall also produce a statement which ends that scoping unit.

C622 (R629) If the EXPAND statement appears as the *action-stmt* of an *if-stmt*, it shall expand to exactly one *action-stmt* that is not an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, or *if-stmt*.

C623 (R629) If the EXPAND statement appears as a *do-term-action-stmt*, it shall expand to exactly one *action-stmt* that is not an *arithmetic-if-stmt*, *continue-stmt*, *cycle-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.

C624 (R629) If the EXPAND statement has a label, the expansion of the macro shall produce at least one statement, and the first statement produced shall not have a label.

C625 (R629) A *macro-actual-arg* shall appear corresponding to each nonoptional macro dummy argument.

C626 (R629) At most one *macro-actual-arg* shall appear corresponding to each optional macro dummy argument.

Expansion of a macro is performed by the EXPAND statement. If the EXPAND statement has a label, the label is interpreted after expansion as belonging to the first statement of the expansion.

R630 *macro-actual-arg* **is** [*macro-dummy-name* =] *macro-actual-arg-value*

C627 (R630) *macro-dummy-name* shall be the name of a macro dummy argument of the macro being expanded.

C628 (R629) The *macro-dummy-name*= shall not be omitted unless it has been omitted from each preceding *macro-actual-arg* in the *expand-stmt*.

C629 (R630) If the first two tokens of *macro-actual-arg-value* are a name and an equals sign, *macro-dummy-name*= shall appear.

R631 *macro-actual-arg-value* **is** *basic-token-sequence*

R632 *basic-token-sequence* **is** *basic-token*
or [*basic-token-sequence*] *nested-token-sequence* ■
■ [*basic-token-sequence*]
or *basic-token basic-token-sequence*

R633 *basic-token* **is** any lexical token except comma, parentheses, array ■
■ constructor delimiters, and semi-colon.

R634 *nested-token-sequence* **is** ([*arg-token*] ...)
or (/ [*arg-token*] ... /)
or *lbracket* [*arg-token*] ... *rbracket*

R635	*arg-token*	**is**	*basic-token*
		or	,

If a macro actual argument is not preceded by *macro-dummy-name* it corresponds to the macro dummy argument in the same position in the macro declaration; otherwise it corresponds to the macro dummy argument having the specified name.

Macro expansion processes any macro declarations of the macro definition, and then expands its macro body block. Any macro expressions in *macro-type-spec*s are evaluated and the kinds of the macro variables thereby declared are determined for that particular expansion.

Macro expansion of a macro body block processes each macro body construct of the macro body block in turn, starting with the first macro body construct and ending with the last macro body construct.

Expansion of a statement within a macro body construct consists of three steps:

(1) token replacement,
(2) token concatenation, and
(3) statement-dependent processing.

6.5.2.2 Token replacement

Token replacement replaces each token of a macro body statement, assignment token sequence, or macro expression that is a macro local variable with the value of that variable.

A macro dummy argument is present if and only if it corresponds to a macro actual argument.

In a macro expression, a reference to the intrinsic function PRESENT with a macro dummy argument name as its actual argument is replaced by the token `.TRUE.` if the specified macro dummy argument is present, and the token `.FALSE.` if the specified macro dummy argument is not present. Otherwise, the value of a macro dummy argument that is present is the sequence of tokens from the corresponding macro actual argument, and the value of a macro dummy argument that is not present is a zero-length token sequence.

The value of an integer macro variable is its minimal-length decimal representation; if negative this produces two tokens, a minus sign and an unsigned integer literal constant. An untyped macro local variable expands to the sequence of tokens assigned to it, or to a zero-length token sequence if no tokens are assigned to it.

6.5.2.3 Token concatenation

Token concatenation is performed with the %% operator, which is only permitted inside a macro definition. After expansion, each sequence of single tokens separated by %% operators is replaced by a single token consisting of the concatenated text of the sequence of tokens. The result of a concatenation shall be a valid Fortran token, and may be a different kind of token from one or more of the original sequence of tokens.

C630 (R625) The result of token concatenation shall have the form of a lexical token.

NOTE 6.2

For example, the sequence

```
    3 %% .14159 %% E %% + %% 0
```

forms the single real literal constant 3.14159E+0.

6.5.2.4 Macro body statements

Processing a macro body statement produces a whole or partial Fortran statement. A macro body statement that is either the first macro body statement processed by this macro expansion or the next macro body statement processed after a macro body statement that did not end with the macro continuation operator &&, is an initial macro body statement. The next macro body statement processed after a macro body statement that ends with && is a continuation macro body statement. An initial macro body statement that does not end with && produces a whole Fortran statement consisting of its token sequence. Each other macro body statement produces a partial Fortran statement, and the sequence of tokens starting with those produced by the initial macro body statement and appending the tokens produced by each subsequent continuation macro body statement form a Fortran statement. The && operators are not included in the token sequence.

6.5.2.5 The MACRO DO construct

The MACRO DO construct specifies the repeated expansion of a macro body block. Processing the MACRO DO statement performs the following steps in sequence.

(1) The initial parameter m_1, the terminal parameter m_2, and the incrementation parameter m_3 are of type integer with the same kind type parameter as the *macro-do-variable-name*. Their values are given by the first *macro-expr*, the second *macro-expr*, and the third *macro-expr* of the *macro-do-stmt* respectively, including, if necessary, conversion to the kind type parameter of the *macro-do-variable-name* according to the rules for numeric conversion (Table ref:Numeric conversion and the assignment statement). If the third *macro-expr* does not appear, m_3 has the value 1. The value of m_3 shall not be zero.

(2) The MACRO DO variable becomes defined with the value of the initial parameter m_1.

(3) The iteration count is established and is the value of the expression $(m_2 - m_1 + m_3)/m_3$, unless that value is negative, in which case the iteration count is 0.

After this, the following steps are performed repeatedly until processing of the MACRO DO construct is finished.

(1) The iteration count is tested. If it is zero, the loop terminates and processing of the MACRO DO construct is finished.

(2) If the iteration count is nonzero, the macro body block of the MACRO DO construct is expanded.

(3) The iteration count is decremented by one. The MACRO DO variable is incremented by the value of the incrementation parameter m_3.

6.5.2.6 The MACRO IF construct

The MACRO IF construct provides conditional expansion of macro body blocks. At most one of the macro body blocks of the MACRO IF construct is expanded. The macro conditions of the construct are evaluated in order until a true value is found or a MACRO ELSE or MACRO END IF statement is encountered. If a true value or a MACRO ELSE statement is found, the macro body block immediately following is expanded and this completes the processing of the construct. If none of the evaluated conditions is true and there is no MACRO ELSE statement, the processing of the construct is completed without expanding any of the macro body blocks within the construct.

6.5.2.7 Macro assignment

Processing a macro integer assignment statement sets the macro local variable value to that of the macro expression.

Processing a macro token assignment statement sets the macro local variable value to be the sequence of tokens following the equals sign. If no tokens appear after the equals sign, the macro local variable is set to the zero-length token sequence.

6.5.2.8 Macro definitions

Processing a macro definition defines a new macro. If a macro definition is produced by a macro expansion, all of the statements of the produced macro definition have token replacement and concatenation applied to them before the new macro is defined.

6.5.2.9 Examples

NOTE 6.3

This is a macro which loops over an array of any rank and processes each array element.

```
  DEFINE MACRO loop_over(array,rank,traceinfo)
    MACRO INTEGER :: i
      BLOCK
    MACRO DO i=1,rank
        INTEGER loop_over_temp_%%i
    MACRO END DO
    MACRO DO i=1,rank
        DO loop_over_temp_%%i=1,size(array,i)
    MACRO END DO
          CALL impure_scalar_procedure(array(loop_over_temp_%%1 &&
    MACRO DO i=2,rank
                                              ,loop_over_temp_%%i &&
    MACRO END DO
                                               ),traceinfo)
    MACRO DO i=1,rank
        END DO
    MACRO END DO
      END BLOCK
  END MACRO
```

NOTE 6.4

One can effectively pass macro names as macro arguments, since expansion of arguments occurs before analysis of each macro body statement. For example:

```
      DEFINE MACRO :: iterator(count,operation)
        MACRO DO i=1,count
          EXPAND operation(i)
        MACRO END DO
      END MACRO

      DEFINE MACRO :: process_element(j)
        READ *,a(j)
        result(j) = process(a(j))
        IF (j>1) PRINT *,'difference =',result(j)-result(j-1)
      END MACRO

      EXPAND iterator(17,process_element)
```

This expands into 17 sets of 3 statements:

NOTE 6.4 (cont.)

```
        READ *,a(1)
        result(1) = process(a(1))
        IF (1>1) PRINT *,'difference =',result(1)-result(1-1)
        READ *,a(2)
        result(2) = process(a(2))
        IF (2>1) PRINT *,'difference =',result(2)-result(2-1)
        ...
        READ *,a(17)
        result(17) = process(a(17))
        IF (17>1) PRINT *,'difference =',result(17)-result(17-1)
```

NOTE 6.5

Using the ability to evaluate constant expressions under macro control and the kind value arrays from ISO_FORTRAN_ENV, one can create interfaces and procedures for all kinds of a type, for example:

```
  DEFINE MACRO :: i_square_procs()
    MACRO INTEGER i, thiskind
      MACRO DO i=1, size(INTEGER_KINDS)
        MACRO thiskind = INTEGER_KINDS(i)
          FUNCTION i_square_kind_%%thiskind (a) RESULT(r)
          INTEGER(thiskind) a,r
          r = a**2
          END FUNCTION
      MACRO END DO
  END MACRO
```

NOTE 6.6

Macros can be used to define other macros on expansion. For example,

```
! Macro that defines a macro which assigns a value to an array element
DEFINE MACRO :: assign_shortcut(rank)
   DEFINE MACRO assign_%%rank(array,indices,value)
      MACRO INTEGER :: i
      array(indices(1)&&
      MACRO DO i=2,rank
           ,indices(i)&&
      MACRO END DO
            )=value
   END MACRO assign_%%rank
END MACRO assign_shortcut

! Create assignment macros for all ranks
MACRO DO i=1,15
   EXPAND assign_shortcut(i)
MACRO END DO

! Now use the rank-3 assignment macro:
REAL :: A(10,10,10)
INTEGER :: indices(3)=[1,5,6]
EXPAND assign_3(A,indices,5.0)
```

NOTE 6.6 (cont.)

```
! Expands to:
! A(indices(1),indices(2),indices(3))=5.0
```

NOTE 6.7

This example demonstrates the use of MACRO IF to generate an interface for subroutines acting on single, double, and (if it exists) quad precision real.

```
DEFINE MACRO my_generic_interface(typename,array_of_kinds)
  MACRO INTEGER :: i, kind
  INTERFACE my_generic_procedure
     MACRO DO i=1, SIZE(array_of_kinds)
        ! Necessary in order to evaluate kind to an integer:
        MACRO kind = array_of_kinds(i)
        MACRO IF (kind>0) THEN
           SUBROUTINE MySpecificProcedure_%%kind(X)
              typename(kind), INTENT(IN) :: X
           END SUBROUTINE
        MACRO END IF
     MACRO END DO
  END INTERFACE
END MACRO my_generic_interface
```

Use of the macro:

```
INTEGER,PARAMETER :: rkinds(3) = [KIND(0.0),KIND(0d0), &
                                  SELECTED_REAL_KIND(P=PRECISION(0d0)*2)]

EXPAND my_generic_interface(REAL,rkinds)
```