To: SC22WG5
From: David Epstein
Subject: Proposed Part 3 of Fortran Standard

CONDITIONAL COMPILATION IN FORTRAN

ISO / IEC 1539-3 : 1996

{Auxiliary to ISO / IEC 1539 : 1996 "Programming Language Fortran"}

CONTENTS

1. INTRODUCTION ---

This part of ISO / IEC 1539 has been prepared by ISO / IEC JTC1 / SC22 / WG5, the technical working group for the Fortran language. This part of ISO / IEC 1539 is an auxiliary standard to ISO / IEC 1539 : 1996, which defines the latest revision of the Fortran language, and is the first part of the multipart Fortran family of standards; this part of ISO / IEC 1539 is the third part. The revised language defined by the above standard is informally known as Fortran 95.

This part of ISO / IEC 1539 defines a conditional compilation language definition.

2. RATIONALE ---

Frequently Fortran programmers need to maintain more than one version of a code, or to run the code in various environments. The easiest solution for the programmer is to keep a single source file that has all the code variations interleaved within it so that any version can be easily extracted. This way, modifications that apply to all versions need only be made once.

Conditional compilation permits the programmer to define special
variables and logical constucts that conditionally control which
source lines in the file are passed on to the compiler and which
lines are skipped over. [1]

There are many uses for conditional compilation. Writing portable
code is one of the most popular uses of conditional compilation.
Examples in Annex A show some other uses for conditional compilation.

3. GENERAL ---

1. Scope

This part of ISO/IEC 1539 defines facilities for use in Fortran for
conditional compilation. This part of ISO/IEC 1539 provides an
auxiliary standard for the version of the Fortran language informally
known as Fortran 95. The international Standard defining this revision
of the Fortran language is

ISO/IEC 1539 : 1996 "Programming Language Fortran"

2. Normative References

The following standard contains provisions which, through reference in
this text, constitute provisions of this part of IOS/IEC 1539. At the
time of publication, the edition indicated was valid. All standards
are subject to revision, and parties to agreements based on this part
of IOS/IEC 1539 are encouraged to investigate the possibility of
applying the most recent editions of the standard indicated below.
Members of IEC and ISO maintain registers of currently valid
International Standards.

IOS/IEC 1539 : 1996, Information technology--Programming
Languages--Fortran.

4. THE CONDITIONAL COMPILATION LANGUAGE DEFINITION ----------------

Section 1 Conditional Compilation

The conditional compilation language definition for Fortran consists of
nine directives and one option. The one option together with the
conditional compilation directives offer the power to select alternative
blocks of code for compilation without modifying the source text.

Section 2 High level syntax

This section introduces the terms associated with the conditional
compilation program.

```
R201   coco-program    is   coco-directive
                       [ coco-directive ] ...

R202   coco-directive   is   coco-type-decl-directive
                    or   coco-executable-construct
                    or   INCLUDE line

R203   coco-executable-construct   is    coco-action-directive
                              or    coco-if-construct

R204   coco-action-directive       is  coco-assignment-directive
                              or  coco-error-directive
```

Section 3 Constants, source form and including text

3.1 Coco constants

```
R301   coco-constant          is  coco-literal-constant
                         or  coco-named-constant

R302   coco-literal-constant   is   coco-int-literal-constant
                         or   coco-logical-literal-constant

R303   coco-int-literal-constant      is     digit-string

R304   coco-logical-literal-constant   is     .TRUE.
                              or     .FALSE.

R305   coco-char-literal-constant      is     ' [ rep-char ] ...  '
                              or     " [ rep-char ] ...  "

R306   coco-named-constant     is   name
```

[A name is specified in part 1, section x.y of this standard.]

3.2 Coco source form

A coco program is a sequence of one or more lines, organized as coco directives and coco comments. A coco directive is a sequence of one or more complete or partial lines. A line is a sequence of zero or more characters.

A coco character context means characters within a coco character literal constant (R305).

A coco comment may contain any character that may occur in any coco character context.

In coco source form, each source line may contain from zero to

132 characters and there are restrictions on where a coco directive (or portion of a coco directive) may appear within a line. However, if a line contains any character that is not of default kind the number of characters allowed on the line is processor dependent.

[Default kind is specified in part 1, section 4.3.2.1 of this standard.]

In coco source form, blank characters shall not appear within coco lexical tokens other than in a coco character context. Blanks may be inserted freely between coco tokens to improve readability. A sequence of blank characters outside of a coco character context is equivalent to a single blank character.

[Lexical token is specified in part 1, section x.y of this standard.]

A blank shall be used to separate coco names, or coco constants from adjacent keywords, coco names, or coco constants.

3.2.1 Coco directive start

R307 coco-dir-start is ??

Constraint: The characters "??" shall be in character positions 1 and 2 of the source line.

3.2.2 Coco commentary

A line that does not start with a coco-dir-start and is not an INCLUDE line is called a coco comment line. Within a coco directive, the character "!" in any character position initiates a coco comment except when it appears within a coco character context. The comment extends to the end of the source line. If the first nonblank character on a line after a coco-dir-start is an "!", the line is also a coco comment line. Lines containing only blanks after a coco-dir-start or containing no characters after a coco-dir-start are also coco comment lines.

Note 3.1
Examples of coco comment lines are:

```
gender = GetRabbitGender(a_rabbit)
! a Fortran comment
?? ! set "system" to "system_A"
??
?>       OPEN(UNIT=6, FILE="system.dependent.name")
?*     OPEN(UNIT=6, FILE="sysdpnt.nam")
```

ENDNote 3.1

3.2.3 Coco directive continuation

The character "&" is used to indicate that the current coco directive is continued on the next line. The next line shall begin with a coco-dir-start. Coco comment lines shall not be continued; an "&" in a coco comment has no effect. When used for continuation, the "&" is not part of the coco directive. No line shall contain a single "&" as the only nonblank character after a coco-dir-start in a coco directive or as the only nonblank character after the coco-dir-start and before an "!" that initiates a comment.

3.2.3.1 Coco-noncharacter context continuation

In a coco directive, if an "&" not in a coco comment is the last nonblank character on a line or the last nonblank character before an "!" that initiates a comment, the coco directive is continued on the next line. If the first nonblank character after a coco-dir-start on the next coco-noncomment line is an "&", the coco directive continues at the next character following the "&"; otherwise, it continues with the first character position after the coco-dir-start of the next coco-noncomment line.

If a coco lexical token is split across the end of a line, the first nonblank character after the coco-dir-start on the first following coco-noncomment line shall be an "&" immediately followed by the successive characters of the split token.

Note 3.2
An example of coco-noncharacter context continuation is:

```
?? LOGICAL :: TOO_GOOD&
 ??&_TO_BE_&
??&TRUE =        &
??         .FALSE.  ! These four lines contain 1 coco directive
```
ENDNote 3.2

3.2.3.2 Coco character context continuation

If a coco character context is to be continued, the "&" shall be the last nonblank character on the line and shall not be followed by coco commentary. An "&" shall be the first nonblank character after the coco-dir-start on the next line and the coco directive continues with the next character following the "&".

Note 3.3
An example of coco character context continuation is:

```
?? ERROR "de&
??       &f&
??          &inately choosing Fortran" ! 3 lines, 1 coco directive
```
ENDNote 3.3

3.2.4 Coco directives

A coco directive shall begin with a coco-dir-start. If a coco directive has one or more continuation lines, every line from the start of the coco directive until the end of the coco directive shall begin with a coco-dir-start.

A coco directive shall not have more than 39 continuation lines.

Note 3.4
Examples of coco directives are:

```
?? INTEGER, PARAMETER :: SYSTEM_A = 1
?? ERROR "system_A = ", SYSTEM_A
??   IF (.FALSE.) THEN
??   ENDIF
```
ENDNote 3.4

3.3 Including source text

Additional text may be incorporated into the source text of a coco program during processing. This is accomplished with the INCLUDE line, which has the form

INCLUDE char-literal-constant

[The INCLUDE line is specified in part 1, section 3.4 of this standard.]

Included source text cannot directly or indirectly include itself.

An INCLUDE line in a coco FALSE block (6.2.2) is not expanded. Any coco-else-if-directive, coco-else-directive and coco-endif-directive shall not appear in included source text unless the matching coco-if-directive appears in the same included source text.

Section 4 Coco type declaration directives

R401 coco-type-declaration-directive is

coco-type-spec [, PARAMETER] :: coco-entity-decl-list

```
R402   coco-type-spec      is     INTEGER
                           or     LOGICAL
```

R403 coco-entity-decl is

coco-object-name [coco-initialization]

Constraint: A coco-object-name shall not be the same as any other coco-object-name.

```
R404    coco-object-name     is      object-name
```

[Object-name token is specified in part 1, section x.y of this standard.]

```
R405    coco-initialization  is      = coco-initialization-expr
```

Constraint: The types of the coco-initialization-expr and the coco-type-spec shall either both be integer or both be logical.

Constraint: In a coco-type-declaration-directive, if the PARAMETER attribute is specified, a coco-initialization shall appear for every coco-object-name.

Note 4.1
Examples of coco type declaration directives are:

```
?? INTEGER, PARAMETER :: F77 = 1, F90 = 2, F95 = 3
?? INTEGER :: FORTRAN_LEVEL = F95
?? LOGICAL :: DEBUG_PROCEDURE_ENTRY_EXIT
```

ENDNote 4.1

Section 5 Coco variables, expressions and assignment directive

5.1 Coco variables

```
R501    coco-variable       is      coco-variable-name
```

Constraint: coco-variable-name shall not have the PARAMETER attribute.

Constraint: coco-variable-name shall appear as an object-name in a coco-type-declaration-directive before appearing elsewhere in a coco program.

5.2 Coco expressions

5.2.1 Coco primary

```
R502    coco-primary  is     coco-constant
                     or     coco-variable
                     or     ( coco-expr )
```

Constraint: A coco-variable shall be defined (8.2) before appearing as a coco-primary.

5.2.2 Level-1 expressions

R503 coco-add-operand is

[coco-add-operand mult-op] primary

[mult-op is specified in part 1, section x.y of this standard.]

R504 coco-level-1-expr is

[[coco-level-1-expr] add-op] coco-add-operand

[add-op is specified in part 1, section x.y of this standard.]

5.2.3 Level-2 expressions

R505 coco-level-2-expr is

[coco-level-1-expr rel-op] coco-level-1-expr

[rel-op is specified in part 1, section x.y of this standard.]

5.2.4 Level-3 expressions

R506 coco-and-operand is

[not-op] coco-level-2-expr

[not-op is specified in part 1, section x.y of this standard.]

R507 coco-or-operand is

[coco-or-operand and-op] coco-and-operand

[and-op is specified in part 1, section x.y of this standard.]

R508 coco-equiv-operand is

[coco-equiv-operand or-op] coco-or-operand

[or-op is specified in part 1, section x.y of this standard.]

R509 coco-level-3-expr is

[coco-level-3-expr equiv-op] coco-equiv-operand

[equiv-op is specified in part 1, section x.y of this standard.]

5.2.5 General form of a coco expression

R510 coco-expr is coco-level-3-expr

5.3 Data type of a coco expression

The data type of a coco expression is either Integer or Logical.

[The data type of a coco expression is specified in part 1, table 7.1 of this standard.]

R511 coco-logical-expr is coco-expr

Constraint: coco-logical-expr shall be type logical.

5.4 Coco initialization expression

R512 coco-initialization-expr is coco-expr

Constraint: A coco-initialization-expr shall be an initialization expression.

[Initialization expression is specified in part 1, section x.y of this standard.]

Note 5.1
The one place in the coco language where an intialization expression is used is the coco type declaration directives.
ENDNote 5.1

5.5 Coco assignment directive

A coco variable may be defined or redefined by execution of a coco assignment directive.

R513 coco-assignment-directive is

coco-variable = coco-expr

where coco-variable is defined in R501 and coco-expr is defined in R510.

In a coco assignment directive, the types of coco-variable and coco-expr shall either both be integer or both be logical.

Note 5.2
Examples of coco assignment directives are:

```
?? DEBUG_LEVEL = DEBUG_LEVEL + 1
```

```
 ?? IS_COMPANY_X_MACHINE = (SYSTEM == SYS_E) .OR. (SYSTEM ==
SYS_F)
 ??    PROJECT_LEVEL = FOO_VERSION + LATEST_RELEASE
```
ENDNote 5.2

Section 6 Coco execution control and conditional compilation

The execution sequence and conditional compilation are controlled by coco if constructs.

Note 6.1
A coco program is not required to contain any coco if constructs, coco error directives or INCLUDE lines. Execution of such a coco program has no effect.
ENDNote 6.1

6.1 Coco blocks

A coco block is a sequence of coco directives that are treated as a unit.

```
R601   coco-block   is     [ coco-directive ] ...
```

Coco executable constructs may be used to control which coco blocks of a coco program are executed.

6.2 Coco IF construct

The coco IF construct selects for execution no more than one of its constituent coco blocks. This coco block is called the coco TRUE block. The remainder of the coco blocks in a coco IF construct, if any, are selected to be ignored by the processor. These coco blocks are called coco FALSE blocks.

6.2.1 Form of the coco IF construct

```
R602   coco-if-construct   is     coco-if-then-directive
                                    coco-block
                                [ coco-else-if-directive
                                    coco-block ] ...
                                [ coco-else-directive
                                    coco-block ]
                                coco-end-if-directive
```

R603 coco-if-then-directive is

IF (coco-logical-expr) THEN

R604 coco-else-if-directive is

ELSE IF (coco-logical-expr) THEN

R605 coco-else-directive is ELSE

R606 coco-end-if-directive is END IF

Note 6.2
An example of two coco if constructs, one nested within the
other, is:

```
?? IF (IS_COMPANY_X_MACHINE) THEN
??  IF (FORTRAN_LEVEL == F95) THEN
       PURE FUNCTION GET_RABBIT_WEIGHT(A_RABBIT)
RESULT(WEIGHT)
       TYPE (RABBIT), INTENT(IN) :: A_RABBIT
??  ELSEIF (FORTRAN_LEVEL == F90) THEN
       FUNCTION GET_RABBIT_WEIGHT(A_RABBIT) RESULT(WEIGHT)
       TYPE (RABBIT) :: A_RABBIT
??  ELSE
??   ERROR "We do not have a FORTRAN 77 product from company X"
??  ENDIF
?? ELSE
       FUNCTION GET_RABBIT_WEIGHT() RESULT(WEIGHT)
      ! Only have Fortran 90 derived types with company X, so
      ! return a weight of 1 for now
?? ENDIF
```
ENDNote 6.2

6.2.2 Execution of an IF construct

At most one of the coco blocks in the coco IF construct is executed.
If there is a coco ELSE directive in the construct, exactly one of
the coco blocks in the construct will be executed. The coco
logical expressions are evaluated in the order of their appearance
in the construct until a true value is found or a coco ELSE directive
or coco END IF directive is encountered. If a true value or a coco
ELSE directive is found, the coco block immediately following is
executed and this completes the execution of the construct. The
coco logical expressions in any remaining coco ELSE IF
directives of the coco IF construct are not evaluated. If none of
the evaluated expressions are true and there is no coco ELSE directive,
the execution of the construct is completed without the execution of

any coco block within the construct.

Execution of a coco END IF directive has no effect.

6.2.2.1 Execution of a coco TRUE block

Source lines contained in a coco TRUE block are selected as possible Fortran statements.

6.2.2.2 Execution of a coco FALSE block

Source lines contained in a coco FALSE block are selected as lines to be ignored by noncoco processing.

Note 6.3
If a processor offers an output file as a result of coco processing, possible options on the handling of lines in a coco FALSE block are:
(1) delete them,
(2) replace them with blank lines,
(3) replace them with a "!" in character position 1 followed by the original source line shifted one to the right, or
(4) replace them with a "!" in character position 1 followed by the the characters starting from character position 2 of the original source line.
ENDNote 6.3

Section 7 Coco error directive and coco stop directive

```
R701    coco-error-directive   is

  ERROR [ coco-output-item-list ]

R702    coco-output-item     is     coco-expr
                             or     coco-char-literal-constant
```

Execution of a coco ERROR directive specifies that an error has occurred during coco processing. At the time of execution of a coco ERROR directive, the output-item-list, if any, is available in a processor-dependent manner.

Note 7.1
Examples of the coco error directive are:

```
?? ERROR "system shall be set to 'sys1' or 'sys2'"
?? ERROR "system = ", SYSTEM
?? ERROR
```
ENDNote 7.1

```
R703    coco-stop-directive   is    STOP
```

Execution of a coco STOP directive specifies that the programmer desires to stop processing.

Section 8 Scope and definition of coco variables

8.1 Scope of coco variables

Coco variables have the scope of the coco program in which they are declared.

8.2 Events that cause coco variables to become defined

Coco variables become defined as follows:

(1)   Execution of a coco assignment directive causes the coco variable that precedes the equals to become defined.

(2)   Execution of a coco initialization in a coco type declaration directive causes the coco variable that precedes the equals to become defined.

(3)   Execution of the coco-set-option (Section 10) causes the coco variable to become defined.

Section 9 Coco program conformance

9.1 A conforming coco program

A coco program shall contain coco directives that abide by the syntax and semantic rules previously described in this part of the standard. A processor shall accept coco programs and coco-set-options (Section 10).

9.2 A nonconforming coco program

A nonconforming coco program is one that violates the coco syntax or constraints. A processor shall have a mechanism to report a nonconforming coco program.

Section 10 The coco SET option

There is one coco option--the coco-set-option. A processor shall supply at least one method of recognizing the coco-set-option separate from the coco program. For example, the invocation line may be the chosen method of communicating the coco-set-option to the processor.

Note 10.1
Another method of communicating the coco-set-option to the processor

could be a coco input file.
ENDNote 10.1

The coco-set-option is a method of either
(1) documenting the value of a coco PARAMETER or
(2) assigning an initial value to a coco variable (R501) or
(3) overriding the initial value assigned to a coco variable in
a coco initialization expression (R512)
without editing the coco program.

Note 10.2
Recall that a coco variable shall not have the PARAMETER attribute.
ENDNote 10.2

R1001 coco-set-option is

coco-set-option-variable=coco-set-option-literal-constant

or

processor-defined-coco-set-option

R1002 coco-set-option-variable is coco-variable

R1003 coco-set-option-literal-constant is coco-literal-constant

Constraint: A processor-defined-coco-set-option shall contain a
coco-set-option-variable and a coco-set-option-literal-constant.

Constraint: The coco-set-option-variable must be a coco variable
declared in the coco type declaration directive.

Constraint: The type of the coco-set-option-variable
shall match the type of the coco-set-option-literal-constant.

Constraint: If the coco-set-option-variable has the PARAMETER
attribute, the value of the coco-set-option-literal-constant
shall match the value supplied in the coco type declaration directive.

Constaint: If the coco-set-option-variable appears in the coco
program as the coco variable in a coco assignment directive, it
shall appear at least once in a previous coco executable construct.

The coco-set-option minimally consists of a coco-set-option-variable
and a coco-set-option-literal-constant. A processor may supply
additional representations for the coco logical literal constant;
for example, the characters 'T' or 'F' could be used to represent
.TRUE. or .FALSE. respectively. In this case it is as if .TRUE.
or .FALSE. were specified as the coco-set-option-literal-constant.

The coco-set-option acts as if a coco assignment directive (R513)--
which consists of the coco-set-option-variable as the coco variable
and the coco-set-option-literal-constant as the coco expression
(R510)--were placed immediately following the coco type declaration
directive that declared the coco-set-option-variable.

Note 10.4
The value assigned to a coco variable with the coco-set-option shall
not override the value assigned to a coco variable with the coco
assignment directive.
ENDNote 10.4

Annex A : EXAMPLES --

This annex includes two examples illustrating the use of facilities
conformant with this part of ISO/IEC 1539.

The first example uses conditional compilation to facilitate the
editing of a large block comment.

The second example uses conditional compilation to provide
debugging information upon entering and exiting procedures.
Note, the conditional compilation directives in this example
could be automatically generated.

Each example contains a conditional compilation program and a possible
output file from conditional compilation processing.

```
--- initial text -----------------------------------------
! EXAMPLE 1 shows a possible shift file for output
?? LOGICAL :: MODIFYING_HEADER_COMMENT = .FALSE.
?? IF (.NOT. MODIFYING_HEADER_COMMENT) THEN

One convenient use of conditional compilation is the
ability to write large comments that span across many
lines without requiring each line to start with a "!".
Since conditional compilation specifies blocks of lines
to be skipped over by the compiler, this whole paragraph
can be written and modified without the overhead of
making sure that each line is a Fortran comment.

One can imagine this use of conditional compilation for
header comments preceding Fortran programs, modules and
procedures.

??  ENDIF

--- text output from conditional compilation processing ---
```

```
! EXAMPLE 1 shows a possible shift file for output
?? LOGICAL :: MODIFYING_HEADER_COMMENT = .FALSE.
??  IF (.NOT. MODIFYING_HEADER_COMMENT) THEN
?>
?>One convenient use of conditional compilation is the
?>ability to write large comments that span across many
?>lines without requiring each line to start with a "!".
?>Since conditional compilation specifies blocks of lines
?>to be skipped over by the compiler, this whole paragraph
?>can be written and modified without the overhead of
?>making sure that each line is a Fortran comment.
?>
?>One can imagine this use of conditional compilation for
?>header comments preceding Fortran programs, modules and
?>procedures.
?>
??  ENDIF

--- initial text -----------------------------------------
! EXAMPLE 2 shows a possible short file for output
?? LOGICAL :: DEBUG_PROC_NAME = .FALSE.
?? LOGICAL :: DEBUG_PROC_ARGS = .FALSE.
?? ! Make sure to debug the procedure name if debugging the arguments
?? DEBUG_PROC_NAME = DEBUG_PROC_NAME .OR. DEBUG_PROC_ARGS
SUBROUTINE INTSWAP (LEFT, RIGHT)
  INTEGER, INTENT(INOUT) :: LEFT, RIGHT
  INTEGER :: WRONG
?? IF (DEBUG_PROC_NAME) THEN
   PRINT *, "Entering IntSwap"
?? ENDIF
?? IF (DEBUG_PROC_ARGS) THEN
   PRINT *, " IntSwap(in):left = ", LEFT
   PRINT *, " IntSwap(in):right = ", RIGHT
?? ENDIF

   WRONG = RIGHT
   LEFT = RIGHT
   RIGHT = WRONG

?? IF (DEBUG_PROC_ARGS) THEN
   PRINT *, " IntSwap(out):left = ", LEFT
   PRINT *, " IntSwap(out):right = ", RIGHT
?? ENDIF
?? IF (DEBUG_PROC_NAME) THEN
   PRINT *, "Exiting IntSwap"
?? ENDIF
ENDSUBROUTINE  INTSWAP

--- text output from conditional compilation processing ---
```

```
! EXAMPLE 2 shows a possible short file for output
SUBROUTINE INTSWAP (LEFT, RIGHT)
  INTEGER, INTENT(INOUT) :: LEFT, RIGHT
  INTEGER :: WRONG

   WRONG = RIGHT
   LEFT = RIGHT
   RIGHT = WRONG

ENDSUBROUTINE  INTSWAP
```

REFERENCES:
[1] Words in the first two paragraphs of Section 2
were borrowed from the Sun Microsystems FPP Manual xxx-yyy.