To: SC22WG5 and X3J3 N1243
From: David Epstein X3J3/97-111
Subject: Part III, Conditional Compilation Page 1 of

Dear SC22WG5 and X3J3,

Below is the proposed Part III of the standard regarding conditional compilation (coco). I introduced a conditional compilation facility (CCF) to X3J3 in 1993. At the last seven X3J3 meetings, coco has been discussed at increased depth and intensity along the road of standardization.

At the San Diego SC22WG5 meeting, a vote between a Fortran-like approach and a cpp-like approach was 13-1-10 in favor of the Fortran-like approach. At the Dresden SC22WG5 meeting, a vote between a Fortran-like definition and fpp (a cpp-like definition) was 13-6 in favor of the Fortran-like definition.

I am pleased to provide the below Fortran-like conditional compilation language definition, an updated version of the definition presented in Dresden. This document was reviewed by Dick Hendrickson, Jeanne Martin and John Reid. My thanks to these reviewers. My thanks also to the coco development body and the primary development body for their designing and reviewing efforts during the last two years.

Looking forward to discussing this paper in Las Vegas,
David Epstein
Imagine1, Inc.
david@imagine1.com

==========

---- paper N1243 and X3J3/97-111 follows ----

CONDITIONAL COMPILATION IN FORTRAN

ISO/IEC 1539-3 : 1997

{Auxiliary to ISO/IEC 1539 : 1997 "Programming Language Fortran"}

CONTENTS

1. Introduction

2. Rationale

3. General
   1. Scope

2. Normative References

4. The Conditional Compilation Language Definition

Annex CCA : Examples

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 : 1997, 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 facility.

2. RATIONALE

Programmers often maintain multiple copies of code to vary the features or allow for different operating systems. Keeping several copies of the source code is error prone. A conditional compilation language facility permits the programmer to rely on the processor to select blocks of code based on specified conditions. The additional lines inserted to control this process and all the lines that are not selected are marked to be skipped by the processor during the compilation phase. These lines have no effect on the interpretation of the source during the compilation phase and conceptually are replaced by comment lines.

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 1539-1 : 1997 "Programming Language Fortran"

2. Normative References

The following standard contains provisions which, through reference in this text, constitute provisions of this part of ISO/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 ISO/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.

ISO/IEC 1539-1 : 1997, Information technology--Programming Languages--Fortran.

4. THE CONDITIONAL COMPILATION LANGUAGE DEFINITION

Section CC1 Overview

CC1.1 Conditional compilation and preprocessing

Conditional compilation (coco) is achieved during a preprocessing phase that is controlled by directives in the source text. These directives and any source lines skipped by the conditional compilation have no effect on the interpretation of the source during the compilation phase. The compilation phase, which is described in part 1 of this standard, is referred to as the post-coco processing phase in this part of ISO/IEC 1539.

The execution of coco directives occurs during a phase called the preprocessing phase because conceptually it occurs before the post-coco processing phase described in part 1 of this standard.

Coco execution is a sequence, in time, of actions specified by the coco directives. The actions are performed in the order that the directives appear. The result is conceptually a sequence of lines that is identical to the original sequence except that all coco directive lines are replaced by comment lines or deleted and all lines within a coco FALSE block (CC6.2.2.2) are replaced by comment lines or deleted. The lines that are conceptually replaced by comment lines or deleted are referred to as lines that are marked to be skipped during post-coco processing.

Note CC1.1
Coco execution may be performed by a processor that is separate from the Fortran processor and passes altered source lines to the Fortran processor. Alternatively, it may be integrated into the Fortran processor.
ENDNote CC1.1

CC1.2 Section numbers and syntax rules

In this part of ISO/IEC 1539, sections are numbered CCs, where s is a one or two-digit section number. The notation used in this part of ISO/IEC 1539 is described in Part 1, section 1.6. However, item (4) in Part 1, section 1.6.2 is replace with:

(4) Each syntax rule is given a unique identifying number of the form CCRsnn, where s is a one or two-digit section number and nn is a two-digit sequence number within that section. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. Some rules in Section CC2 and CC3 are more fully described in later sections; in such cases, the section number s is the number of the later section where the rule is repeated.

Section CC2 High level syntax

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

CCR201 *coco-program* **is** *pp-input-item* [*pp-input-item*] ...

CCR202 *pp-input-item* **is** *coco-construct*
or *noncoco-line*

The term <noncoco-line> refers to any line without the characters "??" in character positions 1 and 2.

CCR203 *coco-construct* **is** *coco-type-declaration-directive*
or *coco-executable-construct*

CCR204 *coco-executable-construct* **is** *coco-action-directive*
or *coco-if-construct*

CCR205 *coco-action-directive* **is** *coco-assignment-directive*
or *coco-error-directive*
or *coco-stop-directive*

Section CC3 Constants, source form and including text

CC3.1 Coco constants

CCR301 *coco-constant* **is** *coco-literal-constant*
or *coco-named-constant*

CCR302 *coco-literal-constant* **is** *coco-int-literal-constant*
or *coco-logical-literal-constant*

CCR303 *coco-int-literal-constant* **is** *digit-string*

CCR304 *coco-logical-literal-constant* **is** .TRUE.
or .FALSE.

CCR305 *coco-char-literal-constant* **is** ' [*rep-char*]... '
or " [*rep-char*] ... "

CCR306 *coco-named-constant* **is** *name*

Constraint: coco-named-constant shall have the PARAMETER attribute.

[digit-string is specified in part 1, R402 of this standard.]
[rep-char is specified in part 1, section 4.3.2.1 of this standard.]

[A name is specified in part 1, section 3.2.1 of this standard.]

CC3.2 Coco source form

A coco program is a sequence of one or more lines, organized as coco directives, coco comment lines (CC3.2.1) and noncoco lines. A coco directive is a sequence of one or more coco lines. A coco line is a line with the characters "??" in character positions 1 and 2. These characters are not part of the coco directive. A noncoco line is a line that does not begin in this way.

A coco character context means characters within a coco character literal constant (CCR305).

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

In coco source, each source line may contain from zero to 132 characters.

In coco source, 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 3.2 of this standard.]

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

One or more blanks shall be used to separate adjacent keywords except in the following cases, where blanks are optional:

Adjacent keywords where separating blanks are optional
ELSE IF
END IF

CC3.2.1 Coco commentary

Within a coco directive, the character "!" in any character position initiates a coco comment except when it appears within a coco character context. The coco comment extends to the end of the source line. If the first nonblank character on a coco line after character positions 1 and 2 is an "!", the line is a coco comment line. Coco lines containing only blanks after character positions 1 and 2 or containing no characters after character positions 1 and 2 are also coco comment lines.

Note CC3.1
Examples of coco comment lines are:

```
?? ! SET "SYSTEM" TO "SYSTEM_A"
```

```
??
```
ENDNote CC3.1

CC3.2.2 Coco directive continuation

The character "&" is used to indicate that the current coco directive is continued on the next line. The next line shall be a coco line. Coco comment lines shall not be continued; an "&" in a coco comment has no effect during coco execution. When used for continuation, the "&" is not part of the coco directive. After character positions 1 and 2, no coco line shall contain a single "&" as the only nonblank character or as the only nonblank character before an "!" that initiates a coco comment.

CC3.2.2.1 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 coco comment, the coco directive is continued on the next line. If the first nonblank character after character positions 1 and 2 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 character positions 1 and 2 of the next coco-noncomment line.

If a coco lexical token is split across the end of a line, the first nonblank character after character positions 1 and 2 on the first following coco-noncomment line shall be an "&" immediately followed by the successive characters of the split token.

Note CC3.2
An example of coco-noncharacter context continuation is:

```
?? LOGICAL :: TOO_GOOD&
??&_TO_BE_&
??&TRUE =            &
?? ! These six lines contain 1 coco directive
?? ! and two coco comment lines.
??           .FALSE.
```
ENDNote CC3.2

CC3.2.2.2 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 character positions 1 and 2 on the next line and the coco directive continues with the next character following the "&".

Note CC3.3
An example of coco character context continuation is:

```
?? ERROR "DE&
??           &F&
??            &INE 'SYSTEM' VALUE" ! 3 lines, 1 coco directive
```
ENDNote CC3.3

CC3.2.3 Coco directives

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 be a coco line.

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

Note CC3.4
Examples of coco directives are:

```
 ?? INTEGER, PARAMETER :: SYSTEM_A = 1
 ?? ERROR "SYSTEM_A = ", SYSTEM_A
 ??   IF (.FALSE.) THEN
 ??   ENDIF
```
ENDNote CC3.4

CC3.3 Including source text

Additional text may be incorporated into the source text of a coco program during coco execution. 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 (CC6.2.2) is not expanded. Each coco directive that is a coco-else-if-directive, coco-else-directive, or coco-endif-directive shall appear in the same source text as the matching coco-if-directive; that is, the directives that create a coco IF construct (CC6.2) shall not be split across included source text.

Note CC3.5
An example of erroneous included source text is:

```
??   ELSEIF (version == V8) THEN
      PRINT *, "The matching coco IF is missing"
??   ELSE
      PRINT *, "Wow, the matching coco ENDIF is also missing"
```
ENDNote CC3.5

Section CC4 Coco type declaration directives

Every data object has a type and may have the PARAMETER attribute. The type of a named data object, and possibly the PARAMETER attribute, is specified in a type declaration directive.

CCR401 *coco-type-declaration-directive* **is** *coco-type-spec* [, PARAMETER] :: ≥
≤ *coco-entity-decl-list*

CCR402 *coco-type-spec* **is** INTEGER
or LOGICAL

CCR403 *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 in its coco-type-declaration-directive or any other exectued coco-type-declaration-directive.

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

CCR404 *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 CC4.1
Examples of coco type declaration directives are:

```
?? INTEGER, PARAMETER :: F77 = 1, F90 = 2, F95 = 3, F2000 = 4
?? INTEGER :: FORTRAN_LEVEL = F95
?? LOGICAL :: DEBUG_PROCEDURE_ENTRY_EXIT
```

ENDNote CC4.1

Note CC4.2
Coco type declaration directives are not required to appear before coco executable constructs.
ENDNote CC4.2

Section CC5 Coco variables, expressions and assignment directive

CC5.1 Coco variables

CCR501 *coco-variable* **is** *coco-variable-name*

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

CC5.2 Coco expressions

CC5.2.1 Coco primary

CCR502 *coco-primary* **is** *coco-constant*
or *coco-variable*
or (*coco-expr*)

Constraint: A coco-variable shall be defined (CC8.2) before appearing as a coco-primary.

CC5.2.2 Level-1 expressions

CCR503 *coco-add-operand* **is** [*coco-add-operand mult-op*] *coco-primary*

CCR504 *coco-level-1-expr* **is** [[*coco-level-1-expr*] *add-op*] *coco-add-operand*

[mult-op and add-op are specified in part 1, section 7.1.1.3 of this standard.]

CC5.2.3 Level-2 expressions

CCR505 *coco-level-2-expr* **is** [*coco-level-1-expr rel-op*] *coco-level-1-expr*

[rel-op is specified in part 1, section 7.1.1.5 of this standard.]

CC5.2.4 Level-3 expressions

CCR506 *coco-and-operand* **is** [*not-op*] *coco-level-2-expr*

CCR507 *coco-or-operand* **is** [*coco-or-operand and-op*] *coco-and-operand*

CCR508 *coco-equiv-operand* **is** [*coco-equiv-operand or-op*] *coco-or-operand*

CCR509 *coco-level-3-expr* **is** [*coco-level-3-expr equiv-op*] *coco-equiv-operand*

[not-op, and-op, or-op and equiv-op are specified in part 1, section 7.1.1.6 of this standard.]

CC5.2.5 General form of a coco expression

CCR510 *coco-expr* **is** *coco-level-3-expr*

CC5.3 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.]

Note CC5.2
The following table of operator precedence differs from table 7.8 in part 1 of this standard only by the removal of defined operators, exponentiation and concatenation:

Table CC7.1 Categories of operations and relative precedences

Category of Operation	Operators	Precedence	Term
Numeric	* or /	Highest	mult-op
Numeric	unary + or -	.	add-op
Numeric	binary + or -	.	add-op
Relational	.EQ., .NE., .LT., .LE., .GT., .GE. ==,/=, <, <=, >, >=	.	rel-op
Logical	.NOT.	.	not-op
Logical	.AND.	.	and-op
Logical	.OR.	.	or-op
Logical	.EQV. or .NEQV.	Lowest	equiv-op

ENDNote CC5.2

CCR511 *coco-logical-expr* **is** *coco-expr*

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

CC5.4 Coco initialization expression

CCR512 *coco-initialization-expr* **is** *coco-expr*

Constraint: A coco-initialization-expr shall not have a coco-variable as a coco-primary.

CC5.5 Coco assignment directive

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

CCR513 *coco-assignment-directive* **is** *coco-variable* = *coco-expr*

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

Note CC5.3
Examples of coco assignment directives are:

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

```
?? IS_COMPANY_X_MACHINE = (SYS == SYS_E) .OR. (SYS == SYS_F)
??    PROJECT_LEVEL = FOO_VERSION + LATEST_RELEASE
```
ENDNote CC5.3

Section CC6 Coco execution control and conditional compilation

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

Note CC6.1
A coco program is not required to contain any coco directives.
ENDNote CC6.1

CC6.1 Coco blocks

A coco block is a sequence of coco directives, coco comment lines, INCLUDE lines and noncoco lines that are treated as a unit.

CCR601 *coco-block* **is** [*pp-input-item*] ...

Coco executable constructs may be used to control which coco blocks of a coco program are skipped during post-coco processing.

CC6.2 Coco IF construct

The coco IF construct marks to be skipped during post-coco processing all but at most one of its constituent coco blocks. If there is a coco block that is not marked to be skipped during post-coco processing, it is called the coco TRUE block. The remaining coco blocks are called coco FALSE blocks.

CC6.2.1 Form of the coco IF construct

CCR602 *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

CCR603 *coco-if-then-directive* **is** IF (*coco-logical-expr*) THEN

CCR604 *coco-else-if-directive* **is** ELSE IF (*coco-logical-expr*) THEN

CCR605 *coco-else-directive* **is** ELSE

CCR606 *coco-end-if-directive* **is** END IF

Note CC6.2
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)
?? ENDIF
```

ENDNote CC6.2

CC6.2.2 Execution of an IF construct

At most one of the coco blocks in the coco IF construct is selected as a coco TRUE block. If there is a coco ELSE directive in the construct, exactly one of the coco blocks in the construct will be selected as a coco TRUE block. 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 selected as a coco TRUE block 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 selection of any coco block within the construct as a coco TRUE block. All other blocks of the construct are selected as coco FALSE blocks.

Any coco IF constructs nested within a coco TRUE block are treated similarly. No coco processing occurs for coco IF constructs nested within a coco FALSE block. No coco processing occurs for coco action directives, coco type declaration directives or INCLUDE lines within a coco FALSE block.

Execution of a coco END IF directive has no effect.

Note CC6.3
An example of declaring a coco variable and including source inside a coco IF construct is:

```
?? IF (MACHINE==BIG) THEN
??   INTEGER :: CHIPS = 3
     INCLUDE "STUFF.BIG"
?? ELSE
??   INTEGER :: CHIPS = 1
     INCLUDE "STUFF.ONE"
?? ENDIF
```

ENDNote CC6.3

CC6.2.2.1 Execution of a coco TRUE block

Source lines contained in a coco TRUE block are processed by the coco processor.

CC6.2.2.2 Execution of a coco FALSE block

Source lines contained in a coco FALSE block are marked to be skipped during post-coco processing.

Note CC6.4
If a processor offers an output file as a result of coco execution,
some possible options on the handling of coco directive lines and
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 position to the right,
(4) replace them with a "!" in character position 1, followed by the the characters starting from character position 2 of the original source line, or
(5) replace them with a "!?>" in character positions 1 to 3, followed by the original source line shifted three positions to the right.

All other lines could be copied without modification into the output file.
ENDNote CC6.4

Section CC7 Coco error directive and coco stop directive

CCR701	*coco-error-directive*	**is**	ERROR [*coco-output-item-list*]
CCR702	*coco-output-item*	**is**	*coco-expr*
		or	*coco-char-literal-constant*

Execution of a coco ERROR directive specifies that a programmer detected error has occurred

during coco execution. At the time of execution of a coco ERROR directive, the output-item-list, if any, is available in a processor-dependent manner. Execution of a coco ERROR directive does not halt coco execution.

Note CC7.1
Examples of the coco error directive are:

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

ENDNote CC7.1

CCR703 *coco-stop-directive* **is** STOP

Execution of a coco STOP directive specifies that a severe programmer detected error has occurred during coco execution. Execution of a coco STOP directive halts coco execution. At the time of execution of a coco STOP directive, the fact that coco execution was halted by the execution of a coco STOP directive is available in a processor-dependent manner.

Note CC7.2
When an error has occurred during coco execution, whether or not post-coco processing follows is processor-dependent.
ENDNote CC7.2

Note CC7.3
An example of using the coco ERROR directive and the coco STOP directive for error reporting is:

```
?? IF (MACHINE==BIG) THEN
??   INTEGER :: CHIPS = 3
     INCLUDE "STUFF.BIG"
?? ELSEIF (MACHINE==SMALL) THEN
??   INTEGER :: CHIPS = 1
     INCLUDE "STUFF.ONE"
?? ELSE
??   ERROR "SET MACHINE TO EITHER BIG OR SMALL"
??   ERROR "MACHINE = ", MACHINE
??   ERROR "PREPROCESSING ERROR.  HALTING!     "
??   STOP ! FATAL ERROR.  HALT COCO EXECUTION
?? ENDIF
```

ENDNote CC7.3

Section CC8 Scope and definition of coco variables

CC8.1 Scope of coco variables

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

CC8.2 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 CC10) causes the coco variable to become defined.

Section CC9 Coco program conformance

A coco program is a standard-conforming coco program if it uses only those forms and relationships herein and if the program has an interpretation according to this standard.

A coco processor conforms to this standard if:

(1) It executes any standard-conforming coco program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the coco program.

(2) It contains the capability to detect and report the use within a submitted coco program of an additional form or relationship that is not permitted by the numbered syntax rules or their associated constraints.

(3) It contains the capability to detect and report the use within a submitted coco program of source form not permitted by Section CC3.

(4) It contains the capability to detect and report the use within submitted coco SET options of values and names not permitted by Section CC10.

(5) It contains the capability to detect and report the reason for rejecting a submitted program.

Section CC10 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. A processor may allow many

coco-set-options.

Note CC10.1
The invocation line may be the chosen method of communicating the coco-set-option to the processor. Another method of communicating the coco-set-option to the processor could be a coco input file.
ENDNote CC10.1

The coco-set-option is a method of
(1) documenting the value of a coco PARAMETER;
(2) assigning an initial value to a coco variable (CCR501);
(3) overriding the initial value assigned to a coco variable in a coco initialization expression (CCR512) without editing the coco program.

The coco-set-option consists of one or more pairs of coco-object-names and coco-literal-constants. The requirements on the coco-object-names and coco-literal-constants are as follows:

(1) A coco-object-name specified in a coco-set-option shall be declared in a coco type declaration directive in the coco program. The coco-literal-constant specified in a coco-set-option shall match the type specified in the coco type declaration directive for the coco-object-name.

(2) The type of a coco-object-name specified in a coco-set-option shall match the type of its coco-literal-constant.

(3) If a coco-object-name specified in a coco-set-option has the PARAMETER attribute specified in its coco type declaration directive, the value of its coco-literal-constant shall match the value supplied in its coco type declaration directive.

(4) If a coco-object-name specified in a coco-set-option 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 or be initialized in its coco type declaration directive. An example that shows the enforcement of this requirement is found in Note CC10.4

Note CC10.2
A processor may allow other representations for constants and expressions in a coco-set-option. 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-literal-constant.
ENDNote CC10.2

If a coco-object-name specified in a coco-set-option has the PARAMETER attribute specified in its coco type declaration directive, this coco-object-name and coco-literal-constant pair has no effect. Otherwise, the coco-set-option for this coco-object-name and coco-literal-constant pair acts as if a coco assignment directive (CCR513)--which consists of the specified coco-object-name as the coco variable and the coco-literal-constant as the coco expression

(CCR510)--were placed immediately following the coco type declaration directive that declared the coco-object-name.

Note CC10.3
A coco-object-name that has the PARAMETER attribute is allowed in a coco-set-option for documentation purposes. For example, the following coco-set-option:

```
COCO-SET-OPTION->DOS=1,MAC=2,UNIX=3,SYSTEM=1
```

will override the initial value given to the coco-variable SYSTEM in the following source text:

```
?? INTEGER, PARAMETER :: DOS = 1, MAC = 2, UNIX = 3, OTHER = 4
?? INTEGER :: SYSTEM = UNIX
```

Note that the syntax shown above for the coco-set-option is simply one example syntax, using an arrow ("->"), and equals ("=") and a comma (","), that may be used for the coco-set-option.
ENDNote CC10.3

Note CC10.4
The value assigned to a coco variable with the coco-set-option does not override the value assigned to a coco variable with the coco assignment directive. The above fourth requirement may save the inexperienced programmer from expecting that the coco-set-option:

```
COCO-SET-OPTION(DOS=1:MAC=2:UNIX=3:SYSTEM=1)
```

will override the initial value given to the coco-variable SYSTEM in the following source text:

```
?? INTEGER, PARAMETER :: DOS = 1, MAC = 2, UNIX = 3, OTHER = 4
?? INTEGER :: SYSTEM
?? SYSTEM = UNIX
```

A coco processor shall detect and report that coco-variable SYSTEM is initialized in the coco-set-option, but the specified initial value is not used.

Note that the syntax shown above for the coco-set-option is simply one example syntax, using parentheses ("(",")"), and equals ("=") and a colon (":"), that may be used for the coco-set-option.
ENDNote CC10.4

Annex CCA : 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. The example intentionally has a programming bug in it. 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 execution. The first example shows an output file from conditional compilation execution that handles the lines treated as comments by shifting them three positions to the right and placing the characters "!?>" in character positions one to three.

--- initial text ----------

```
! EXAMPLE 1 shows a possible shift file for output
?? IF (.FALSE.) 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 processor, 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
```

--- possible text output from conditional compilation execution ---

```
! EXAMPLE 1 shows a possible shift file for output
!?>?? IF (.FALSE.) 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 processor, 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
```

--- possible text output from conditional compilation execution ---

```
! 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
```