

Information technology – Programming languages – Fortran

Part 3: Conditional compilation in Fortran

Foreword

[General part to be provided by ISO CS]

This is the third part of ISO/IEC 1539 and has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language. It is an auxiliary standard to ISO/IEC 1539-1: 1997, which defines the latest revision of the primary Fortran language (known informally as Fortran 95).

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

Introduction

Programmers often need to maintain several versions of code to allow for different systems and different applications. Keeping several copies of the source code is error prone. It is far better to maintain a master code from which any of the versions may be selected.

This conditional compilation facility has deliberately been kept very simple. The additional lines inserted to control the process and all the lines that are not selected are omitted from the output or are replaced by comments. Those that are selected are copied to the output completely unchanged.

Examples of the need for such a facility are:

- (1) Parametrized types do not solve all the problems associated with different precisions. Parameterized derived types are not part of Fortran 95.

- (2) A version of a code for complex arithmetic may differ little from the version for real arithmetic.
- (3) The relative efficiency of different constructions may vary from processor to processor. While array syntax may be needed on a massively parallel machine, do loops may be more efficient on a scalar processor. Some may benefit from special comments.
- (4) Versions may be required for different message-passing libraries.
- (5) Additional print statements may be inserted into a program when under development. It may be very helpful have these readily available in case some unexpected results are found in production use.
- (6) Versions may be required with character constants in different languages (internationalization).
- (7) For OPEN statements, the file naming convention varies between systems.

Some of these cases may be addressed within the Fortran code itself by run-time tests, but this will result in a large object code and some run-time overhead. Without conditional compilation, however, most of them can only be solved by maintaining separate versions of the code.

Section 1: General

1.1 Scope

This part of ISO/IEC 1539 defines facilities for conditional compilation in Fortran. This part of ISO/IEC 1539 provides an auxiliary standard for the version of the Fortran language specified by the international standard ISO/IEC 1539-1 and informally known as Fortran 95.

1.2 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.*

Section 2: The conditional compilation language definition

Section CC1: Overview

CC1.1 Conditional compilation

Conditional compilation (coco) is described in this document as an independent process that yields a source program for a Fortran processor. It is expected that implementations will usually integrate the two processes and only optionally provide the coco output code (Fortran source code).

The coco process is controlled by directives that are either omitted from the coco output or are changed to Fortran comments. Coco comments may be introduced to explain the actions and these, too, are either omitted from the coco output or are changed to Fortran comments. Other lines (noncoco lines) are either copied unchanged to the output, omitted, or changed to Fortran comments. There is no requirement that the coco output is a valid Fortran program. The lines of the coco output are in the same order as the corresponding lines of the coco program.

Coco execution is a sequence of actions specified by the coco directives and performed in the order that they appear. The combination of a computing system and the mechanism by which these actions are performed is called a **coco processor** in this part of this standard.

CC1.2 Section numbers and syntax rules

In this part of ISO/IEC 1539, sections are numbered CC s , 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 replaced with:

- (4) Each syntax rule is given a unique identifying number of the form CCR snn , 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.

CC1.3 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 part of this standard.

A coco processor conforms to this part of this standard if:

- (1) It executes any standard-conforming coco program and its SET file 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 and its SET file.
- (2) It contains the capability to detect and report the use within a submitted coco program and its SET file 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 and its SET file of

source form not permitted by Section CC3.

- (4) It contains the capability to detect and report the reason for rejecting a submitted coco program and its SET file.

If a coco program contains a STOP directive that is executed, there is no requirement for the processor to report on any directives that follow the STOP directive.

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-action-construct*

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

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

Note CC2.1

A coco program is not required to contain any coco directives.

Section CC3: Constants, source form and text inclusion

CC3.1 Coco constants

CCR301 <i>coco-constant</i>	is <i>coco-literal-constant</i> or <i>coco-named-constant</i>
CCR302 <i>coco-literal-constant</i>	is <i>coco-int-literal-constant</i> or <i>coco-logical-literal-constant</i>
CCR303 <i>coco-int-literal-constant</i>	is <i>digit</i> [<i>digit</i>] ...
CCR304 <i>coco-logical-literal-constant</i>	is .TRUE. or .FALSE.
CCR305 <i>coco-char-literal</i>	is ' [<i>rep-char</i>] ... ' or " [<i>rep-char</i>] ... "
CCR306 <i>coco-named-constant</i>	is <i>name</i>
CCR307 <i>name</i>	is <i>letter</i> [<i>alphanumeric-character</i>] ...
CCR308 <i>alphanumeric-character</i>	is <i>letter</i> or <i>digit</i> or <i>underscore</i>

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

Constraint: The maximum length of a *name* is 31 characters.

Each *digit* is one of the digits

0 1 2 3 4 5 6 7 8 9

and each *coco-int-literal-constant* is interpreted as a decimal value.

Each *letter* is one of the upper-case letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

or one of the lower-case letters

a b c d e f g h i j k l m n o p q r s t u v w x y z

Each *rep-char* is a character in the processor-dependent character set, which includes the letters, the digits, the underscore, the blank, the currency symbol, and the characters

= + - * / () , . ' : ! " % & ; < > ?

In a coco directive, a lower-case letter is equivalent to the corresponding upper-case letter except in a coco character literal.

The delimiting apostrophes or quotation marks are not part of the value of a coco character literal.

An apostrophe character within a coco character literal delimited by apostrophes is represented by two consecutive apostrophes (without intervening blanks); in this case, the two apostrophes are counted as one character. Similarly, a quotation mark character within a character literal delimited by quotation marks is represented by two consecutive quotation marks (without intervening blanks) and the two quotation marks are counted as one character.

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 **keyword** is word that is part of the syntax of a coco directive. Examples of keywords are IF, INTEGER, LOGICAL, and MESSAGE.

A coco comment may contain any character that may occur in a coco character literal. Outside commentary, a coco directive consists of a sequence of coco lexical tokens. Each token is a keyword, a name, a literal constant, an operator (see Table CC5.2), a comma, a parenthesis, an equals sign, or the separator :: .

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 literal. Blanks may be inserted freely between tokens to improve readability. A sequence of blank characters outside of a coco character literal is equivalent to a single blank character.

Blanks are optional between the following pairs of adjacent coco keywords:

```
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 literal. 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

An example of the use of a coco comment in a coco IF construct (Section CC6.2) is:

```
?? IF (DEVELOPING) THEN
?? ! The following output statement was used when
?? ! developing the code
    WRITE(UNIT=*,FMT=*) 'The value of A is', A
?? END IF
```

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 Continuation other than of a coco character literal

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 continuation in a coco type declaration directive (Section CC4) is:

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

CC3.2.2.2 Continuation of a coco character literal

If a coco character literal 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 the continuation of a coco character literal in a coco message directive (Section CC7) is:

```
?? MESSAGE: "DE&
??      &F&
??      &INE VALID 'SYSTEM' VALUE" ! 3 lines, 1 coco directive
```

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

In the following extract from a coco program, all the coco lines are coco directives. This extract permits a segment of code to be adapted according to whether the compiler is more efficient with

array section syntax or with loop syntax.

```

?? LOGICAL :: USE_SECTIONS
   :
?? IF (USE_SECTIONS) THEN
   A(1:10,1:10) = B(1:10,1:10) + C(1:10,1:10)
?? ELSE
   DO I = 1, 10
     DO J = 1, 10
       A(I,J) = B(I,J) + C(I,J)
     ENDDO
   ENDDO
?? ENENDIF

```

CC3.3 Source text inclusion

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 coco-char-literal
```

An INCLUDE line is not a coco line. An INCLUDE line shall appear on a single source line; it shall be the only nonblank text on this line other than an optional trailing comment.

The effect of the INCLUDE line is as if the referenced source text physically replaced the INCLUDE line during coco processing. An INCLUDE line in a coco FALSE block (CC6.2.2) is not expanded. Included text may contain any source text, including additional INCLUDE lines; such nested INCLUDE lines are similarly replaced with the specified source text. The maximum depth of nesting of any nested INCLUDE lines is processor dependent. Inclusion of the source text referenced by an INCLUDE line shall not, at any level of nesting, result in inclusion of the same source text.

When an INCLUDE line is resolved, the first included line shall not be a coco continuation line and the last included line shall not be a coco line that is continued. 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*.

The interpretation of *coco-char-literal* is processor dependent. An example of a possible valid interpretation is that *coco-char-literal* is the name of a file that contains the source text to be included.

Note CC3.5

The following example shows the use of a coco construct to allow for different naming conventions for INCLUDE files on different systems.

```

?? IF (SYSTEM == DOS) THEN
   INCLUDE "C:\mydir\myfile.txt"
?? ELSEIF (SYSTEM == UNIX) THEN
   INCLUDE "/mydir/myfile.txt"
?? ENENDIF

```

Note CC3.6

The fact that coco processing affects which INCLUDE lines are resolved means that it may be unreasonably difficult to check the coco syntax of a program without actually executing it. This is the reason for all coco directives being treated as executable.

Section CC4: Coco type declaration directives

A **coco data object** is a constant or is a variable. An object with the PARAMETER attribute is a constant and has a value that does not change. An object without the PARAMETER attribute is a variable and is undefined or has a value; during execution of a coco program, the value of a variable may change.

Every coco data object has a type. The type of a named data object, and possibly the PARAMETER attribute, is specified by the execution of a type declaration directive. The initial value of a coco variable is undefined unless it is given an initial value by *coco-initialization* or by a SET file (Section CC9).

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*]

CCR404 *coco-object-name* **is** *name*

Constraint: A *coco-object-name* declared in an executed *coco-type-declaration-directive* shall not be the same as any other *coco-object-name* in its *coco-type-declaration-directive* or any other executed *coco-type-declaration-directive*, except that a *coco-object-name* declared in an executed *coco-type-declaration-directive* in a *coco-program* may be the same as a *coco-object-name* declared in its *coco-set-file*.

Constraint: *coco-object-name* shall be declared in an executed *coco-type-declaration-directive* before appearing in any other executed coco directive.

CCR405 *coco-initialization* **is** = *coco-initialization-expr*

Constraint: In an executed *coco-type-declaration-directive*, the types of the *coco-initialization-expr* and the *coco-type-spec* shall either both be integer or both be logical.

Constraint: In an executed *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
?? INTEGER, PARAMETER :: F95 = 3, F2000 = 4
?? INTEGER :: FORTRAN_LEVEL = F95
?? LOGICAL :: DEBUG_PROCEDURE_ENTRY_EXIT
```

Note CC4.2

Although a *coco-object-name* must be declared in an executed *coco-type-declaration-directive* before appearing in any other executed *coco* directive, there is no requirement that all *coco* type declaration directives appear before other directives.

Section CC5: Coco variables, expressions and assignment directive

CC5.1 Coco variables

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

CCR502 *coco-variable-name* is name

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

CC5.2 Coco expressions**CC5.2.1 Coco primary**

CCR503 *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

CCR504 *coco-add-operand* is [*coco-add-operand mult-op*] *coco-primary*

CCR505 *coco-level-1-expr* is [[*coco-level-1-expr*] *add-op*] *coco-add-operand*

CCR506 *mult-op* is *
 or /

CCR507 *add-op* is +
 or -

CC5.2.3 Level-2 expressions

CCR508 *coco-level-2-expr* is [*coco-level-1-expr rel-op*] *coco-level-1-expr*

CCR509 *rel-op* is .EQ.
 or .NE.
 or .LT.
 or .LE.
 or .GT.
 or .GE.

or ==
or /=
or <
or <=
or >
or >=

CC5.2.4 Level-3 expressions

CCR510 *coco-and-operand* **is** [*not-op*] *coco-level-2-expr*
 CCR511 *coco-or-operand* **is** [*coco-or-operand and-op*] *coco-and-operand*
 CCR512 *coco-equiv-operand* **is** [*coco-equiv-operand or-op*] *coco-or-operand*
 CCR513 *coco-level-3-expr* **is** [*coco-level-3-expr equiv-op*] *coco-equiv-operand*
 CCR514 *not-op* **is** .NOT.
 CCR515 *and-op* **is** .AND.
 CCR516 *or-op* **is** .OR.
 CCR517 *equiv-op* **is** .EQV.
 or .NEQV.

CC5.2.5 General form of a coco expression

CCR518 *coco-expr* **is** *coco-level-3-expr*

Note CC5.1

An example of the use of a coco expression is:

?? IF (VERSION*100+RELEASE > 402) THEN ! Most recent 3 products

CC5.3 Data type and value of a coco expression

The data type of a coco expression is either integer or logical. The data type of the operands of an operator shall be as specified in Table CC5.1 and the type of the result is as specified in Table CC5.1.

Table CC5.1 Types of operands and results

Operator	Type of operands	Type of result
+ , - , * , /	Integer	Integer
.EQ., .NE., .LT., .LE., .GT., .GE.	Integer	Logical
==, /=, <, <=, >, >=	Integer	Logical
.NOT., .AND., .OR., .EQV., .NEQV.	Logical	Logical

CCR519 *coco-logical-expr* is *coco-expr*

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

Note CC5.2

There is a precedence among the operations implied by the general form in CC5.2, which determines the order in which the operands are combined, unless the order is changed by the use of parentheses. This precedence order is summarized in Table CC5.2.

Table CC5.2 Categories of operations and relative precedences

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

The value of a coco expression shall be determined by interpreting each operation as specified in Tables CC5.3, CC5.4, and CC5.5.

Table CC5.3 Interpretation of the numeric operators

Operator	Representing	Use of operator	Interpretation
/	Division	x_1 / x_2	Divide x_1 by x_2
*	Multiplication	$x_1 * x_2$	Multiply x_1 by x_2
–	Subtraction	$x_1 - x_2$	Subtract x_2 from x_1
–	Negation	$-x_2$	Negate x_2
+	Addition	$x_1 + x_2$	Add x_1 and x_2
+	Identity	$+x_2$	Same as x_2

Table CC5.4 Interpretation of relational operators

Operator	Representing	Use of operator	Interpretation
.LT.	Less than	x_1 .LT. x_2	x_1 less than x_2
<	Less than	$x_1 < x_2$	x_1 less than x_2
.LE.	Less than or equal to	x_1 .LE. x_2	x_1 less than or equal to x_2
<=	Less than or equal to	$x_1 \leq x_2$	x_1 less than or equal to x_2
.GT.	Greater than	x_1 .GT. x_2	x_1 greater than x_2
>	Greater than	$x_1 > x_2$	x_1 greater than x_2
.GE.	Greater than or equal to	x_1 .GE. x_2	x_1 greater than or equal to x_2
>=	Greater than or equal to	$x_1 \geq x_2$	x_1 greater than or equal to x_2
.EQ.	Equal to	x_1 .EQ. x_2	x_1 equal to x_2
==	Equal to	$x_1 == x_2$	x_1 equal to x_2
.NE.	Not equal to	x_1 .NE. x_2	x_1 not equal to x_2
/=	Not equal to	$x_1 /= x_2$	x_1 not equal to x_2

Table CC5.5 Result values of logical operators

x_1	x_2	.NOT. x_2	x_1 .AND. x_2	x_1 .OR. x_2	x_1 .EQV. x_2	x_1 .NEQV. x_2
true	true	false	true	true	true	false
true	false	true	false	true	false	true
false	true	false	false	true	false	true
false	false	true	false	false	true	false

CC5.4 Coco initialization expression

Execution of a coco directive containing *coco-initialization* causes the evaluation of the expression *coco-initialization-expr* and the definition of the coco object with the resulting value.

CCR520 *coco-initialization-expr* **is** *coco-expr*

Constraint: Every primary of a *coco-initialization-expr* in the *coco-initialization* for an executed *coco-type-declaration-directive* that specifies the PARAMETER attribute shall be a *coco-constant* or a *coco-initialization-expr* enclosed in parentheses.

Note CC5.3

Note that coco variables with defined values are permitted in a *coco-initialization-expr* for a coco variable.

CCR603 <i>coco-if-then-directive</i>	is IF (<i>coco-logical-expr</i>) THEN
CCR604 <i>coco-else-if-directive</i>	is ELSE IF (<i>coco-logical-expr</i>) THEN
CCR605 <i>coco-else-directive</i>	is ELSE
CCR606 <i>coco-end-if-directive</i>	is END IF

Note CC6.1

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
??     MESSAGE: "Company X does not have a FORTRAN 77 product"
??   ENDIF
?? ELSE
        FUNCTION GET_RABBIT_WEIGHT( ) RESULT(WEIGHT)
?? ENDIF

```

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, the TRUE block is executed, and this completes the execution of the construct. All other blocks of the construct are coco FALSE blocks. 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.

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 CC6.2

An example of declaring a coco variable and accessing a module inside a coco IF construct is:

```
?? IF (MACHINE==BIG) THEN
??   INTEGER :: CHIPS = 3
??   USE MODULE_FOR_BIG
?? ELSE
??   INTEGER :: CHIPS = 1
??   USE MODULE_FOR_SMALL
?? ENDIF
```

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 omitted from the coco output or are converted to Fortran comments and are not otherwise processed by the coco processor. Coco directives in FALSE blocks shall be in accord with the syntax rules, but have no effect and need not satisfy the constraints.

Note CC6.3

Coco directives in a FALSE block are required to be in accord with the syntax rules in order that the end of the block is recognizable when the block has coco IF constructs nested within it.

Section CC7: Coco message and stop directives

CCR701 *coco-message-directive* **is** MESSAGE: [*coco-output-item-list*]

CCR702 *coco-output-item* **is** *coco-expr*
 or *coco-char-literal*

Execution of a coco MESSAGE directive makes the *coco-output-item-list*, if any, available in a processor-dependent manner.

Note CC7.1

Here is an example of the use of a coco message directive:

```
?? IF (SYSTEM == DOS) THEN
??   OPEN(10, "C:\mydir\myfile.txt")
?? ELSEIF (SYSTEM == UNIX) THEN
??   OPEN(10, "/mydir/myfile.txt")
?? ELSE
??   MESSAGE: "system = ", SYSTEM
```

```
?? ENDF
```

CCR703 *coco-stop-directive* is STOP

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

An example of using the coco MESSAGE directive and the coco STOP directive for error reporting is:

```
?? IF (MACHINE==BIG) THEN
??   INTEGER :: CHIPS = 3
??   USE MODULE_FOR_BIG
?? ELSEIF (MACHINE==SMALL) THEN
??   INTEGER :: CHIPS = 1
??   USE MODULE_FOR_SMALL
?? ELSE
??   MESSAGE: "SET MACHINE TO EITHER BIG OR SMALL"
??   MESSAGE: "MACHINE = ", MACHINE
??   MESSAGE: "PREPROCESSING ERROR.  HALTING!      "
??   STOP ! FATAL ERROR.  HALT COCO EXECUTION
?? ENDF
```

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 for a coco variable in a coco type declaration directive causes the variable to become defined.

Note CC9.2

For example, the following coco SET file:

```
?? ALTER: SHIFT-3
?? INTEGER, PARAMETER :: DOS = 1
?? INTEGER :: SYSTEM = DOS
```

will lead to a use of the module DOS_MODULE in the following coco program:

```
?? INTEGER, PARAMETER :: DOS = 1, MAC = 2, UNIX = 3, OTHER = 4
?? INTEGER :: SYSTEM = UNIX
?? IF (SYSTEM==DOS) THEN
    USE DOS_MODULE
?? ELSEIF (SYSTEM==UNIX) THEN
    USE UNIX_MODULE
?? ENDIF
:
```

and the following output:

```
!>?? INTEGER, PARAMETER :: DOS = 1, MAC = 2, UNIX = 3, OTHER = 4
!>?? INTEGER :: SYSTEM = UNIX
!>?? IF (SYSTEM==DOS) THEN
    USE DOS_MODULE
!>?? ELSEIF (SYSTEM==UNIX) THEN
!>    USE UNIX_MODULE
!>?? ENDIF
:
```

Note CC9.3

Provided the coco program has no lines that commence with !>, the following Fortran program can be used to restore the coco program from its SHIFT-3 coco output:

```
PROGRAM RESTORE
  CHARACTER(LEN=135) :: LINE
  DO
    READ(*, '(A)') LINE
    IF (LINE(1:3)=='!>') LINE(1:132) = LINE (4:135)
    WRITE(*, '(A)') LINE(1:LEN_TRIM(LINE))
  END DO
END PROGRAM RESTORE
```

Note CC9.4

The reason for the strict rules for the matching of declarations in the SET file and the coco program is to ensure that the coco program is complete, apart from the initialization of those variables that control which alternative is constructed, and that the SET file accords with it. For instance, if a name is misspelled in the SET file, this will be diagnosed by the system unless the misspelled name happens to be that of another coco object with matching properties.

Note CC9.5

A large program is usually presented to the processor in parts of a manageable size. In this case, each part will need a coco SET file, and must obey the rules of this standard over execution of duplicated coco type declarations. Unless the parts use distinct sets of coco names, merging several parts to be executed as a whole with a single merged SET file will require some editing of the source.

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 that the conditional compilation directives in this example could be automatically generated.

Each example contains a conditional compilation program and a SET file.

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 can be asked to convert these to comments, 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 program units.

```
?? ENDF
```

Execution with a SET file consisting of the single line

```
?? ALTER: SHIFT-3
```

will yield the following output:

```
! 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 can be asked to
!>convert these to comments, 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 program units.
!>
!>?? ENDF
```

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"
?? ENDF
?? IF (DEBUG_PROC_ARGS) THEN
  PRINT *, " IntSwap(in):left = ", LEFT
  PRINT *, " IntSwap(in):right = ", RIGHT
?? ENDF

  WRONG = RIGHT
  LEFT = RIGHT
  RIGHT = WRONG

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

```

Execution with a SET file consisting of the single line

```

?? ALTER: DELETE

```

will yield the following output:

```

! 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

```