

Reference number of working document: **ISO/IEC JTC1/SC22/WG5 Nxxxx**

Date: 2018-08-11

Reference number of document: **ISO/IEC TS 99999:2019(E)**

Committee identification: ISO/IEC JTC1/SC22

Secretariat: ANSI

**Information technology — Programming languages — Fortran —  
Coroutines and Iterators**

*Technologies de l'information — Langages de programmation — Fortran —  
Coroutines et Iterators*

## Contents

0	Introduction . . . . .	i
0.1	History . . . . .	i
0.2	What this technical specification proposes . . . . .	ii
1	General . . . . .	1
1.1	Scope . . . . .	1
1.2	Normative References . . . . .	1
2	Requirements . . . . .	2
2.1	General . . . . .	2
2.2	Summary . . . . .	2
2.3	Coroutine syntax and semantics . . . . .	3
2.4	ITERATOR and ITERATE construct syntax . . . . .	10
2.5	VALUE attribute . . . . .	14
2.6	PRESENT (A) . . . . .	14
3	Examples . . . . .	15
3.1	Quadrature example . . . . .	15
3.2	Iterator for a queue . . . . .	18
3.3	Preserving automatic variables . . . . .	19
4	Required editorial changes to ISO/IEC 1539-1:2018(E) . . . . .	20

## 0 Introduction

### 0.1 History

Fortran has historically been primarily but not exclusively used to solve problems in science and engineering. Solving problems in science and engineering primarily but not exclusively depends upon computational mathematics algorithms. Computational mathematics algorithms frequently require access to software that is provided by the user, to specify the problem. Examples include evaluating integrals, solving differential equations, minimization, and nonlinear parameter estimation.

Software to solve problems in science and engineering also benefits from the application of principles of software engineering, as explained, for example, in **Scientific Software Design: The Object-Oriented Way**, by our colleagues Damian Rouson and Jim Xia, and their coauthor Xiaofeng Xu. An important paradigm related to object-oriented programming is a *container*. Support to develop containers in Fortran is part of the work plan for the next revision. It is important to be able to iterate over the contentx of a container, without exploiting the representation of the container. Examples include traversing a list or tree, or a row or column of a sparse matrix. The procedures of a container that iterate over its contents require access to software that is provided by the user, to perform actions using the members of the container.

In Fortran, access to software that is provided by the user has been provided in three ways.

- The procedure that implements the algorithm invokes a procedure of a specific name,
- The name of the procedure that defines the problem is passed to the procedure that implements the algorithm, or
- The procedure that implements the algorithm returns to the invoker whenever it requires a computation that defines the problem.

The first two of these methods are called *forward communication*; the last is called *reverse communication*.

Forward communication works well in the simple cases where the procedure that implements the algorithm can provide all the information needed by the procedure that defines the problem.

Before Fortran 2003, when additional information was needed, programs exploited methods known to reduce the reliability of programs or increase the cost of their development and maintenance: global data. Fortran 2003 provides type extension, which reduces the problem substantially, but can introduce other problems such as performance penalties caused by pointer components.

Programs developed in Fortran 2003 would probably use type extension to pass additional data to the procedure that defines the problem. Revising existing programs that use reverse communication to use type extension could be prohibitively expensive, especially if rigorous recertification is required, while revising them to use coroutines would be relatively inexpensive.

Reverse communication does not require information necessary to define the problem to be passed through the procedure that implements the algorithm, or require the procedure that defines the problem to access such information by using global data or type extension. There is, however, no structured support for reverse communication in Fortran. In order for the procedure to continue after the calculations that define the problem, it has to know it isn't starting a problem, and how to find its way to continue its process. This usually involves GO TO statements, or transformation of the procedure into an inscrutable "state machine." The state of the computation is usually represented in SAVE variables, which causes the procedure that implements the algorithm not to be thread safe.

A third alternative is mutual recursion with tail calls.

In some problems, it is desirable to preserve the activation record, primarily to avoid re-creating automatic variables. If a procedure is used to solve a large number of related problems, and it requires substantial "working storage," re-creating working storage as automatic variables, or allocating allocatable variables or pointers that do not have the SAVE attribute, can be a significant fraction of the total cost of solving one problem. Alternatives are allocatable variables or pointers with the SAVE attribute, which are not thread safe, and host association, which militates against reuse.

If coroutines had been available during the development of Fortran 2003, defined input/output would not have been needed. Instead, it could have been possible to specify a coroutine to process the input or output list, having an unlimited polymorphic argument to associate with each list item in turn.

## 0.2 What this technical specification proposes

This technical specification proposes two forms of procedures. They both have the property that they have a persistent internal state that is created by their initial invocation. They can be suspended and later resumed, to proceed from the point where they were suspended. The persistent internal state is represented by local entities. Local entities and the state of execution of the procedure are preserved in an *activation record*; local entities do not become undefined when the procedure is suspended.

A *coroutine* can be invoked in the same way as a subroutine. It can be resumed wherever and whenever necessary.

An *iterator* can be invoked in the same way as a function, but only in a new ITERATE construct. When a function is invoked, it returns a value. One would expect that when an iterator is resumed, it would return a value, but there is only one way to indicate which instance of the iterator is to be resumed, to provide a value: a looping construct. A Wikipedia article describes the iterator as

... one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The term "coroutine" first appeared in documentation of the language Simula. Tasks and protected variables in Ada are similar to coroutines.

Coroutines are supported directly in the following languages:

- Aikido
- AngelScript
- BCPL
- Pascal (Borland Turbo Pascal 7.0 with uThreads module)
- BETA
- BLISS
- C#
- ChuckK
- CLU
- D
- Dynamic C
- Erlang
- F#
- Factor
- GameMonkey Script
- GDScript (Godot's scripting language)
- Go
- Haskell
- High Level Assembly
- Icon
- Io
- JavaScript (since 1.7, standardized in ECMAScript 6) ECMAScript 2017 also includes await support.
- Julia
- Kotlin (since 1.1)
- Limbo
- Lua
- Lucid
- $\mu$ C++
- MiniD
- Modula-2
- Nemerle
- Perl 5 (using the Coro module)
- Perl 6
- PHP (with HipHop, native since PHP 5.5)
- Picolisp
- Prolog
- Python (since 2.5, with improved support since 3.3 and with explicit syntax since 3.5)
- Ruby
- Sather
- Scheme
- Self
- Simula 67
- Smalltalk
- Squirrel
- Stackless Python
- SuperCollider
- Tcl (since 8.6)
- urbiscript

Iterators are supported directly in the following programming languages:

- C++
- C# and other .NET languages
- Java
- JavaScript
- Matlab
- PHP
- Python
- Ruby
- Rust
- Scala

All of the alternatives that presently exist in Fortran, described in the previous subclause, require to invoke and return from a procedure to respond to a need to execute “user” code. In contrast, when a coroutine or iterator is suspended its activation record is not destroyed, and when it is resumed its activation record is not reconstructed. Therefore, suspending and resuming a coroutine or iterator is generally more efficient than the alternatives.

# Information technology – Programming Languages – Fortran

## Technical Specification: Coroutines and iterators

### 1 General

#### 1 1.1 Scope

2 This technical specification specifies extensions to the programming language Fortran. The Fortran  
3 language is specified by International Standard ISO/IEC 1539-1:2018(E). The extensions are varieties  
4 of procedures known as *coroutines* and *iterators*. They have the property that an instance of one can  
5 be suspended, and later resumed to continue execution from the point where it was suspended. Local  
6 entities and the state of execution of the procedure are preserved in an *activation record*, and do not  
7 become undefined when the procedure is suspended. The invoking scope retains the activation record,  
8 and can have as many separate activation records for each procedure as necessary.

9 Clause 2 of this technical specification contains a general and informal but precise description of the  
10 extended functionalities. Clause 3 contains several illustrative examples. Clause 4 contains detailed  
11 instructions for editorial changes to ISO/IEC 1539-1:2018(E).

#### 12 1.2 Normative References

13 The following referenced documents are indispensable for the application of this document. For dated  
14 references, only the edition cited applies. For undated references, the latest edition of the referenced  
15 document (including any amendments) applies.

16 ISO/IEC 1539-1:2018(E) : *Information technology – Programming Languages – Fortran; Part 1: Base*  
17 *Language*

## 2 Requirements

### 2.1 General

The following subclauses contain a general description of the extensions to the syntax and semantics of the Fortran programming language to provide coroutines and iterators.

### 2.2 Summary

#### 2.2.1 What is provided

This technical specification defines new forms of procedures, called *coroutines* and *iterators*, an instance of which can be suspended and later resumed to continue execution from the point where it was suspended. Local entities and the state of execution of the procedure are preserved in an *activation record*, and do not become undefined when the procedure is suspended. The invoking scope retains the activation record, and can have any number of activation records. There is presently nothing comparable in Fortran, but coroutines and iterators have been provided by numerous other programming languages.

This technical specification describes statements to define coroutines and iterators, statements to suspend, resume, and terminate coroutines, an inquiry function to determine whether a coroutine is suspended, and a looping control construct that invokes an iterator.

#### 2.2.2 Coroutines

A coroutine is a procedure that is invoked similarly to the way a subroutine is invoked. Unlike a subroutine, it can be suspended, and later resumed to continue execution from the point where it was suspended. Local entities and the state of execution of a coroutine are preserved in an *activation record*, and do not become undefined when it is suspended. Each invocation of a coroutine creates a new instance, independently of whether an instance is already in a state of execution. The invoking scope retains the activation record, and can have as many activation records as necessary. A coroutine can be pure, but it cannot be elemental. A coroutine identifier shall have explicit interface where it is invoked or resumed.

#### 2.2.3 Iterators

An iterator is a procedure that produces a result value, as does a function subprogram. It is intended to be used as an abstraction to produce the elements of a data structure, one at a time. It can be invoked or resumed only within the ITERATE statement of an ITERATE construct. Local entities and the state of execution of an iterator are preserved in an *activation record*, and do not become undefined when it is suspended. A different instance exists for each ITERATE construct. Nested ITERATE constructs can use the same iterator. An iterator identifier shall have explicit interface where it appears in an ITERATE construct.

#### 2.2.4 ITERATE construct

The ITERATE construct uses an iterator to process the elements of a data structure, one at a time. When execution of the construct commences, the iterator is invoked and a new instance of it is created. Therefore, an ITERATE construct within another ITERATE construct can use the same iterator. Each time the iterator suspends it provides a value, or a pointer associated with a value, and the body of the construct is executed. After the construct body is executed, the iterator is resumed at the first executable construct after the SUSPEND statement that suspended execution of the iterator. Execution of the ITERATE construct completes, the activation record of the instance is destroyed, and the instance of the iterator ceases to exist when

- the iterator executes a RETURN or END statement,



- 2 *[internal-subprogram-part]*  
3 *end-coroutine-stmt*
- 4 R1537b *coroutine-stmt* is *[prefix]* COROUTINE *coroutine-name* ■  
5 ■ [ ( [ *dummy-arg-name-list* ] ) ]
- 6 R1537c *end-coroutine-stmt* is END COROUTINE [ *coroutine-name* ]
- 7 C1251a (R1537b) Neither *declaration-type-spec* nor ELEMENTAL shall appear in *prefix*.
- 8 C1251b (R1537a) An internal coroutine subprogram shall not contain an *internal-subprogram-part*.
- 9 C1251c (R1537c) If a *coroutine-name* appears in the *end-coroutine-stmt* it shall be identical to the  
10 *coroutine-name* in the *coroutine-stmt*.

**NOTE 2.2**

When a coroutine is invoked by a CALL statement, a new instance of its activation record is created, regardless whether it is invoked recursively. Therefore, whether RECURSIVE or NON-RECURSIVE appears in the prefix is irrelevant.

**Unresolved Technical Issue Recursive Coroutine**

The appearance of RECURSIVE or NON-RECURSIVE in the prefix could be prohibited instead of ignored.

**2.3.2 Coroutine interface body**

12 The interface of a coroutine can be declared by an interface body.

- 13 R1505 *interface-body* is ...  
14 or *coroutine-stmt* ■  
15 ■ [ *specification-part* ]  
16 ■ *end-coroutine-stmt*

**2.3.3 Coroutine reference****2.3.3.1 General**

19 An identifier of a coroutine shall have explicit interface where it is invoked or resumed.

**2.3.3.2 Coroutine instance variables**

21 A *coroutine instance variable* represents an instance of a coroutine's activation record.

22 Within a scoping unit, if the *coroutine-name* of a coroutine, or a name associated with one by use or host  
23 association, appears as the *procedure-designator* in a CALL statement, or as an actual argument that  
24 corresponds to a dummy argument that does not have the VALUE attribute, a local instance variable  
25 identified by that *procedure-designator* exists and has a scope of that inclusive scope.

26 A coroutine procedure pointer, or a dummy procedure that has a coroutine interface, is an instance  
27 variable.

28 If an object is of a type that has a type-bound coroutine, that object contains an instance variable for  
29 that coroutine, identified by that binding.

1 An instance variable is not a local variable if it is

- 2 • a dummy coroutine without the VALUE attribute,
- 3 • accessed by use or host association, or
- 4 • represented within an object of derived type that has a binding to the coroutine, and the object is
- 5 not a local variable.

6 Otherwise, it is a local variable.

7 An instance variable is an object of a private derived-type with private components. It identifies a  
 8 coroutine and represents an instance of its activation record. The types of different instance variables  
 9 are not necessarily the same, but they all have a private allocatable activation record component, and  
 10 a private procedure pointer component that identifies the coroutine. If it is a dummy procedure with  
 11 a coroutine interface, the association of the procedure pointer component is that of the corresponding  
 12 actual argument. Otherwise, if it is a coroutine pointer, the procedure pointer component has default  
 13 initialization of NULL(). Otherwise, the procedure pointer component is associated with the coroutine  
 14 specified by the *procedure-designator*.

### 15 2.3.3.3 Coroutine activation records

16 An instance variable has a private allocatable component that represents the coroutine's activation  
 17 record. It is allocated if and only if the instance of the coroutine is active. The activation record  
 18 represents the state of execution of the instance, and its unsaved local variables. A local variable of a  
 19 coroutine that has the SAVE attribute is shared by all instances; it is not part of an activation record.  
 20 Variables accessed by use and host association are not part of an activation record.

21 The activation record component of a local instance variable is initially deallocated, even if it is a dummy  
 22 coroutine with the VALUE attribute. A local instance variable does not initially represent an active  
 23 instance when the procedure is invoked, even if it is a dummy coroutine with the VALUE attribute and  
 24 the corresponding actual argument represents an active instance. Unlike a dummy data object with the  
 25 VALUE attribute, the allocation status, and value if any, of the allocatable component that represents  
 26 its activation record, is not copied from the actual argument that corresponds to a dummy coroutine  
 27 with the VALUE attribute.

#### NOTE 2.3

Because the activation record component of an instance variable is allocatable, it is or becomes deallocated, and the instance it represents is terminated, under the same conditions that an allocatable component of a derived-type object is or becomes deallocated.

28 An instance of a coroutine is accessible if and only if is represented by an accessible instance variable  
 29 that represents an active instance.

### 30 2.3.3.4 Creating an instance of a coroutine

31 When a coroutine is invoked by a CALL statement, the following occur in the order specified:

- 32 1. Arguments associations are established.
- 33 2. An instance of the coroutine is created.
- 34 3. The activation record component of its instance variable is allocated as if by an ALLOCATE
- 35 statement.
- 36 4. Expressions within its specification part are evaluated and its specification part is elaborated,
- 1 creating local variables of the instance that do not have the SAVE attribute.

2 When the instance executes a RETURN, END, or SUSPEND statement, or completes execution of  
 3 the last executable construct of the coroutine's *execution-part*, execution of the CALL statement is  
 4 completed.

#### 5 **2.3.3.5 Suspending a coroutine instance**

6 When an instance of a coroutine executes a SUSPEND statement, execution of the instance of the  
 7 coroutine is suspended and the execution sequence continues by executing the executable construct  
 8 following the CALL statement that invoked that instance of that coroutine, or the RESUME statement  
 9 that resumed execution of that instance of that coroutine, whichever occurred most recently. The  
 10 activation record component of its instance variable is not deallocated.

#### 11 **2.3.3.6 Resuming a coroutine instance**

12 An instance of a coroutine is resumed by executing a RESUME statement (2.2.6) with a designator  
 13 that designates its instance variable. When it is resumed, argument associations are re-established and  
 14 control is transferred to the first executable construct after the SUSPEND statement that most recently  
 15 suspended execution of the instance of the coroutine represented by the instance variable used to resume  
 16 it. Its activation record is not re-created. Expressions in the specification part are not re-evaluated,  
 17 and the specification part is not elaborated again. Therefore, local variables of the instance, including  
 18 automatic variables, retain the same bounds, length parameter values, definition status, and values if  
 19 any, that they had when the instance was suspended.

#### **NOTE 2.4**

Because argument associations are re-established, dummy arguments might have different extents,  
 length parameter values, allocation status, pointer association status, or values (if any).

20 If a coroutine is invoked before a DO CONCURRENT construct begins execution, the same instance of it  
 21 shall not be resumed during more than one iteration of that execution of that construct. A coroutine shall  
 22 not be invoked using the same instance variable during more than one iteration of a DO CONCURRENT  
 23 construct. If a coroutine is invoked during an iteration of a DO CONCURRENT construct, that instance  
 24 of it shall be terminated during that iteration, and it shall not be terminated or resumed during a  
 25 different iteration of that execution of that construct.

26 If a coroutine is invoked from within a CRITICAL construct or from within a procedure invoked during  
 27 execution of a CRITICAL construct, the same instance of it shall be terminated during that execution  
 28 of that construct, and it shall not be resumed after that execution of that construct completes. If a  
 29 coroutine is invoked before execution of a CRITICAL construct begins, the same instance of it shall not  
 30 be resumed from within that execution of that CRITICAL construct or from within a procedure invoked  
 31 during that execution of that CRITICAL construct.

#### **Unresolved Technical Issue Critical**

The restrictions concerning critical sections might not be necessary or useful.

32 An instance of a coroutine that has ceased to exist shall not be resumed.

#### 33 **2.3.3.7 Terminating a coroutine instance**

34 An instance of a coroutine is terminated, and the activation record component of the instance variable  
 35 used to terminate the instance becomes deallocated, when

- 36 • a RETURN or END statement is executed by the instance of the coroutine,
- 1 • the last executable construct of the *execution-part* of the coroutine completes execution,

- 2 • a TERMINATE statement that designates the instance variable is executed,
- 3 • a CALL statement invokes the coroutine using its instance variable,
- 4 • the instance variable is an unsaved local variable of a procedure that is not a coroutine, and
- 5 execution of the procedure in which it is a local variable is terminated,
- 6 • the instance variable is an unsaved local variable of a coroutine and the instance of that coroutine
- 7 is terminated,
- 8 • the instance variable is the *proc-pointer-object* in a pointer assignment statement that is executed,
- 9 • the instance variable is a *proc-pointer-object* in a NULLIFY statement that is executed, or
- 10 • the instance variable corresponds to a dummy procedure pointer that has INTENT(OUT) and the
- 11 CALL statement or function reference is executed.

### Unresolved Technical Issue Duplicate

Executing a CALL statement that references a coroutine using a designator with which an instance is associated could alternatively be defined to be an error.

#### 12 2.3.4 Coroutine procedure pointers

13 A coroutine procedure pointer is an instance variable. The ASSOCIATED intrinsic function inquires  
 14 whether the procedure pointer component is associated with a coroutine. The SUSPENDED intrinsic  
 15 function inquires whether its activation record component is allocated, that is, whether it represents an  
 16 instance of a coroutine that has not terminated.

17 A coroutine procedure pointer shall not be a coindexed object or a subobject of a coindexed object.

#### 18 2.3.5 SUSPEND statement

19 Execution of a suspend statement within a coroutine suspends execution of an instance of that coroutine  
 20 (2.3.3.5).

21 Execution of a suspend statement within an iterator suspends execution of an instance of that iterator  
 22 (2.4.4).

23 R1542a *suspend-stmt* is SUSPEND

24 C1276a (R1241a) A *suspend-stmt* shall appear only within the inclusive scope of a coroutine or iterator.

#### 25 2.3.6 RESUME statement

26 Execution of a RESUME statement causes execution of an instance of a coroutine to be resumed (2.3.3.6).

27 R1525a *resume-stmt* is RESUME *procedure-designator* [ ( [ *actual-arg-spec-list* ] ) ]

28 C1537b (R1525a) The *procedure-designator* shall designate a coroutine instance variable.

29 C1537b (R1525a) The *procedure-designator* shall not be a coindexed object or a subobject of a coindexed  
 30 object.

31 The *procedure-designator* shall designate a suspended instance of a coroutine.

32 When a RESUME statement is executed, argument associations are re-established, but expressions in the  
 33 specification part of the coroutine are not re-evaluated and the specification part is not elaborated again.  
 34 Therefore, local variables, including automatic variables, of the instance retain the same bounds, length  
 1 parameter values, definition status, and values if any, that they had when the instance was suspended.

**NOTE 2.5**

Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

2 When the instance of the coroutine that is resumed by execution of a RESUME statement executes a  
 3 SUSPEND, RETURN, or END statement, execution of the RESUME statement is completed.

4 **2.3.7 SUSPENDED ( PROC )**

5 **Description.** Whether a coroutine is suspended.

6 **Class.** Transformational function.

7 **Argument.** PROC shall be a *procedure-designator* that designates a coroutine instance variable. It  
 8 shall not be a coindexed object or a subobject of a coindexed object.

9 **Result Characteristics.** Default logical.

10 **Result Value.** The result has the value true if and only if the activation record component of PROC  
 11 is allocated.

12 **2.3.8 The TERMINATE statement**

13 Execution of a TERMINATE statement causes an instance of a coroutine to be terminated (2.3.3.7).

14 R1525b *terminate-stmt* is TERMINATE ( *instance-variable* [ *terminate-opt-list* ]

15 R1525c *terminate-opt* is STAT = *stat-variable*  
 16 or ERRMSG = *errmsg-variable*

17 R1525d *instance-variable* is *procedure-name*  
 18 or *proc-pointer-object*  
 19 or *proc-component-ref*

20 C1537c (R1525c) The *instance-variable* shall designate a coroutine instance variable.

21 C1537d (R1525c) The *instance-variable* shall not be a subobject of a coindexed object.

22 The *procedure-designator* shall designate an instance variable of a coroutine, and its activation record  
 23 component shall be allocated. A coroutine instance shall not terminate itself by executing a TERMI-  
 24 NATE statement.

25 When a TERMINATE statement is executed, the activation record component of the instance variable  
 26 becomes deallocated, as if by execution of a DEALLOCATE statement. The effects of STAT= and  
 27 ERRMSG= specifiers include the same effects as in a DEALLOCATE statement. In addition, if a  
 28 coroutine instance terminates itself by executing a TERMINATE statement, a processor-dependent  
 29 nonzero value shall be assigned to *stat-variable*, and that value shall be different from any value that  
 30 might be assigned by a DEALLOCATE statement. If the activation record component of the instance  
 31 variable is not allocated or a coroutine instance terminates itself by executing a TERMINATE statement,  
 32 and STAT= does not appear, an error condition exists.

33 **2.3.9 Coroutine to process input or output statement**

34 The READ and WRITE statements are revised to include an optional PROCESSOR=*coroutine-name*  
 1 specifier. The PROCESSOR=specifier shall not appear in a statement that specifies namelist or list-

2 directed formatting, or that has both ASYNCHRONOUS='YES' and SIZE= specifiers. The specified  
3 coroutine shall have the following interface:

```
4  coroutine coroutine-name ( unit, item, format, iostat, iomsg, size )
5      integer, intent(in) :: unit
6      class(*), INTENT(intent-spec), optional :: item(..)
7      character(*), intent(in), optional :: format
8      integer, intent(out), optional :: iostat
9      character(*), intent(inout), optional :: iomsg
10     integer, intent(out), optional :: size
11 end coroutine coroutine-name
```

### Unresolved Technical Issue Item argument

Instead of requiring the *item* argument to be unlimited polymorphic, it could be required to be type compatible with every data transfer list item.

12 If the statement is a READ statement, the *intent-spec* of its *item* argument shall be OUT. If it is a  
13 WRITE statement, the *intent-spec* of its *item* argument shall be IN.

14 When a data transfer statement with a PROCESSOR=*coroutine-name* specifier is executed, the specified  
15 coroutine is invoked even if there is no first list item. The processor resumes the coroutine if and only  
16 if there is another list item, to process each list item. The *item* argument is present if and only if there  
17 is another list item.

18 The *format* argument is present if and only if the data transfer statement is a formatted data transfer  
19 statement. The value of the *format* argument begins and ends with parentheses, and corresponds to  
20 the *item* argument, as if the item and format were processed without using the coroutine. It might  
21 contain edit descriptors even if the *item* argument is not present; for example, it might contain control  
22 or character string edit descriptors.

23 If a list item is of a derived type that has a pointer or allocatable direct component, and the data transfer  
24 statement is a formatted data transfer statement, the corresponding format item shall be a DT edit  
25 descriptor. If the corresponding format item is a DT edit descriptor, or the list item is of a derived type  
26 that has a pointer or allocatable direct component, the list item is associated with the *item* argument.  
27 Otherwise, the list item is expanded as specified in subclause 12.6.3 of ISO/IEC 1539-1:2018(E)

28 The *iostat* or *iomsg* argument is present if and only if the corresponding specifier appears in the data  
29 transfer statement; it is associated with the specified entity.

30 If an error, end-of-file, or end-of-record condition occurs, and the *iostat* argument is present, the  
31 coroutine shall assign the appropriate value to that argument, as specified in subclause 12.11 of ISO/IEC  
32 1539-1:2018(E). If the *iomsg* argument is present, a value may be assigned to it. If the *iostat* argument  
33 is absent, the coroutine shall return rather than suspending. If no error occurs and the *iostat* argument  
34 is present, the value zero shall be assigned to it. A value shall not be assigned to the *iomsg* argument  
35 unless a nonzero value is or would be assigned to the *iostat* argument. If no error, end-of-file, or  
36 end-of-record condition occurs the coroutine shall suspend.

37 The *size* argument is present if and only if the data transfer statement is a READ statement in which  
38 a SIZE= specifier appears. If it is present, a value shall be assigned to it, to specify the number of  
39 characters transferred from the file.

40 If the data transfer statement is a formatted data transfer statement, data transfer statements other  
1 than those that specify an internal file that are executed while the coroutine is active are processed as

2 if ADVANCE='NO' were specified, even if ADVANCE='YES' is specified in the statement that caused  
 3 the coroutine to be executed.

4 After processing the last list item, or if the coroutine assigns a nonzero value to the *iostat* argument,  
 5 the processor terminates the coroutine. Because the coroutine might use asynchronous data transfer  
 6 statements, after terminating the coroutine, the processor performs a wait operation if the statement  
 7 that caused the coroutine to be executed is not an asynchronous data transfer statement.

8 If the coroutine terminates instead of suspending, an error condition occurs in the statement that caused  
 9 the coroutine to be executed.

## 10 2.4 ITERATOR and ITERATE construct syntax

### 11 2.4.1 ITERATOR syntax

12 An iterator is a subprogram. It can be an external subprogram, a module subprogram, an internal  
 13 subprogram, or a separate module procedure. It can be bound to a type. It can be pure, but it cannot  
 14 be elemental.

15 R1532a *iterator-subprogram*            **is** *iterator-stmt*  
 16    [ *specification-part* ]  
 17    [ *execution-part* ]  
 18    [ *internal-subprogram-part* ]  
 19    *end-iterator-stmt*

20 R1532b *iterator-stmt*                   **is** [ *prefix* ] ITERATOR *iterator-name* ■  
 21    ■ ( [ *dummy-arg-name-list* ] ) [ RESULT ( *result-name* ) ]

22 R1532c *end-iterator-stmt*            **is** END ITERATOR [ *iterator-name* ]

23 C1564a (R1532b) If RESULT appears, *result-name* shall not be the same as *iterator-name*.

24 C1564b (R1532b) If RESULT appears, the *iterator-name* shall not appear in any specification statements  
 25 in the scoping unit of the iterator subprogram.

26 C1564c (R1532b) ELEMENTAL shall not appear in *prefix*.

27 C1564d (R1532a) An internal iterator subprogram shall not contain an *internal-subprogram-part*.

28 C1564e (R1532c) If an *iterator-name* appears in the *end-iterator-stmt* it shall be identical to the *iterator-*  
 29 *name* in the *iterator-stmt*.

1 The result variable name of an iterator is the *result-name* if one appears; otherwise it is the *iterator-name*.

**NOTE 2.6**

When an iterator is invoked by an ITERATE construct, a new activation record is created, even if it is invoked recursively. Therefore, whether RECURSIVE or NON\_RECURSIVE appears in the prefix is irrelevant.

**Unresolved Technical Issue Recursive Iterator**

The appearance of RECURSIVE or NON\_RECURSIVE in the prefix could be prohibited instead of ignored.

## 2.4.2 Iterator interface body

An iterator interface can be declared by an interface body.

```
R1505 interface-body          is ...
                                     or iterator-stmt
                                     [ specification-part ]
                                     end-iterator-stmt
```

## 2.4.3 ITERATE construct syntax

An ITERATE construct is used to iterate over the elements of a data structure, which elements are provided by invoking and resuming an iterator.

```
R1139a iterate-construct      is iterate-stmt
                                     block
                                     end-iterate-stmt
```

```
R1139b iterate-stmt          is [ iterate-construct-name: ] ITERATE [ CONCURRENT ] ■
                                     ■ ( iteration-control )
```

```
R1139c iteration-control      is variable = iterator-reference
                                     or data-pointer-object => iterator-reference
                                     or declaration-type-spec [ , ALLOCATABLE ] :: ■
                                     ■ variable-name [ ( array-spec ) ] = iterator-reference
                                     or declaration-type-spec [ , POINTER ] :: ■
                                     ■ variable-name [ ( array-spec ) ] => iterator-reference
```

```
R1139d end-iterate-stmt      is END ITERATE [ iterate-construct-name ]
```

C1143a (R1139a) If the *iterate-stmt* of an *iterate-construct* specifies an *iterate-construct-name*, the corresponding *end-iterate-stmt* shall specify the same *iterate-construct-name*. If the *iterate-stmt* of an *iterate-construct* does not specify an *iterate-construct-name*, the corresponding *end-iterate-stmt* shall not specify an *iterate-construct-name*.

C1143b (R1139c) If = appears and ALLOCATABLE does not appear, *array-spec* shall specify explicit shape. If ALLOCATABLE appears or => appears, *array-spec* shall specify deferred shape.

C1143c (R1139c) If = appears, the type, type parameters, and rank of *variable* or *variable-name* shall conform to those of the result of *iterator-reference* in the same way that those of *variable* and *expr* are required to conform in an intrinsic *assignment-stmt*.

C1143d (R1139c) If => appears, the type, type parameters, and rank of *data-pointer-object* or *variable-name* shall conform to those of the result of *iterator-reference* in the same way that those of *data-pointer-object* and *data-target* are required to conform in a *pointer-assignment-stmt*.

C1143e (R1139c) The *variable* shall not be a coindexed object or a subobject of a coindexed object.

C1143f (R1139c) If *declaration-type-spec* appears it shall specify the same declared type and kind type parameters as the result of *iterator-reference*, and shall not specify any assumed length type parameters.

C1143g (R1139c) If => appears, either *declaration-type-spec* shall appear, or *data-pointer-object* shall

2 have the POINTER attribute.

3 C1143h (R1139c) If CONCURRENT appears, *declaration-type-spec* shall appear.

4 C1143j (R1139a) If CONCURRENT appears, the construct shall not contain an EXIT statement that  
5 belongs to the construct or an outer construct, a CYCLE statement that belongs to an outer  
6 construct, or a branching statement that has a branch target that is not the END ITERATE  
7 statement or a statement within the block of the construct.

8 R1520a *iterator-reference* is *procedure-designator* ( [ *actual-arg-spec-list* ] )

9 C1524a (R1520a) The *procedure-designator* shall designate an iterator.

10 C1524b (R1520a) The *procedure-designator* shall not be a coindexed object or a subobject of a coindexed  
11 object.

12 If *declaration-type-spec* appears, it specifies the type and type parameter values of the *variable-name*,  
13 and *variable-name* is a construct entity of the ITERATE construct. If => also appears it has the pointer  
14 attribute, and this may be confirmed by the appearance of POINTER. If = appears the *variable-name*  
15 may be declared to have the ALLOCATABLE attribute. It does not have any additional attributes.

#### 16 2.4.4 ITERATE construct and iterator execution semantics

17 When the *iterate-stmt* of an ITERATE construct is executed the construct becomes active. If the  
18 *procedure-designator* in *iterator-reference* is a pointer, it shall be associated with an iterator. The values  
19 of the nondeferred length parameters of *variable*, *variable-name*, or *data-pointer-object* shall be the same  
20 as corresponding parameters of the result of *iterator-reference*.

21 When an *iterate-stmt* is executed, the following occur in the specified order:

- 22 1. Argument associations are established.
- 23 2. An instance of the iterator is associated with the *iterate-stmt*; it is not represented by an instance  
24 variable.
- 25 3. The iterator is invoked.
- 26 4. An activation record is created for the instance by evaluating expressions within the specification  
27 part of the iterator and elaborating the specification part.
- 28 5. Execution of the iterator begins with its first executable construct.

29 While the construct is active, the following occur in the specified order:

- 30 1. If = appears the iterator result value is assigned to *variable* or *variable-name* as if by an assignment  
31 statement; if => appears the result value is assigned to *data-pointer-object* or *variable-name* as if  
32 by pointer assignment.

#### NOTE 2.7

Because the assignment of the result of *iterator-reference* to *variable* or *variable-name* is as if by an assignment statement, it might cause finalization of *variable*, invocation of defined assignment, or allocation or reallocation of an allocatable *variable*.

- 1 2. The *block* of the ITERATE construct is executed.

- 2 3. The instance of the iterator is resumed by re-establishing argument associations and transferring  
 3 control to the first executable construct after the SUSPEND statement whose execution suspended  
 4 its execution. Expressions in the specification part are not re-evaluated and the specification part  
 5 is not elaborated again. Therefore, local variables, including automatic variables, of the instance  
 6 retain the same bounds, length parameter values, definition status, and values if any, that they  
 7 had when the instance was suspended.

**NOTE 2.8**

Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

8 Invoking or resuming the iterator, assigning or associating its result, and executing the *block*, is an  
 9 iteration. If *declaration-type-spec* appears, each iteration has a different instance of *variable-name*.

10 An iterator terminates when it executes a RETURN, END, or SUSPEND statement, or completes  
 11 execution of the final executable construct of its *execution-part*.

12 If CONCURRENT appears, the processor may invoke and resume the iterator, and assign its value, in  
 13 the sequence of execution that began execution of the construct, and then execute each corresponding  
 14 block in a separate sequence of execution. Alternatively, it may invoke and resume the iterator, assign  
 15 its value, and execute the corresponding block, in a separate sequence of execution for each iteration.  
 16 The processor shall ensure that when the iterator is invoked or resumed, no other iteration of the same  
 17 execution of the construct resumes the construct's instance of the iterator until it terminates. In either  
 18 case, the separate sequences of execution may be executed in any order, or concurrently.

**NOTE 2.9**

If the processor chooses to invoke or resume the iterator, assign values to instances of *variable-name*, and execute corresponding blocks, independently within separate sequences of execution, instead of invoking and resuming the iterator within the sequence of execution that initiated the construct, this effectively requires an iterator to be a monitor procedure, or that invoking or resuming it is protected as if by a critical section.

19 Because the *variable-name* is a construct entity, if it is allocatable, it is not allocated before the iterator  
 20 is invoked, and it becomes deallocated at the end of each iteration. The *variable* is not a construct  
 21 entity.

22 When the iterator terminates, a value is not assigned to *variable* or *variable-name*, or associated with  
 23 *data-pointer-object*. If the result variable is allocatable, it shall be deallocated before the iterator termi-  
 24 nates. Whether a non-allocatable result variable is finalized is processor dependent.

**NOTE 2.10**

Because an iterator is allowed but not required to have assigned a value to its result variable when it terminates, requiring a processor to finalize the result variable would require the processor to keep track of its definition status.

25 If CONCURRENT does not appear, execution of an ITERATE construct completes, the activation  
 26 record of the iterator instance is destroyed, the iterator instance ceases to exist, and the construct  
 27 becomes inactive when

- 28 • the iterator terminates,
- 29 • an EXIT statement that belongs to the ITERATE construct is executed,
- 30 • an EXIT or CYCLE statement that belongs to an outer construct and is within the range of the  
 1 ITERATE construct is executed,

- 2 • a branch occurs from a statement within the range of the ITERATE construct to a statement that
- 3 is neither the *end-iterate-stmt* nor within the range of the ITERATE construct, or
- 4 • a RETURN statement within the ITERATE construct is executed.

5 If CONCURRENT appears, execution of an ITERATE construct completes, the activation record of the  
6 iterator instance is destroyed, the iterator instance ceases to exist, and the construct becomes inactive  
7 when the iterator terminates and execution of all iterations is completed.

8 When execution of the ITERATE construct completes, if *declaration-type-spec* does not appear

- 9 • if = appears and *block* was executed, the value of *variable* is the value assigned by the ITERATE
- 10 statement before the final execution of *block*, or assigned during the final execution of *block*;
- 11 otherwise its definition status and value (if any) are the same as before execution of the ITERATE
- 12 construct, or
- 13 • if => appears and *block* was executed, the association status of *data-pointer-object* is as established
- 14 by the ITERATE statement before the final execution of *block*, or established during the final
- 15 execution of *block*; otherwise its association status is the same as before execution of the ITERATE
- 16 construct.

**NOTE 2.11**

The *variable* might become undefined during the final execution of *block*. The association status of *data-pointer-object* might become undefined during the final execution of *block*.

17 **2.4.5 Restrictions on DO CONCURRENT constructs**

18 Subclause 11.1.7.5 of ISO/IEC 1539-1:2018(E) concerning restrictions on DO CONCURRENT constructs  
19 is revised to apply to ITERATE CONCURRENT constructs as well.

20 **2.5 VALUE attribute**

21 The VALUE attribute shall be allowed for a dummy coroutine.

22 **2.6 PRESENT (A)**

23 The PRESENT intrinsic function inquires whether an optional dummy argument is associated with an  
1 actual argument in a function or iterator reference, a CALL statement, or a RESUME statement.

## 2 3 Examples

### 3 3.1 Quadrature example

4 This subclause presents four examples of a simple quadrature procedure. One uses forward communica-  
5 tion, two use reverse communication without coroutine syntax, and the fourth uses reverse communica-  
6 tion with coroutine syntax.

#### 7 3.1.1 Forward communication example

```

8  subroutine INTEGRATE ( A, B, ANSWER, ERROR, FUNC )
9      real, intent(in) :: A, B ! Bounds of the integral
10     real, intent(out) :: ANSWER, ERROR
11     interface
12         real function FUNC ( X )
13             real, intent(in) :: X
14         end function FUNC
15     end interface
16     real, parameter :: ABSCISSAE(...) = [ ... ]
17     real, parameter :: WEIGHTS(...) = [...]
18     integer :: I
19     answer = weights(1) * func( 0.5*(b+a) )
20     do i = 2, size(weights)
21         answer = answer + weights(i) * func( 0.5*(b+a) + (b-a) * abscissae(i) )
22         answer = answer + weights(i) * func( 0.5*(b+a) - (b-a) * abscissae(i) )
23     end do
24     answer = ( b - a ) * answer
25     error = ...
26 end subroutine INTEGRATE

```

#### 27 3.1.2 First reverse communication example

28 This example uses computed GO TO to resume computation after each integrand value is computed.  
29 Notice that the DO construct cannot be used because computation needs to be resumed within the  
30 construct. Further, this subroutine is not thread safe.

```

31  subroutine INTEGRATE ( A, B, ANSWER, ERROR, WHAT )
32     real, intent(in) :: A, B ! Bounds of the integral
33     real, intent(inout) :: ANSWER, ERROR
34     integer, intent(inout) :: WHAT
35     real, parameter :: ABSCISSAE(...) = [ ... ]
36     real, parameter :: WEIGHTS(...) = [...]
37     real, save :: RESULT
38     integer, save :: I
39     go to ( 10, 20, 30 ), what
40     i = 1
41     answer = 0.5 * ( a + b )
42     what = 1
43     return
44 10  result = answer * weights(1)
45 11  i = i + 1
46     if ( i > size(weights) ) then
47 1  what = 0

```

```

2         answer = ( a - b ) * result
3         error = ...
4         return
5     end if
6     answer = 0.5*(b+a) + (b-a) * abscissae(i)
7     what = 2
8     return
9 20     result = result + weights(i) * answer
10    answer = 0.5*(b+a) - (b-a) * abscissae(i)
11    what = 3
12    return
13 30     result = result + weights(i) * answer
14     go to 11
15     end subroutine INTEGRATE

```

16 This subroutine is used as follows:

```

17     what = 0
18     do
19         call integrate ( a, b, answer, error, what )
20         if ( what == 0 ) exit
21         ! evaluate the integrand at ANSWER and put the value into ANSWER
22     end do
23     ! Integral is in ANSWER here

```

### 24 3.1.3 Second reverse communication example

25 This example avoids GO TO statements and statement labels by structuring the quadrature subroutine  
26 as a “state machine.” The state indicates how to resume computation after each integrand value is  
27 computed. Although a DO construct can be used, control flow is difficult to follow because it is controlled  
28 by the state variable. This subroutine is also not thread safe.

```

29     subroutine INTEGRATE ( A, B, ANSWER, ERROR, WHAT )
30         real, intent(in) :: A, B ! Bounds of the integral
31         real, intent(inout) :: ANSWER, ERROR
32         integer, intent(inout) :: WHAT
33         real, parameter :: ABSCISSAE(...) = [ ... ]
34         real, parameter :: WEIGHTS(...) = [...]
35         real, save :: RESULT
36         integer, save :: I
37     do
38         select case ( what )
39         case ( 0 )
40             i = 1
41             answer = 0.5 * ( a + b )
42             what = 1
43             return
44         case ( 1 )
45             result = weights(1) * answer
46             what = 2
47         case ( 2 )
48             i = i + 1
49             if ( i > size(weights) ) then

```

```

2         what = 0
3         answer = ( a - b ) * result
4         error = ...
5         return
6     end if
7     answer = 0.5*(b+a) + (b-a) * abscissae(i)
8     what = 3
9     return
10    case ( 3 )
11        result = result + weights(i) * answer
12        answer = 0.5*(b+a) - (b-a) * abscissae(i)
13        what = 4
14        return
15    case ( 4 )
16        result = result + weights(i) * answer
17        what = 2
18    end select
19 end do
20 end subroutine INTEGRATE

```

21 This example is used the same way as the previous example.

### 22 3.1.4 Example using a coroutine

23 The coroutine organization is much clearer than the previous two examples.

```

24 coroutine INTEGRATE ( A, B, ANSWER, ERROR )
25     real, intent(in) :: A, B ! Bounds of the integral
26     real, intent(out) :: ANSWER, ERROR
27     real, parameter :: ABSCISSAE(...) = [ ... ]
28     real, parameter :: WEIGHTS(...) = [...]
29     integer :: I
30     answer = 0.5*(b+a)
31     suspend
32     result = answer * weights(1)
33     do i = 2, size(weights)
34         answer = 0.5*(b+a) + (b-a) * abscissae(i)
35         suspend
36         result = result + answer * weights(i)
37         answer = 0.5*(b+a) - (b-a) * abscissae(i)
38         suspend
39         result = result + answer * weights(i)
40     end do
41     answer = ( b - a ) * result
42     error = ...
43 end subroutine INTEGRATE

```

44 This coroutine is used as follows:

```

45 call integrate ( a, b, answer, error )
46 do while ( suspended(integrate) )
47     ! Evaluate the integrand at ANSWER and put the value into ANSWER
48     resume integrate ( a, b, answer, error )

```

```

2   end do
3   ! Integral is in ANSWER here

```

### 4 3.2 Iterator for a queue

5 This example performs a breadth-first traversal of a binary tree. It illustrates that the *block* of an  
6 ITERATE construct might change the object that is the attention of its iterator. Whether this “makes  
7 sense” in the general case is the responsibility of the iterator and other procedures that act on its  
8 arguments, or variables to which it has access by use or host association; it is not the responsibility the  
9 processor or the standard.

```

10  type :: Tree_Node_t
11      class(tree_node_t), pointer :: LeftSon => NULL(), RightSon => NULL()
12  end type Tree_Node_t
13
14  class(tree_node_t), pointer :: Root => NULL()
15
16  type :: Queue_Element_t
17      class(*), pointer :: Thing => NULL()
18      class(queue_element_t), pointer :: Next => NULL()
19  end type Queue_Element_t
20
21  type :: Queue_t
22      class(queue_element_t), pointer :: Head => NULL(), Tail => NULL()
23  contains
24      procedure :: DeQueue
25      procedure :: EnQueue
26  end type Queue_t
27
28  type(queue_t) :: MyQueue
29
30  call Fill_The_Tree ( root )
31  call myQueue%enQueue ( root ) ! Doesn't enqueue if root is NULL()
32  iterate ( class(*) :: node => myQueue%deQueue() )
33      ! This is an example where it ought to be possible to invoke (or resume) a
34      ! type-bound iterator (or function) that has no arguments other than
35      ! the passed-object argument without ().
36      select type ( node )
37      class ( tree_node_t )
38          call node%processIt
39          call myQueue%enQueue ( node%leftSon )
40          call myQueue%enQueue ( node%rightSon )
41      end select
42  end iterate
43
44  contains
45
46      iterator DeQueue ( TheQueue ) result ( Thing )
47          class(queue_t), intent(inout) :: TheQueue
48          class(*), pointer :: Thing
49          class(queue_element_t), pointer :: This
50          do
51              this => theQueue%head
52              if ( .not. associated(this) ) return ! terminate ITERATE construct

```

```

2      thing => this%thing
3      theQueue%head => this%next
4      deallocate ( this )
5      suspend ! Process Thing and come back here
6  end do
7  end iterator DeQueue
8
9  subroutine Enqueue ( TheQueue, Thing )
10     class(queue_t), intent(inout) :: TheQueue
11     class(*), intent(in), pointer :: Thing
12     class(queue_element_t), pointer :: This
13     if ( associated(thing) ) then
14         allocate ( this )
15         this%thing => thing
16         if ( associated(theQueue%tail) ) then
17             theQueue%tail%next => this
18         else
19             theQueue%head => this
20         end if
21         theQueue%tail => this
22     end if
23 end subroutine Enqueue

```

### 24 3.3 Preserving automatic variables

25 If one needs to invoke a procedure to solve several differently-sized problems, and the expense of creating  
26 local automatic variables is significant, it can be invoked initially in such a way as to create its automatic  
27 variables with the maximum extents necessary for the entire spectrum of problems to be solved. It can  
28 then be suspended, which does not destroy its automatic variables. When it is resumed to solve each  
1 problem, the automatic variables are intact.

2 **4 Required editorial changes to ISO/IEC 1539-1:2018(E)**

681 To be provided in due course.