Subject: Proposed technical changes
From: Van Snyder

Introduction

The following propositions are offered as potential technical changes to be advocated by USTAG as formal US public comments on the committee draft.

The reasons for these are:

(1) To correct a recently-added feature that was broken at its inception.

(2) To remove a processor-dependent feature that is worthless and, with one no-longer-used exception, has never been implemented.

(3) To remedy an inconsistency.

(4) So as not to foreclose future extension.

(5) By ignoring the advice in 98-170r2, facilities of C interoperability that could have been provided in a simple integrated way, but were insisted not to be necessary, have since been dribbled into Section 15 in an unnecessarily complex way.

(6) To remedy another inconsistency.

Edits are offered, with respect to 02-007r3, to illustrate the magnitude of the proposed change and to serve as a starting point for developing edits if the changes are accepted.

1 Using ACHAR(10) to signal a new line doesn't work

Using ACHAR(10) to signal a new line in formatted stream access doesn't work as well as we expect features of Fortran to work. The problem results from a conspiracy of the facts that the result of ACHAR(10) is a character of default kind, the *variable* and the *expr* have to be of the same kind in intrinsic assignment for characters, and both operands have to be of the same kind in an intrinsic character concatenation operation.

The reason for providing a character that causes a new line when it is output to a unit connected for formatted stream access was to allow a stream to be constructed in one or several parts of a program using concatenation and assignment, and output – perhaps to several units – in a different part of the program. The alternative was to use / formatting, but the sentiment was that that was inadequate.

The current mechanism, ACHAR(10), works just fine for characters of default kind, but it cannot be put into character strings of any other kinds. Essentially everything else in Fortran works for all kinds of the data type to which they apply.

Multiple kinds of characters were put into Fortran to support the needs of our colleagues who use other kinds of characters. If the facility to put a character that signals a new line when it is output to a unit connected for formatted stream access into a character string is useful for default kind, it is equally useful for other kinds of characters. The facility ought to be made complete. If a case cannot be made to make it complete, the case that it is necessary at all is very weak.

Either finish it or delete it.

1.1 Proposition

Replace the specification that ACHAR(10) causes a new line when it is output to a unit connected for formatted stream access with a specification that the result of an intrinsic function, say NEW_LINE, does that. The intrinsic function should have an argument that specifies the kind of the result – else there's little point in changing anything. The argument ought to be optional, and if it's absent the kind of the result ought to be default kind.

1.2 Edits to 02-007r3

[Editor: "the intrinsic ... ACHAR(10)" ⇒ "a reference to the intrinsic function NEW_LINE".] 230:6-7

13.7.82 NEW_LINE ([KIND]) 333:18-

Description. Returns a character that causes a new line when it is output to a unit connected for formatted stream access.

Class. Inquiry function.

Argument. KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. The result is a character of length one; it is of the kind given by KIND if KIND is present, or of default kind if KIND is absent.

Result Value. The result value is a processor-dependent character that causes a new line when it is output to a unit connected for formatted stream output.

It is recommended that the result of NEW_LINE is ACHAR(10) if KIND is absent or present with the value SELECTED_CHAR_KIND ('DEFAULT'), or CHAR(10,KIND) if KIND is present with the value SELECTED_CHAR_KIND ('ASCII') or SELECTED_CHAR_KIND ('ISO_10646').

2 Disappearing common blocks and module variables are an anachronism

Fortran 77 provided that a nonsaved named common block may cease to exist when no program unit is referencing it. Fortran 90 provided that a nonsaved module variable may cease to exist when no executing program unit is accessing the module in which it is declared. To my knowledge, only a Burroughs compiler actually caused nonsaved named common blocks to disappear, and no compiler causes nonsaved module variables to cease to exist when no program unit is accessing the module in which the variables are declared.

Because one cannot detect or control whether nonsaved module variables or nonsaved named common blocks cease to exist, removing this facility from the Fortran standard cannot invalidate any standard-conforming program.

Modern style guides recommend to use module variables instead of common blocks, so whether nonsaved named common blocks remain defined when no executing program unit is referencing them is becoming a moot question.

Modern style guides recommend to use module procedures instead of external procedures. If a program consists entirely of a Fortran main program and module procedures, every module is always accessible. Even if a program includes external procedures, every module is always accessible if none of the external procedures includes a USE statement – and it is unlikely that a developer would put a USE statement in an external procedure. Therefore, whether nonsaved module variables cease to exist when no executing program unit is referencing the module in which they are defined is probably a moot question.

Memory is cheap and plentiful and will become cheaper and more plentiful, and virtual memory is nearly universally available. If it hasn't been sufficiently important for processors to cause unreferenced variables to cease to exist, there will be less need in the future to do so. One can control exactly when storage is in use by a variable by using ALLOCATE and DEALLOCATE statements. Because one cannot depend on processors automatically causing unreferenced variables to cease to exist, careful developers use those facilities already.

The only way to allow module variables to cease to exist when no executing scoping unit is accessing them is to provide a reference counter for each module, and increment and decrement it whenever a scoping unit that accesses the module comes into existence or ceases to exist. If a small procedure is in a module that has numerous USE statements, it is possible that most of the execution time of that procedure is consumed in incrementing and decrementing reference counters, even if the program is processed by a processor that does aggressive inter-module inlining.

Therefore, the possibility that module variables might cease to exist is not only not useful, it has the potential to be downright harmful.

2.1 Proposition

Remove the discussion that nonsaved named common blocks may become undefined when no executing program unit is executing them, and the discussion that nonsaved module variables may become undefined when no executing program unit is referencing the module in which they are declared. Mark the use of SAVE for common block names and for module variables as obsolescent.

2.2 Edits to 02-007r3

(3) Previous standards provided that module variables and variables in common blocks could become undefined when no active program unit is accessing them. This feature has not been implemented by any processor, and provision for it is removed from this standard.	3:20+
[Editor: Delete "If the object ... finalized ... undefined."]	60:19-21
[Editor: Delete "A variable ... the module."]	61:0+4-5
[Editor: Delete "An entity ... undefined." (That's the whole paragraph.) Some of it will be re-inserted below.]	82:9-11
[Editor: Set "**or** / *common-block-name* /" in obsolescent font.]	89:14
[Editor: Set "or included ... the list" in obsolescent font.]	89:21
[Editor: Delete. Some of it will be re-inserted below.]	89:23-90:1
The current definition status of the common block storage sequence, and the values of those common block objects that are defined, are made available to each scoping unit that specifies the common block. For a named common block, this may be confirmed by specifying the SAVE attribute for the common block name. The definition status of each object in the common block storage sequence depends on the association that has been extablished for the common block storage sequence.	96:11+ New ¶
[Editor: Delete "(1) Execution ... undefined (16.5.6)." (That's the whole list item.)]	98:21-23
[Editor: Delete "If an unsaved ... deallocated" and Note 6.23.]	113:9-11ff
[Editor: Delete "that appears ... execution".]	115:8-9
[Editor: Delete "if the module ... execution".]	115:10-11
[Editor: Replace C1107 with the following nonconstraint paragraph:]	246:23-25

A procedure pointer or variable declared in the scoping unit of a module retains its association status, allocation status, definition status, and value unless it is a pointer and its target becomes undefined. This may be confirmed by specifying the SAVE attribute for the entity.

[I can't think of a reason for the *only-list* to be optional other than to keep nonsaved module variables in existence. Editor: Set the square brackets in obsolescent font.] 247:24

[Editor: Set "SAVE" in obsolescent font.] 250:4

---

(3) When execution of an instance of a subprogram completes, its unsaved local procedure pointers and variables become undefined. 411:33-41

2.3 On the other hand . . .

When we removed "Printing" we violated a "contract" with the users of the standard that features would not be deleted until they had endured in one edition marked as obsolescent. Perhaps the "disappearing module variables and common block variables" feature, and the "printing" feature, should both be set in obsolescent font.

3 A bizarre inconsistency

It is bizarre that one can write

```
typealias :: FOO => INTEGER
type(foo) :: BAR
```

but one cannot write

```
type(integer) :: BAR
```

There's nothing other than tradition preventing this: A derived type is prohibited from having the same name as an intrinsic type, so there is no possibility of confusion.

3.1 Proposition

Allow *type-spec*s for intrinsic types in TYPE() type specifiers.

3.2 Edits to 02-007r3

[Editor: Replace syntax rule R503 by] 67:16-23

R503	*type-spec*	**is**	*intrinsic-type-spec*
		or	TYPE (*derived-type-spec*)
		or	TYPE (*type-alias-name*)
		or	TYPE (*intrinsic-type-spec*)
R503$\frac{1}{2}$	*intrinsic-type-spec*	**is**	INTEGER [*kind-selector*]
		or	REAL [*kind-selector*]
		or	DOUBLE PRECISION
		or	COMPLEX [*kind-selector*]
		or	CHARACTER [*char-selector*]
		or	LOGICAL [*kind-selector*]

4 NONKIND is an unfortunate attribute name

NONKIND is an unfortunate name for an attribute of a type parameter. By using this name, we imply that two attributes of this variety are all that we will ever permit. We may want additional attributes of this variety. One possibility is an INITIALIZATION attribute, that indicates the parameter value has to be specified by an initialization expression, but it's not used for generic resolution. This would not be a KIND attribute, but what we currently call nonkind is explicitly prohibited from being used for initialization.

4.1 Proposition

What we currently call nonkind type parameters can only ultimately be used for character lengths or array dimensions. So as to allow other attributes of the KIND–NONKIND variety, change NONKIND to something more focused, such as EXTENT.

4.2 Edits to 02-007r3

[Editor: "a nonkind" ⇒ "an extent" (change the index entry too).]	32:7
[Editor: "A nonkind" ⇒ "An extent".]	32:12
[Editor: "a nonkind" ⇒ "an extent" at the following places: [32:13-14], [32:14+2], [32:22], [33:1], [45:28], [70:21-22], [415:36].]	
[Editor: "nonkind" ⇒ "extent" at the following places: [41:11], [44:35], [46:1], [50:12], [110:7], [125:13], [199:15], [269:10], [382:28], [424:26].]	
or EXTENT	42:17
[Editor: "NONKIND" ⇒ "EXTENT".]	46:4+5
[Editor: In the fourth line of Note 4.24, "NONKIND" ⇒ "EXTENT".]	47
[Editor: In the first line of Note 4.70, "a nonkind" ⇒ "an extent".]	65:bottom
[Editor: "Nonkind" ⇒ "Extent".]	77:18
[Editor: "Nonkind" ⇒ "Extent".]	77:23
[Editor: "A nonkind" ⇒ "An extent".]	418:7

5 Lots of C interoperability stuff is too complicated

In 98-170r2 it was proposed to use a POINTER(C) attribute to indicate that an entity interoperates with a C pointer. The advantages cited for this approach were:

- A C_PTR type would not be needed.
- A VALUE attribute for dummy arguments would not be needed – at least not strictly for the purpose of C interoperability: If the interface is BIND(C) then every argument is passed by value (as in C).
- The C_LOC intrinsic function would not be needed.
- Safe C pointer dereferencing would be possible, using semantics very similar to existing Fortran semantics.
- It would not be necessary to define a C_NULL_PTR named constant.

Since 26 June 1998, two more unnecessary intrinsic functions, *viz.* C_F_POINTER and C_ASSOCIATED, have been added to Section 15.

The argument that won the day against the approach proposed in 98-170r1 in 1998 was that many of the things that would be possible if it were adopted were never going to be necessary. They have in fact been implemented, but in unnecessarily complex ways.

One can produce pointers to pointers by the usual Fortran subterfuge of a structure having only a pointer component. The C standard does not, however, require the same physical representation for a pointer to a pointer and a pointer to a struct whose only component is a pointer, and this was one of the arguments advanced against the approach advocated in 98-170r1. Nonetheless, the present design assumes that all C pointers have the same physical representation.

5.1 Proposal

Define a variation on the pointer attribute, possibly spelled POINTER, BIND(C), or more tersely POINTER(C), that indicates the entity is a C pointer, not a Fortran pointer. Once we have an entity that's a pointer, much of the already-defined semantics are available.

Require that such a pointer be a scalar nonpolymorphic object with no nonkind type parameters. Provide no additional operations on the pointer association status beyond those already provided for any other scalar Fortran pointer.

Then

- C_LOC and C_F_POINTER are subsumed by ordinary pointer assignment,
- The C_NULL_PTR constant's functionality is provided by the NULLIFY statement, the NULL() intrinsic, or pointer assignment from a disassociated pointer *of either the Fortran or C variety*,
- C_ASSOCIATED is subsumed by ASSOCIATED (with variations in its semantics to make it behave like C_ASSOCIATED for the two-argument case).

After a net reduction of nearly three pages, we have a simpler facility with more power and no less safety.

5.2 Edits to 02-007r3

[Editor: "POINTER or ALLOCATABLE attribute" ⇒ "ALLOCATABLE attribute or the POINTER attribute without the (C) annotation".]	32:21
R432 *component-attr-spec* **is** POINTER [(C)]	42:33
[Editor: "POINTER attribute" ⇒ "the POINTER attribute without the (C) annotation".]	43:1-2
C428$\frac{1}{2}$ (R431) A component that has the POINTER attribute with the (C) annotation shall be a scalar.	43:7+
R437 *proc-component-attr-spec* **is** POINTER [(C)]	43:27
[Editor: Insert "[(C)]" after "POINTER".]	43:33
If a pointer component is specified with the (C) annotation it is an interoperable pointer, as described in 5.1.2.11.	48:7+ New ¶
[Editor: Insert "[(C)]" after "POINTER".]	68:10

[Editor: “POINTER attribute” $\Rightarrow$ “the POINTER attribute without the (C) annotation”.] 68:31

C509$\frac{1}{2}$ (R501) An object that has the POINTER attribute with the (C) annotation shall be a scalar. 68:31+

[Editor: Delete “POINTER,” at [69:25] and insert “, and the POINTER attribute shall not be specified unless it has the (C) annotation” after “specified” at [69:27].] 69:25,27

[Editor: Delete “, POINTER,” and insert “, and the POINTER attribute shall not be specified unless it has the (C) annotation,” after “specified”.] 69:31

If the POINTER attribute is specified with the (C) annotation it is an **interoperable pointer**; it has the following properties: 81:18+ New ¶

(1) If it has interoperable type and type parameters (15.2) it shall have the same representation as the companion processor would use for a pointer of the same type and type parameters; otherwise it shall have the same representation as the companion processor would use for a C `void` pointer.

(2) If it is not associated with a target its representation shall be the same as the companion processor uses for the value NULL specified by the C standard.

(3) If it is associated with a target its representation shall be the same as would result if the companion processor were to apply the C `&` operator to its target.

[Editor: Insert “[(C)]” after “POINTER”.] 89:2

[Editor: Add a sentence at the end of the paragraph: “A data pointer with the C annotation shall not be associated with a data pointer that does not have the C annotation; A procedure pointer with the C annotation shall not be associated with a procedure pointer that does not have the C annotation.”] 98:16

[Editor: After “one” insert “or be a pointer with the (C) annotation”.] 143:7

[Editor: “, and ... *data-target*” $\Rightarrow$ “. If *data-target* is not a pointer with the (C) annotation, its size”; before “The” insert “If it is a pointer with the (C) annotation, it shall be associated with an element of a rank-one array, and the number of elements from the element that is the target of the pointer to the end of the array, inclusive, shall not be less than the size of the *data-pointer-object*.”] 144:3-4

[Editor: “and” $\Rightarrow$ “whether it is a target (5.1.2.14 5.2.13),”; “or ... 5.2.13)” $\Rightarrow$ “, and if it is a pointer, whether it has the (C) annotation”.] 252:30

[Editor: After “pointer, insert “, if it is a pointer, whether it has the (C) annotation”.] 252:36

[Editor: Delete “and”; after “procedure pointer” insert “, and if it is a pointer or procedure pointer, whether it has the (C) annotation”.] 253:5

[Editor: Insert “[(C)]” after “POINTER”.] 260:20

A procedure pointer that has the POINTER attribute with the (C) annotation is an **interoperable procedure pointer**; it has the following properties: 261:13+ New ¶

(1) If *proc-interface* appears and *proc-interface* specifies the interface of an interoperable procedure, then it shall have the same representation as the companion processor would use for a function pointer having the same characteristics; otherwise it shall have the same representation as the companion processor would use for a C `void` pointer.

(2) If it is not associated with a target procedure its representation shall be the same as the companion processor uses for the value NULL specified by the C standard.

(3) If it is associated with a target procedure its representation shall be the same as would result if the companion processor were to apply the C & operator to its target.

[Editor: Delete Note 12.15.] 262:top

[Editor: Before "If" insert "If the dummy argument is a pointer with the (C) annotation, the actual argument shall be a pointer with the (C) annotation, a reference to a function that returns a pointer with the (C) annotation, or a reference to the NULL intrinsic function with a MOLD argument that is a pointer with the (C) annotation. If the dummy argument is a pointer that does not have the (C) annotation, the actual argument shall not be a pointer with the (C) annotation, a function that returns a pointer with the (C) annotation, or a reference to the NULL intrinsic function with a MOLD argument that is a pointer with the (C) annotation."] 265:25

If the dummy argument is a procedure pointer that does not have the (C) annotation, the associated actual argument shall be a procedure pointer that does not have the (C) annotation, a reference to a function that returns a procedure pointer that does not have the (C) annotation, or a reference to the NULL intrinsic function that does not have a MOLD argument that is a pointer with the (C) annotation. If the dummy argument is a procedure pointer with the (C) annotation, the actual argument shall be a procedure pointer with the (C) annotation, a function that returns a procedure pointer with the (C) annotation, or a reference to the NULL intrinsic function with a MOLD argument that is a pointer with the (C) annotation. 267:13-14

Case (ii$\frac{1}{3}$) If TARGET is present and POINTER has the (C) annotation the result is false if POINTER is not associated with a target. 300:16+

Case (ii$\frac{2}{3}$) If TARGET is present and is a pointer with the (C) annotation, and POINTER has the (C) annotation, the result is true if the representations of TARGET and POINTER compare equal in the sense of 6.3.2.3 and 6.5.9 of the C standard, and false if they do not compare equal in that sense.

[Editor: Delete subclause 15.1.2.] 382:9-385:0

[Editor: Delete subclause 15.2.2.] 386:1-4-

6 An old inconsistency

At [128:5-6] we have "The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement." Therefore

```
  call S ( intentoutarg=Y, intentinarg=F(Y) )
```

is OK, because Y isn't evaluated: If Y is defined before the statement is executed, it's defined when F is invoked. On the other hand

```
   X = G ( intentoutarg=Y, intentinarg=F(Y) )
```

is prohibited by [128:6-7], where we have "If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities shall not appear elsewhere in the same statement."

Here's an interesting one:

```
type T; integer :: X = 2; end type T
type(t) :: V(2) = (/ t(1), t(2) /)
call S ( intentoutarg = v(v(1)%x) )
```

Does v(1) undergo default initialization before v(1)%x is used for a subscript, in which case v(1)%x is 2, in which case it's v(2) that undergoes default initialization, in which case v(1)%x still has the value 1, in which case it's v(1) that undergoes default initialization, ...?

Clearly, expressions within designators have to be evaluated before actual arguments associated with INTENT(OUT) dummy arguments become undefined or undergo default initialization. So there should be no problem with

```
X = G ( intentoutarg=Y, intentinarg=F(Y) )
```

if F is an array instead of a function. Unfortunately, [128:6-7] explicitly makes this statement illegal.

6.1 Proposition

Instead of putting up with this, we should specify an order for things that happen during procedure invocation, but without putting an order on the processing of arguments. Actual arguments associated with INTENT(OUT) dummy arguments are finalized, then become undefined, then undergo default initialization. This change wouldn't invalidate a standard-conforming Fortran 95 program, so no interpretation is necessary.

6.2 Edits to 02-007r3

[Editor: After "statement" insert "except as a primary in an expression that is an actual argument to the same function reference".] 128:7

[Editor: Insert a new third-level subclause 12.4.2, and make the existing 12.4.2 and 12.4.3 subsidiary to it:] 272:1

12.4.2 Procedure reference

When a procedure is invoked, the following events occur, in the order specified.

(1) Expressions within actual arguments are evaluated, and expressions that are actual arguments associated with dummy arguments that do not have INTENT(OUT) are evaluated. The order of evaluation of these expressions is not specified.

(2) Each actual argument is associated with its corresponding dummy argument. If the dummy argument has INTENT (OUT) its corresponding actual argument is finalized and then

   (a) If it is not allocatable and not a pointer it becomes undefined; if it is of derived type any of its ultimate components that are allocatable become deallocated as if by a DEALLOCATE statement and the pointer association status of any of its ultimate components that are pointers becomes undefined; then it undergoes default initialization;

   (b) If it is a pointer its pointer association status becomes undefined;

   (c) If it is allocatable it becomes deallocated as if by a DEALLOCATE statement.

   The order of processing arguments, the relative order of these events between one argument and another, and whether arguments are associated before, during or after finalization and events (2a), (2b), or (2c) above are not specified.

(3) The sequence of execution transfers to the procedure.

12.4.2.1 Function reference

[Editor: Delete "When ... executed."] 272:3-4

12.4.2.2 Subroutine reference 272:9

[Editor: Delete "When ... executed."] 272:11-12