X3J3/94-009 r3
4-Oct-94

TECHNICAL PROPOSALS

This document contains technical proposals for revisions to Fortran. This is an internal working document of X3J3 and is regularly revised. These technical proposals are all in response to requirements defined by WG5. It requires a vote by X3J3 to add a proposal to this document or to change the status of one of the proposals.

There are currently three possible status values for a proposal.

D - Draft

This status indicates that the general concept of a proposal has been adopted, but that the proposal needs further work.

A - Approved

This status indicates that a proposal has been approved and is ready for incorporation into the draft standard (document 007).

I - Incorporated

This status indicates that a proposal has been incorporated into the draft standard. An I* status indicates that a proposal has been incorporated but not yet approved - such proposals might need to be pulled back out if they are not eventually approved; in that case, their status would revert to Draft.

The target date for a proposal can be either 95, 2k, or unspecified.

List of Proposals

Number	Status	Target date	Title
001	I	95	Changes to the MAXLOC and MINLOC intrinsics
002	I	95	NAMELIST comments
003	I	95	Minimal field widths
004	I	95	FORALL
005	I	95	PURE procedures
005a	A	95	Rationale for FORALL and PURE (belongs as part of 004 and 005)
006	I	95	Object Initialization
007	I	95	Language evolution
008	I	95	Conflicts with IEEE 754/854
009	I	95	CPU time
010	I	95	Nested WHERE
011	I	95	Specification Functions
012	I*	U	Enable
013	I	95	ELEMENTAL procedures
014	I	95	Automatic deallocation
015	I*	U	Allocatable components

Number: 001
Title: Changes to the MAXLOC and MINLOC Intrinsics
(B9 Item A5)
Status: Incorporated in 94-007r1
Target date: 95
Last revision: Feb 94
X3J3 reference: 94-037

Technical Description:

HPF modified the MAXLOC and MINLOC intrinsic functions to include an optional argument, DIM, which specifies the dimension along which the search is to be performed. This includes this functionality in Fortran 9x.

Discussion:

The HPF proposal creates an unfortunate incompatibility with Fortran 90. The incompatibility is a result of the new argument being inserted between the original first and second arguments. This was probably done for consistency with the argument location in the MAXVAL and MINVAL intrinsics. To avoid this incompatibility with Fortran 90, these functions are specified so that the two optional arguments may occur in either order. This places the burden on the processor to determine the order of the arguments (if the second argument is a scalar of type integer it is the DIM argument and if it is of type logical and conformable with the ARRAY argument it is the MASK argument). For consistency, this allowance for either order of the two optional arguments is extended to MAXVAL, MINVAL, PRODUCT, and SUM.

As part of this proposal, a Fortran 90 compatibility section has been added. It does not replace the Fortran 77 compatibility section and it assumes that Fortran 9x will be upward compatible from Fortran 90. Both of these issues are yet to be decided and if the text below is accepted, it may require revision when these decisions are made.

Detailed Edits:

[3/3+]: Add the following section:

1.4.1 Fortran 90 Compatibility

Except as noted in this section, this International Standard is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539:1991, informally known as Fortran 90, and a standard-conforming processor for this International Standard is a standard-conforming processor for

Fortran 90. Any standard-conforming Fortran 90 program remains standard-conforming under this International Standard. The following Fortran 90 features have different interpretations in this International Standard:

(1) The interfaces to the MAXLOC, MAXVAL, MINLOC, MINVAL, PRODUCT, and SUM intrinsics have been extended by this standard. This may conflict with a program that has supplied a generic interface to these intrinsics.

[3/4]: Change "1.4.1" to "1.4.2".

[219/6]: Add "DIM ," before "MASK" and add "or MAXLOC (ARRAY, MASK, DIM)" to the end of the line.

[219/7]: Add "DIM ," before "MASK".

[219/8]: Add "along dimension DIM" after "ARRAY".

[219/12+]: Add the following description of the DIM argument:

DIM (optional) must be scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

[219/14]: Change "; it" to ". If DIM is absent the result".

[219/15]: Before "." add "; otherwise, the result is an array of rank n-1 and shape (d(1), ..., d(DIM-1), d(DIM+1), ..., d(n)), where (d(1), ..., d(n)) is the shape of ARRAY".

[219/17]: Change "If MASK is absent, the result" to "The result of MAXLOC (ARRAY)".

[219/24]: Change "If MASK is present, the result" to "The result of MAXLOC (ARRAY, MASK = MASK)".

[219/31+]: Add the following case to the description of the result value:

Case (iii): If ARRAY has rank one, MAXLOC (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of MAXLOC (ARRAY [, MASK = MASK]). Otherwise, the value of element (s(1), ..., s(DIM-1), s(DIM+1), ..., s(n)) of MAXLOC (ARRAY, DIM = DIM [, MASK = MASK]) is equal to MAXLOC (ARRAY (s(1), s(2), ..., s(DIM-1), :, s(DIM+1), ..., s(n)) [, MASK = MASK (s(1), s(2), ..., s(DIM-1), :, s(DIM+1), ..., s(n))]).

[219/36+]: Add the following case to the examples:

Case (iii): The value of MAXLOC ((/ 5, -9, 3

```
                        /), DIM=1) is 1.  If B has the
                        value     | 1   3  -9 |
                                  | 2   2   6 |,  MAXLOC
                        (B,  DIM=1) is [ 2  1  2 ] and
                        MAXLOC  (B,  DIM=2) is [ 2  3 ].
                        Note that this is true even if B
                        has a declared lower bound other
                        than 1.
[219/37]:    Add "or MAXVAL (ARRAY, MASK, DIM)" to the end
             of the line.
[220/22,24]: Change "DIM" to "DIM = DIM".
[220/22,24]: Change "MASK" to "MASK = MASK".
[221/23]:    Add "DIM ," before "MASK" and add "or MINLOC
             (ARRAY, MASK, DIM)" to the end of the line.
[221/24]:    Add "DIM ," before "MASK".
[221/25]:    Add "along dimension DIM" after "ARRAY".
[221/29+]:   Add the following description of the DIM
             argument:
             DIM (optional)  must be scalar and of type
                             integer with a value in the
                             range 1 <= DIM <= n, where n is
                             the rank of ARRAY.  The
                             corresponding actual argument
                             must not be an optional dummy
                             argument.
[221/31]:    Change "; it" to ".  If DIM is absent the
             result".
[221/32]:    Before "." add "; otherwise, the result is an
             array of rank n-1 and shape (d(1), ...,
             d(DIM-1), d(DIM+1), ..., d(n)), where (d(1),
             ..., d(n)) is the shape of ARRAY".
[221/34]:    Change "If MASK is absent, the result" to "The
             result of MAXLOC (ARRAY)".
[222/6]:     Change "If MASK is present, the result" to "The
             result of MAXLOC (ARRAY, MASK = MASK)".
[222/13+]:   Add the following case to the description of
             the result value:
             Case (iii):  If ARRAY has rank one, MINLOC
                          (ARRAY, DIM = DIM [, MASK = MASK])
                          has a value equal to that of
                          MINLOC (ARRAY [, MASK = MASK]).
                          Otherwise, the value of element
                          (s(1), ..., s(DIM-1), s(DIM+1),
                          ..., s(n)) of MINLOC (ARRAY,
                          DIM = DIM [, MASK]) is equal to
                          MINLOC (ARRAY (s(1), s(2), ...,
                          s(DIM-1), :, s(DIM+1), ..., s(n))
                          [, MASK = MASK (s(1), s(2), ...,
                          s(DIM-1), :, s(DIM+1), ..., s9n))
                          ] ).
[222/18+]:   Add the following case to the examples:
             Case (iii):  The value of MINLOC ((/ 5, -9, 3
                          /), DIM=1) is 2.  If B has the
                          value     | 1   3  -9 |
```

| 2 2 6 |, MINLOC (B, DIM=1) is [1 2 1] and MINLOC (B, DIM=2) is [3 1]. Note that this is true even if B has a declared lower bound other than 1.

[222/19]: Add "or MINVAL (ARRAY, MASK, DIM)" to the end of the line.

[223/1,3]: Change "DIM" to "DIM = DIM".

[223/1,3]: Change "MASK" to "MASK = MASK".

[226/22]: Add "or PRODUCT (ARRAY, MASK, DIM)" to the end of the line.

[227/5,7]: Change "DIM" to "DIM = DIM".

[227/5,7]: Change "MASK" to "MASK = MASK".

[235/5]: Add "or SUM (ARRAY, MASK, DIM)" to the end of the line.

[235/27,29]: Change "DIM" to "DIM = DIM".

[235/27,29]: Change "MASK" to "MASK = MASK".

[Rationale]: Add the following sections to the new rationale:

R13.13.65 MAXLOC

Fortran 90 specified the MAXLOC intrinsic with only the ARRAY and MASK arguments. HPF added the DIM argument between the original two arguments for consistency with the MAXVAL intrinsic. This creates an incompatibility with Fortran 90 unless MAXLOC is specified as a generic interface with two specific interfaces: the first matching Fortran 90 and the second adding a non-optional DIM argument as the second argument. At X3J3 meeting 127, it was decided that this Standard should allow the DIM and MASK arguments to be specified in either order. For consistency, this provision for DIM and MASK to be specified in either order was extended to the other intrinsics, MAXVAL, MINVAL, PRODUCT, and SUM, which have the same arguments.

R13.13.66 MAXVAL

MAXVAL was extended to allow the DIM and MASK arguments to be specified in either order as part of the extensions to MAXLOC and MINLOC (R13.13.65,R13.13.70).

R13.13.70 MINLOC

Fortran 90 specified the MINLOC intrinsic with only the ARRAY and MASK arguments. HPF added the DIM argument between the original two arguments for consistency with the MINVAL intrinsic. This creates an incompatibility

with Fortran 90 unless MINLOC is specified as a generic interface with two specific interfaces: the first matching Fortran 90 and the second adding a non-optional DIM argument as the second argument. At X3J3 meeting 127, it was decided that this Standard should allow the DIM and MASK arguments to be specified in either order. For consistency, this provision for DIM and MASK to be specified in either order was extended to the other intrinsics, MAXVAL, MINVAL, PRODUCT, and SUM, which have the same arguments.

R13.13.71 MINVAL

MINVAL was extended to allow the DIM and MASK arguments to be specified in either order as part of the extensions to MAXLOC and MINLOC (R13.13.65,R13.13.70).

R13.13.81 PRODUCT

PRODUCT was extended to allow the DIM and MASK arguments to be specified in either order as part of the extensions to MAXLOC and MINLOC (R13.13.65,R13.13.70).

R13.13.103 SUM

SUM was extended to allow the DIM and MASK arguments to be specified in either order as part of the extensions to MAXLOC and MINLOC (R13.13.65,R13.13.70).

History:

X3J3/93-275r1 (meeting 127)
X3J3/94-037 (meeting 128)

Number: 002
Title: WG5 B9, B4.3: NAMELIST comments
Status: Incorporated in 94-007r1
Target date: 95
Last revision: Feb 94
X3J3 reference: 94-021r1

Technical Description:

Provide a mechanism to allow comments in namelist input records.

Discussion:

The straw votes taken at meeting 127 indicated a preference for allowing comments, of the "to end of line" variety. Comments are allowed before and after "name-value" subsequences as well as before the initial "&" defining a particular namelist group name.

Detailed Edits:

On page 151, section 10.9.1, change item 1 to read:

1) Optional blanks and namelist comments,

On page 152, section 10.9.1.2, add the following paragraph after the 4th paragraph:

A namelist comment may appear after any value separator except a slash. A namelist comment is also permitted to start in the first position of an input record except within a character literal constant.

On page 154, renumber section 10.9.1.6 to be 10.9.1.7, and add a new 10.9.1.6:

10.9.1.6 Namelist Comments

Except within a character literal constant, a "!" character after a value separator or in the first non-blank position of a namelist input record initiates a comment. The comment extends to the end of the current input record. The comment is ignored. A slash within the namelist comment does not terminate execution of the namelist input statement.

History: WG5/N901, X3J3/93-204r4, item 4 in X3J3 SD004, WG5/N840, X3J3/93-272, X3J3/94-021

Number: 003
Title: WG5 B9 item: B4.1. Minimal Field Widths
Status: Incorporated in 94-007r1
Target date: 95
Last revision: Feb 94
X3J3 reference: 94-022r1

Technical Description:

Allow a field width of zero for I, B, O, Z, and F edit descriptors (in formatted output) to request that the processor select the smallest field width which will avoid "*****"s (field overflow).

For the I, B, O, and Z edit descriptors, no leading blanks are produced, and a leading "+" is never written. When an "m" (minimal printable digit count) is specified, the appropriate number of leading zeros is still produced, i.e., the processor will chose a width >= m.

For the F edit descriptor, no leading blanks are produced, and a leading "+" is never written. The optional leading zero just before the decimal point is not produced unless d was specified to be zero.

A field width of zero is not permitted for other edit descriptors.

Discussion:

The straw votes taken at meeting 127 indicated a preference for this approach, rather than new edit descriptors (EX/LT, item 10 in SD004) to toggle the desired behavior.

The processor, when it sees a zero field width specified, choses the smallest possible value for the field width, such that, if the user had specified that particular value, the processor would have printed "useful" data (not "*"s), and chosing any smaller field width would have resulted in the processor printing "*"s (field width overflow).

This feature provides the ability to reduce the number of characters in an output file. Also, the user can maximize how many values can be printed on a single line and easily viewed on a terminal, while avoiding overflow "****"s in the output fields.

Detailed Edits:

On page 136, change the 3rd constraint to be:

Constraint: e must be positive.

Add this constraint after the 3rd constraint on page 136:

Constraint: w must be zero or positive for the I, B, O, Z, and F edit descriptors. w must be positive for all other edit descriptors.

In section 10.5.1.1, in the first paragraph, insert the following before the "." ending the first sentence [139:34]:

, except when w is zero. On input, w must not be zero. When w is zero, the processor selects the field width

In section 10.5.1.1, add the following to the end of the 4th paragraph [139:46]:

When w is zero, the processor choses a positive field width such that no leading blanks are produced, nor a leading plus, and the characters produced do not exceed the processor selected field width.

In section 10.5.1.1, add the following to the end of the 5th paragraph [140:4]:

When w is zero, the processor choses a positive field width such that no leading blanks are produced, and the characters produced do not exceed the processor selected field width.

In section 10.5.1.1, in the 6th paragraph, change the phrase "value of w" to be [140:8]:

value of w, except when w is zero

In section 10.5.1.1, add the following to the end of the 6th paragraph [140:9]:

When w is zero, the processor choses a positive field width such that no leading blanks are produced, nor a leading plus, and the characters produced do not exceed the processor selected field width. When m and w are both zero, and the value of the internal datum is zero, no characters are produced, regardless of the sign control in effect.

In section 10.5.1.2.1, in the 1st paragraph, insert the following before the 1st ",":

(except when w is zero)

In section 10.5.1.2.1, in the 1st paragraph, insert the following after the last sentence:

When w is zero, the processor selects the field width.

In section 10.5.1.2.1, in the last paragraph, insert the following after the last sentence [140:36]:

When w is zero, the processor choses a positive field width such that no leading blanks are produced, nor a leading plus, and the characters produced do not exceed the processor selected field width.

End of EDITS

History: WG5/N901, X3J3/93-204r4, items 9 and 10 in X3J3 SD004, X3J3/93-273, X3J3/94-022

Number: 004
Title: FORALL
Status: Incorporated in 94-007r2
Target date: 95
Last revision: May 94
X3J3 reference: 94-150r1

X3J3 94-150R1

TO: X3J3
FROM: Dick Hendrickson
SUBJECT: FORALL proposal
REFERENCE: 94-013, 94-054, 94-096, meeting 128 scribe notes, B9 resolution items A2 and A3
DATE: May 4, 1994

This is the FORALL proposal modified by the straw votes and discussion at meetings 128 and 129.

The Rationale is in paper 94-097 which was passed at meeting 128 and covers both the FORALL and PURE procedure proposal.

I did not do anything to resolve the copyright isues. As a result, I believe this text is still copyright by Rice University and copied here with their permission. It is unclear to me how to proceed with this. We can contact Rice and see if they will give us permission to include some of their text without acknowledgement. We can contact ANSI/ISO/??? and see if we can include a copyright notice in the text of the standard. We can rewrite the text to remove the dependence on Rice's material. It is also possible that using small parts of the text, and modifing it as much as I already have, removes any question of copyright "infringement". I believe that these issues are basically "editorial" and that we should decide what we want to do on a technical basis and then get the words right. Leaving the words close to the HPF words for now allows us to see what we've done to their technical content .

As an unresolved editorial issue /edit should consider "merging" the text for WHERE and FORALL with the text in Chapter 8 describing blocks. /edit should probably also consider "merging" the definitions and constraints for constructs which can have names.

FORALL proposal.

[add to 14.1.3 Statement entities after page 245, line 22. We need to define the index-name variables as statement entities to the forall statement or construct. This was not in the original HPF draft nor in the original paper. The intent is to say that the index name is purely local and its use does not affect the value of other variables with the same name in the rest of the program. This is the same as the implied Do index in an array constructor. Note that there are several other edits to this section from interpretations.]

The name of a variable that appears as an *index-name* in a FORALL statement or FORALL construct has a scope of the statement or construct. It has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the FORALL and this type must be INTEGER.

Change the section name for 14.1.3, [245:15] to "Statement and construct entities".

In [245:23-24] change the reference to "statement entities" to "statement or construct entities" twice.

In 14.1.2, [241:23] change the reference to "statement entities" to "statement or construct entities".

[Extend rule R215 for executable-construct to include the forall-construct and R216 to include the forall-stmt:]

Page 8, add to R215 executable construct:

or *forall-construct*

Page 9, add to R216 action-stmt

or *forall-stmt*

[Add FORALL construct to the list of things that may be branch targets. This allows branching to a FORALL, but not to an END FORALL]

Page 107, line 5: Add *forall-construct-stmt* to the list of possible branch targets.

[Add at the end of the first paragraph in section 7.5:]

Execution of a FORALL statement or FORALL construct controls the assignment to elements of arrays by using a set of index variables and a mask expression.

[Add a new section to chapter 7 at the end 7.5.3 [page 94] after the WHERE assignment section, number the BNF rules continuously:]

7.5.4 Element array assignment - FORALL

Element array assignment is used to mask the evaluation of expressions and assignment of values in assignment statements with selection by sets of index values and an optional mask expression.

7.5.4.1 The FORALL Construct

The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL statements and constructs to be controlled by a single *forall-triplet-spec-list* and *scalar-mask*.

General Form of the FORALL Construct

R781 *forall-construct* **is**
[*forall-construct-name*:] FORALL *forall-header*
forall-body-stmt
[*forall-body-stmt ...*]
END FORALL [*forall-construct-name*]

R782 *forall-header* **is** *(forall-triplet-spec-list* [,*scalar-mask-expr*] *)*

R783 *forall-triplet-spec* **is** *index-name* =
subscript:*subscript*[:*stride*]

R784 *forall-body-stmt* **is** *forall-assignment-stmt*
or *where-stmt*
or *where-construct*
or *forall-stmt*
or *forall-construct*

R785 *forall-assignment-stmt* **is** *assignment-stmt*
or *pointer-assignment-stmt*

Constraint: Any procedure referenced in the *scalar-mask-expr* of a *forall-header*, including one referenced by a defined operation, must be a *pure* procedure (12.xxx).

Constraint: The *scalar-mask-expr* must be scalar and of type logical. {footnote 1}

begin footnote 1

The *scalar-mask-expr* may depend on the *index-name* values as well as on the values of data items. This allows a wide range of masking operations.

end footnote

Constraint: A *forall-body-stmt* must not define any of the *index-name*s.

Constraint: The *index-name* must be a scalar integer variable.

Constraint: A subscript or stride in a *forall-triplet-spec* must not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

Constraint: If the *forall-construct* has a *forall-construct-name* the END FORALL statement must have the same *forall-construct-name*. If the END FORALL statement has a *forall-construct-name*. the *forall-construct* must have the same *construct-name*.

Constraint: Any procedure referenced in a *forall-body-stmt*, including one referenced by a defined operation or assignment, must be a *pure* procedure.

Constraint: A *forall-body-stmt* must not be a branch target.

To determine the set of values that each *index-name* in the *forall-header* takes let:

m1 be value of the first subscript ("lower bound");

m2 be value of the second subscript ("upper bound");

m3 be the stride; and

max be INT((m2 - m1 + m3)/m3)

If stride is missing, it is as if it were present with the value 1. Stride must not have the value 0.

The set of values is m1 + (k-1) * m3, k = 1, 2, ..., max. If max <= 0 for some *index-name*, the statements within the *forall-body* are not executed.

Each *forall-assignment-stmt* contained in a *forall-construct* assigns a value to a variable specified by the values of the *index-name* variables. A program must not assign multiple values to a variable by a single *forall-assignment-stmt* in a *forall-construct*. A program may, however, assign to the same variable in different *forall-assignment-stmt*s in a *forall-construct*. For the purposes of this restriction, any assignment (including array assignment or assignment to a variable of derived type) to a data object is considered to assign to all subobjects contained in that object. {footnote 2}

begin footnote 2

A syntactic consequence of the semantic rule that no two execution instances of a *forall-assignment-stmt* may assign to the same data object is that each of the *index-name* variables must appear on the left-hand side of a *forall-assignment-stmt*. The converse is not true (i.e., using all of the *index-name* variables on the left-hand side does not guarantee there will be no interference). Because the condition is not sufficient, it does not appear as a constraint. This restriction allows cases such as

```
FORALL ( I = 1:10 )
    A(INDEX(I)) = B(I)
END FORALL
```

if and only if INDEX(1:10) contains no repeated values. Note that it restricts FORALL behavior, but not syntax. Syntactic restrictions to enforce this behavior would be either incomplete (i.e., allow undefined behavior) or exclude useful programs such as the above example.

Statements can use the results of computations in lexically earlier statements, including computations done for other *index-name* values. However, an assignment never uses a value assigned in the same statement by another *index-name* value combination.

end footnote

The scope of an *index-name* is the *forall-construct* itself(14.1.3). {footnote 3}

footnote 3

The index variables inherit their type and type parameters from the host scope, but their use does not modify any host variables of the same name. Given a sequence such as:

```
INTEGER X
REAL XX
REAL A(5,4)
X=-1
J=10
FORALL(X=1:5, J=1:4)
    A(X,J) = J
END FORALL
```

After execution of the FORALL the variables X and J have the values -1 and 10 and the columns of A have the values 1, 2, 3, and 4. It would be a syntax error to use XX as a FORALL index.

end footnote

7.5.4.2 Interpretation of the FORALL Construct

Execution of a FORALL construct consists of the following steps:

Evaluation, in any order, of the subscript and stride expressions in the *forall-triplet-spec-list*. The set of combinations of *index-name* values is then the Cartesian product of the sets defined by these triplets.

Evaluation of the *scalar-mask-expr* for all combinations of *index-name* values. If the scalar mask expression is omitted, it is as if it were present with the value true. The mask elements may be evaluated in any order. The set of active combinations of *index-name* values is the subset of the combinations for which the *scalar-mask-expr* evaluates to .TRUE.. {footnote 4}

Footnote 4
Right-hand sides and expressions on the left hand side of a *forall-assignment-stmt* are defined as evaluated only for combinations of *index-names* for which the *scalar-mask-expr* evaluates to .TRUE. This has implications when the computation might create an error condition.
For example,

```
FORALL (I=1:N, Y(I).NE.0.0)
    X(I) = 1.0 / Y(I)
END FORALL
```

does not cause a division by zero nor does it assign any values to elements of X that correspond to zero elements of Y.

end footnote

Execution of the *forall-body-stmts* in the order they appear. Each statement is executed completely (that is, for all active combinations of *index-name* values) according to the following interpretation:

*forall-assignment-stmt*s evaluate the *expr* and all expressions within *variable* (in the case of *assignment-stmt*) or *target* and all expressions within *pointer-object* (in the case of *pointer-assignment-stmt*) of the *forall-assignment-stmt* for all active combinations of *index-name* values. These evaluations may be done in any order. The *expr* values are then assigned to the corresponding variable (in the case of *assignment-stmt*) or the *target* values are associated with the corresponding *pointer-object* (in the case of *pointer-assignment-stmt*). The assignment or association operations may also be performed in any order.

*where-stmt*s and *where-construct*s evaluate their *mask-expr* for all active combinations of values of *index-name*s. All elements of all masks may be evaluated in any order. The WHERE statements' assignment (or assignments within the WHERE block of the construct) are then executed in order using the above interpretation of array assignments within the FORALL, but the only array elements assigned are those selected by both the active *index-name* values and the WHERE mask. Finally, the assignments in the ELSEWHERE block are executed if that block is present. The assignments here are also treated as array assignments, but elements are only assigned if they are selected by both the active combinations and by the negation of the WHERE mask.

*forall-stmt*s and *forall-construct*s first evaluate the subscript and stride expressions in the *forall-triplet-spec-list* for all active combinations of the outer FORALL constructs. The set of valid combinations of *index-names* for the inner FORALL is then the union of the sets defined by these bounds and strides for each active combination of the outer *index-names*, the outer *index-names* being included in the combinations generated for the inner FORALL. The *scalar-mask-expression* is then evaluated for all valid combinations of the inner FORALL's *index-names* to produce the set of active combinations. If there is no *scalar-mask-expression*, it is as if it were present with the value true. Each statement in the inner FORALL is then executed for each active combination (of the inner FORALL), recursively following the interpretations given in this section. {footnote 5}

foot note 5
In general, any expression in a FORALL is evaluated only for valid combinations of all surrounding *index-names* for which all the *scalar-mask-expr*s are true.

Nested FORALL bounds and strides can depend on outer FORALL *index-names*. They cannot redefine those names, even temporarily.
end footnote

7.5.4.3 General form of the element array assignment statement

An element array assignment statement is a FORALL statement.

R786 *forall-stmt* **is** FORALL *forall-header*
forall-assignment-stmt

A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-stmt* that is a *forall-assignment*. {footnote 6}

The scope of an *index-name* is the FORALL statement itself(14.1.3).

FORALL statements cannot have construct names.

footnote 6

A FORALL construct means roughly the same thing as does replicating the FORALL header in front of each array assignment statement in the block, except that any expressions in the FORALL header are evaluated only once, rather than being re-evaluated before each of the statements in the body. The exceptions to this rule are nested FORALL statements and WHERE statements, which introduce syntactic and functional complications into the copying.

end footnote

7.5.4.4 FORALL Examples

Example 1:

```
FORALL (J=1:M, K=1:N) X(K,J) = Y(J,K)
FORALL (K=1:N) X(K,1:M) = Y(1:M,K)
```

These statements both copy columns 1 through N of array Y into rows 1 through N of array X. They are equivalent to

```
X(1:N,1:M) = TRANSPOSE(Y(1:M,1:N))
```

Example 2:

```
FORALL (I=1:N, J=1:N) X(I,J) = 1.0 / REAL(I+J-1)
```

This FORALL sets array element X(I,J) to the value 1/(I+J-1) for values of I and J between 1 and N.

Example 3:

```
FORALL (I=1:N, J=1:N, Y(I,J).NE.0 .AND. I .NE. J) &
         X(I,J) = 1.0/Y(I,J)
```

This statement takes the reciprocal of each non-zero non-diagonal element of array Y(1:N,1:N) and assigns it to the corresponding element of array X. Elements of Y that are zero or on the diagonal do not have their reciprocal taken, and no assignments are made to the corresponding elements of X.

Example 4:

```
FORALL (I=2:N-1) X(I) = (X(I-1) + 2*X(I) + X(I+1))/4
```

Has the same effect as the statement

```
X(2:N-1) = (X(1:N-2) + 2*X(2:N-1) + X(3:N+1))/4
```

Example 5:

```
FORALL (I=1:N) A(I,I) = X(I)
```

This FORALL statement sets the elements of the main diagonal of matrix A to the elements of vector X.

Example 6:

```
FORALL (K=1:5) J(K) = SUM(J(1:K))
```

This FORALL statement computes five partial sums of subarrays of J. (SUM is allowed in a FORALL because intrinsic functions are pure; see Section 12.xxx.) If before the FORALL

```
J =(/ 1, 2, 3, 4, 5/)
```

then after the FORALL

```
J =(/1, 3, 6, 10, 15/)
```

Example 7:

```
FORALL ( I=2:N-1, J=2:N-1 )
    A(I,J) = A(I,J-1) + A(I,J+1) + A(I-1,J) + A(I+1,J)
    B(I,J) = 1.0/A(I,J)
END FORALL
```

The assignment to array B uses the values of array A computed in the first statement, not the values before the FORALL began execution.

Example 8:

```
FORALL ( I=1:N-1 )
    FORALL ( J=I+1:N )  A(I,J) = A(J,I)
END FORALL
```

This FORALL construct assigns the transpose of the lower triangle of array A (i.e., the section below the main diagonal) to the upper triangle of A. For example, if N=3 and A originally contained the values

```
0       3       6
1       4       7
2       5       8
```

then after the FORALL it would contain

```
0       1       2
1       4       5
2       5       8
```

This could also be achieved with a single FORALL statement

FORALL (I = 1:N-1, J=1:N, J >I) A(I,J) = A(J,I)

Example 9:

```
INTEGER A(5,4), B(5,4)
FORALL ( I=1:5 )
    WHERE ( A(I,:) .EQ.  0 )  A(I,:) = I
    B(I,:) = I / A(I,:)
END FORALL
```

This FORALL construct, when executed with the input array

```
A =
      0   0   0      0
      1   1   1      0
      2   2   0      2
      1   0   2      3
      0   0   0      0
```

will produce as results

```
A =
      1   1   1      1
      1   1   1      2
      2   2   3      2
      1   4   2      3
      5   5   5      5
```

and

```
B =
       1   1   1       1
       2   2   2       1
       1   1   1       1
       4   1   2       1
       1   1   1       1
```

Note that assignments to A in the WHERE block may affect computations in the ELSEWHERE block such as to B(1,1).

Number: 005
Title: Pure procedures
Status: Incorporated in 94-007r2
Target date: 95
Last revision: May 94
X3J3 reference: 94-149r2 as amended (amended text not yet available)

X3J3 94-149R2

TO: X3J3
FROM: Dick Hendrickson
SUBJECT: Pure procedures
REFERENCES: 94-013, 94-054, 94-098, meeting 128 scribe notes
B9 resolution item A4
DATE: May 4, 1994

This is a modified version of 94-098 based on the discussions and scribe notes from meetings 128 and 129.

The Rationale is in paper 94-097 which was passed at meeting 128 and covers both the FORALL and PURE procedure proposal.

I did not do anything to resolve the copyright isues. As a result, I believe this text is still copyright by Rice University and copied here with their permission. It is unclear to me how to proceed with this. We can contact Rice and see if they will give us permission to include some of their text without acknowledgement. We can contact ANSI/ISO/??? and see if we can include a copyright notice in the text of the standard. We can rewrite the text to remove the dependence on Rice's material. It is also possible that using small parts of the text, and modifing it as much as I already have, removes any question of copyright "infringement". I believe that these issues are basically "editorial" and that we should decide what we want to do on a technical basis and then get the words right. Leaving the words close to the HPF words for now allows us to see what we've done to their technical content .

PURE proposal

[Add a new section to chapter 12 at an /editorially appropriate place.]

12.xxxx Pure Procedures

12.xxxx.1 Pure Procedure Declaration and Interface

A *pure* procedure is a procedure that has no side effects; it does not change the status of any variables known outside of its scope, except possibly for dummy arguments to a *pure* subroutine, and it does not perform any input or output to an external unit.

Intrinsic functions are always *pure*. Intrinsic subroutines are *pure* if they are elemental (e.g., MVBITS) but not otherwise. No explicit declaration of this fact is permitted for intrinsic procedures. A statement function is *pure* if and only if all functions that it references are *pure* .

The following constraints apply to *pure* functions or subroutines.

Constraint: The *specification-part* of a *pure* function must specify that all dummy arguments have INTENT (IN) except procedure arguments and arguments with the POINTER attribute.

Constraint: The *specification-part* of a *pure* subroutine must specify the intents of all dummy arguments except procedure arguments, alternate return specifiers, and arguments with the POINTER attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a *pure* procedure must not have the SAVE attribute. {footnote 1}

```
footnote 1
Variable initialization in a type-declaration-stmt or a data-stmt
implies the SAVE attribute; therefore, such initialization is also
disallowed.
end footnote
```

Constraint: If a procedure is used in a context that requires it to be pure, then its interface must be explicit in the scope of that use and that interface must specify the PURE keyword. {footnote 2}

```
footnote 2
It is expected that most mathematical library procedures will be
pure, this form of restriction allows these procedures to be used
in contexts where they are not required to be pure without the need
for an interface-block.
end footnote
```

Constraint: All internal procedures in a *pure* procedure must be *pure*.

Constraint: In a *pure* procedure any variable which is in common or accessed by host or use association; is a dummy argument to a *pure* function, is a dummy argument with INTENT (IN) to a *pure* subroutine, or an object that is storage associated with any such variable or subobject thereof, must not be used in the following contexts

As the *assignment variable* of an *assignment-stmt*;

As a DO variable or implied DO variable;

As an *input-item* in a *read-stmt* from an internal file;

As an *internal-file-unit* in a *write-stmt*;

As an IOSTAT= or SIZE= specifier in an input or output statement with an internal file;

In an *assign-stmt*;

As the *pointer-object* of a *pointer-assignment-stmt*;

As the *target* of a *pointer-assignment-stmt*;

As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type or is a pointer to a derived type, if the derived type has a pointer component at any level of component selection;

As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or

As an actual argument associated with a dummy argument with INTENT (OUT) or INTENT (INOUT) or with the POINTER attribute.

Constraint: Any procedure referenced in a *pure* procedure, including one referenced via a defined operation or assignment, must be *pure*.

Constraint: A *pure* procedure must not contain a *print-stmt, open-stmt, close-stmt, backspace-stmt, endfile-stmt, rewind-stmt, or inquire-stmt*.

Constraint: A *pure* procedure must not contain a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or *.

Constraint: A *pure* procedure must not contain a *stop-stmt*.

The above constraints are designed to guarantee that a *pure* procedure is free from side effects (i.e., modifications of data visible outside the procedure), which means that it is safe to reference it in constructs such as a FORALL *assignment-statement* where there is no explicit order of evaluation. {footnote 3}

footnote 3

The constraints on *pure* procedures may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a *pure* procedure must not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument, or perform any I/O or STOP operation. Note the use of the word conceivably; it is not sufficient for a *pure* procedure merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a *pure* function. The exclusion of functions of this nature is unavoidable if strict compile-time checking is to be used. In the choice between compile-time checking and flexibility, the committee decided in favor of enhanced checking.

It is expected that most library procedures will conform to the constraints required of *pure* procedures (by the very nature of library procedures), and so can be declared *pure* and referenced in FORALL

```
statements and constructs and within user-defined pure procedures.
It is also anticipated that most library procedures will not ref-
erence global data, whose use may sometimes inhibit concurrent ex-
ecution.  See Annex X for further discussion of the constraints.

end footnote
```

Pure subroutines are included to allow subroutine calls from *pure* procedures in a safe way, and to allow *forall-assignments* to be defined assignments. The constraints for *pure* subroutines are based on the same principles as for *pure* functions, except that side effects to INTENT (OUT) and INTENT (INOUT) dummy arguments are permitted. Pointer dummy arguments are always treated as INTENT (INOUT).

[Add PUREness to the list of procedure characteristics. Add after "subroutine" on line 36, page 165, section 12.2]

"whether or not it is *pure*, "

[Replace BNF rules on page 175 and 177 with:]

R1217 *prefix* **is** *prefix-spec* [*prefix-spec* ...]

R1217A *prefix-spec* **is** *type-spec*
or RECURSIVE
or PURE

Constraint: A *prefix* must contain at most one of each *prefix-spec*.

R1220 *subroutine-stmt* **is** [*prefix*] SUBROUTINE
subroutine-name [([*dummy-arg-list*])]

Constraint: The *prefix* of a *subroutine-stmt* must not contain a *type-spec*.

[To define interface specifications for *pure* procedures, the following constraints are added to Rule R1204 in Section 12.3.2.1 (defining interface-body)]:

Constraint: An interface-body of a *pure* procedure must specify the intents of all dummy arguments except POINTER**,** alternate return, and procedure arguments.

[To define *pure* procedure references, the following extra constraint is added to Rules R1209 and R1210 in Section 12.4.1 (defining function-reference and call-stmt):]

Constraint: In a reference to a *pure* procedure, a *procedure-name actual-arg* must be the name of a *pure* procedure. {footnote 4}

```
footnote  4
This constraint ensures that the purity of a procedure cannot be
undermined by allowing it to call a non-pure procedure.
end footnote
```

Examples of Pure Procedure Usage

Pure functions may be used in expressions in FORALL statements and constructs, unlike general functions which may have side effects. Several examples of this are given below.

```
! Intrinsic functions are always pure
      FORALL ( I = 1:N ) A(I,I) = LOG( ABS( A(I,I) ) )
```

Because a *forall-assignment* may be an array assignment, the *pure* function can have an array result. Such functions may be particularly helpful for performing row-wise or column-wise operations on an array.

```
INTERFACE
    PURE FUNCTION F(X)
    REAL, DIMENSION(3) :: F
    REAL, DIMENSION(3), INTENT (IN) :: X
    END FUNCTION F
END INTERFACE
REAL V (3,10,10)
...
FORALL (I=1:10, J=1:10) V(:,I,J) = F(V(:,I,J))
```

Because *pure* procedures have no constraints on their internal control flow (except that they may not use the STOP statement), they also provide a means for encapsulating more complex operations than could otherwise be nested within a FORALL. For example, the fragment below performs an iterative algorithm on every element of an array. Note that different amounts of computation may be required for different inputs.

```
PURE INTEGER FUNCTION ITER(X)
COMPLEX, INTENT (IN) :: X
COMPLEX XTMP
INTEGER I
I = 0
XTMP = -X
DO WHILE (ABS(XTMP).LT.2.0 .AND.  I.LT.1000)
    XTMP = XTMP * XTMP - X
    I = I + 1
END DO
ITER = I
END FUNCTION

...
FORALL (I=1:N, J=1:M) &
              IX(I,J) = ITER(CMPLX(A+I*DA,B+J*DB))
```

[Add to appendix B, the deleted features appendix.]

Constraint: A *pure* procedure must not contain a *pause-stmt.* {footnote 5}

footnote 5
A pause requires some form of input or output and is disallowed for the same reasons that other I/O statements are disallowed.
end footnote

[Add to the annex as a description of the *pure* procedure constraints.]

The constraints on *pure* procedures are limited to those necessary to check statically for freedom from side effects, for processor independence, and for lack of saved internal state. Subject to these restrictions, maximum functionality has been preserved in the definition of *pure* procedures. This has been done to make function calls in FORALL as widely available as possible, and so that quite general library procedures can be classified as *pure*.

A drawback of this flexibility is that *pure* procedures permit certain features whose use may hinder, and in the worst case prevent, concurrent execution in FORALL (that is, such references may have to be implemented by sequentialization). Foremost among these features are the access of global data, particularly distributed global data, and the fact that the arguments and, for a *pure* function, the result may be pointers or data structures with pointer components, including recursive data structures such as lists and trees. The programmer should be aware of the potential performance penalties of using such features.

The constraint requiring explicit INTENT (IN) for function arguments declares behavior that is ensured by the following constraints. It is not technically necessary, but is included for consistency with the explicit declaration rules for defined operators. Note that POINTER arguments may not have the INTENT attribute; the restrictions ensure that POINTER arguments also behave as if they had INTENT (IN), for both the argument itself and the object pointed to.

The constraint disallowing variables with the SAVE attribute ensures that a *pure* procedure does not retain an internal state between calls, which would allow side effects between calls to the same procedure

The constraint giving the restrictions on use of global (common or accessed by host or use association) variables and dummy arguments ensures that dummy arguments and global variables are not modified by the procedure. In the case of a dummy or global pointer, this applies to both its pointer association and its target value, so it cannot be subject to a pointer assignment or to an ALLOCATE, DEALLOCATE, or NULLIFY statement. These constraints imply that only local variables and the dummy function result variable can be subject to assignment or pointer assignment.

In addition, a dummy or global data object cannot be the target of a pointer assignment (i.e., it cannot be used as the right hand side of a pointer assignment to a local pointer or to the result variable), for then its value could be modified via the pointer. (An alternative approach would be to allow such objects to be pointer targets, but disallow assignments to those pointers; syntactic constraints to allow this would be even more draconian than these.)

In connection with the last point, it should be noted that an ordinary (as opposed to pointer) assignment to a variable of derived type that has a pointer component at any level of component selection may result in a

pointer assignment to the pointer component of the variable. That is certainly the case for an intrinsic assignment. In that case, the expression on the right hand side of the assignment has the same type as the assignment variable, and the assignment results in a pointer assignment of the pointer components of the expression result to the corresponding components of the variable (see section 7.5.1.5). However, it may also be the case for a defined assignment to such a variable, even if the data type of the expression has no pointer components; the defined assignment may still involve pointer assignment of part or all of the expression result to the pointer components of the assignment variable. Therefore, a dummy or global object cannot be used as the right hand side of any assignment to a variable of derived type with pointer components, for then it, or part of it, might be the target of a pointer assignment, in violation of the restriction mentioned above.

The last two paragraphs only prevent the reference of a dummy or global object as the only object on the right hand side of a pointer assignment or an assignment to a variable with pointer components. There are no constraints on its reference as an operand, actual argument, subscript expression, etc., in these circumstances.

Finally, a dummy or global data object cannot be used in a procedure reference as an actual argument associated with a dummy argument of INTENT (OUT) or INTENT (INOUT) or with a dummy pointer, for then it may be modified by the procedure reference. This constraint, like the others, can be statically checked, since any procedure referenced within a *pure* function must be either a *pure* function, which does not modify its arguments, or a *pure* subroutine, whose interface must specify the INTENT or POINTER attributes of its arguments. Incidentally, notice that in this context it is assumed that an actual argument associated with a dummy pointer is modified, since Fortran does not allow its intent to be specified.

The constraint that only *pure* procedures may be called ensures that all procedures called from a *pure* procedure are themselves side-effect free, except, in the case of subroutines, for modifying actual arguments associated with dummy pointers or dummy arguments with INTENT (OUT) or INTENT (INOUT).

A constraint prevents external I/O and file operations, whose order would be non-deterministic in the context of concurrent execution. Note that internal I/O is allowed, provided that it does not modify global variables or dummy arguments.

Finally, a constraint disallows STOP statements. A STOP brings execution to a halt, which is a rather drastic side effect.

Number: 005a
Title: Rationale for FORALL and PURE
Status: Approved
Target date: 95
Last revision: Feb 94
X3J3 reference: 94-097

TO: X3J3
FROM: Dick Hendrickson
SUBJECT: Rationale for FORALL and PURE

The following is the rationale section for the FORALL and PURE features.

NOTE that the HPF document is copyrighted by Rice University. Rice has given free permission to copy from the document, but the copyright must be mentioned. Paper 94-013 should have mentioned this fact also.

Rational:

The FORALL statement and construct and PURE procedures were added to Fortran 95 to allow the majority of programs coded in High Performance Fortran (HPF) to run on a standard conforming Fortran 95 processor with little change. These added concepts are the major new syntatic features of HPF. HPF was primarily designed to allow programs to execute efficiently on multi-processor systems. Adding these features to Fortran 95 does not imply that a Fortran 95 processor is a multi-processor nor that any features of the language are safe or consistent on a multi-processor system.

Some of the text describing the FORALL and PURE features was taken directly from the High Performance Fortran Language Specification, Version 1.0, May 3, 1993 c1993 Rice University, Houston Texas. The text was copied with the permission of Rice University.

The purpose of the FORALL statement and construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement and the WHERE construct in Fortran 90. FORALL differs from these constructs in its syntax, which is intended to be more suggestive of local operations on each element of an array, and in its generality, which allows a larger class of array sections to be specified. In addition, a FORALL may invoke user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax).

HPF defines a new procedure attribute, PURE, to declare the class of functions that may be invoked in this way. Both single-statement and block FORALL forms are defined in this section, as well as the PURE attribute and constraints arising from the use of PURE.

Fortran 90 places several restrictions on array assignments. In particular, it requires that operands of the right side expressions be conformable with the left

hand side array. These restrictions can be relaxed by introducing the element array assignment statement, usually referred to as the FORALL statement. This statement is used to specify an array assignment in terms of array elements or groups of array sections, possibly masked with a scalar logical expression. In functionality, it is similar to array assignment statements and WHERE statements. The FORALL statement essentially preserves the semantics of Fortran 90 array assignments and allows for convenient assignments like

```
FORALL ( i=1:n, j=1:m ) a(i,j)=i+j
```

as opposed to standard Fortran 90

```
a =         SPREAD((/(i,i=1,n)/), DIM=2, NCOPIES=m) + &
            SPREAD((/(i,i=1,m)/), DIM=1, NCOPIES=n)
```

It can also express more general array sections than the standard triplet notation for array expressions. For example,

```
FORALL ( i = 1:n ) a(i,i) = b(i)
```

assigns to the elements on the main diagonal of array a.

It is important to note, however, that FORALL is not intended to be a general parallel construct; for example, it does not express pipelined computations or MIMD computation well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics.

A PURE function is one that obeys certain syntactic constraints that ensure it produces no side effects. This means that the only effect of a pure function reference on the state of a program is to return a result---it does not modify the values, pointer associations, or data mapping of any of its arguments or global data, and performs no external I/O. A pure subroutine is one that produces no side effects except for modifying the values and/or pointer associations of INTENT (OUT) and INTENT (INOUT) arguments. These properties are declared by a new attribute (the PURE attribute) of the procedure.

A pure procedure (i.e., function or subroutine) may be used in any way that a normal procedure can. However, a procedure is required to be pure if it is used in any of the following contexts:

- In the mask or body of a FORALL statement or construct;
- Within the body of a PURE procedure; or
- As an actual argument in a PURE procedure reference.

The freedom from side effects of a pure function allows the function to be invoked concurrently in a FORALL without such undesirable consequences as nondeterminism, and additionally assists the efficient implementation of concurrent execution. Syntactic constraints (rather than semantic constraints on behavior) are used to enable compiler checking.

Number: 006
Title: B9/B1 Object Initialization
Status: Incorporated in 94-007r2
Target date: 95
Last revision: May 94
X3J3 reference: 94-138r3

X3J3/94-138r3

To: X3J3
From: /OOF
Subject: Text for X3J3/009 re Object Initialization (B1)
References: WG5-N930 Resolutions of the Berchtesgaden WG5 Meeting, B9
WG5-N932 Requirement for the Initialization of Pointers and Objects
X3J3/93-207 Pointer and Derived Type Initialization
X3J3/93-204r4 B9 Implementation Plan
X3J3/93-237 OOF Report from Meeting 126
X3J3/93-259 Proposal for Object Initialization (B1)
X3J3/93-296 Constructors and Destructors plus Object Initialization (Tutorial)
X3J3/94-030 OOF Report from Meeting 127
X3J3/94-031r2 Proposal for Object Initialization (B1)
X3J3/94-081 Object Initialization and Memory Conservation (Overheads)

Number: 006

Requirement Title: B9/B1 Object Initialization

Status: Approved

Technical Description: Currently all pointers are created with an undefined association status. There is a requirement to change this to allow some pointers to be created with a disassociated association status. The means chosen by straw vote within X3J3 to specify an initial association status of disassociated involves the appearance in initialization contexts of a new intrinsic function NULL with a single optional argument. The syntax "=> NULL( )" may appear in a type declaration statement. The intrinsic function may also appear in a structure constructor to correspond with a pointer component.

There is a further requirement to extend a type definition to contain the specification of a default initial value for a nonpointer component and the specification of disassociated status as the default for a pointer component. It is not necessary for a default value to be specified for each nonpointer component in a definition nor to change the initial association status of each pointer component from undefined to disassociated. If a component is of derived type, its initial state may be specified in the type definition of that type. Conversely, even though a derived-type component has an initial state specified in its type definition, the initial state may be overridden by the use of a structure constructor for that type in the (higher-level) component specification. The effect of specifying default initial values for nonpointer components and the disassociated status for pointer components is that whenever an object of the type is created, by declaration or

allocation, it will be automatically initialized as indicated in the type definition.

As is the case with objects of intrinsic type, it is possible to specify initial values for objects of derived type in type declarations and DATA statements. For an object of derived type, an initial value so specified overrides any default initialization contained in the type definition. An object of a derived type with default initialization specified must not appear in a DATA statement. Note that whereas initialization in a type declaration statement or a DATA statement implies that the object initialized has the SAVE attribute, specification of default initialization carries no such implication for objects of the type.

The syntax needed to meet this requirement for nonpointer components, would seem to be a natural extension of the initialization mechanism used for an object in a type declaration. It is merely a matter of allowing = *initialization* to appear in a component specification. The means described above for initializing a pointer to have an initial status of disassociated can then be used to specify a default disassociated status for a pointer component.

To summarize these extensions:

- => NULL( ) can appear in a type declaration.
- NULL([<pointer-name>]) can appear in a structure constructor.
- Initialization specifications that can appear in a type declaration statement can also appear in a component specification to specify default initialization.
- Default initialization can be overridden by a higher level default initialization or by explicit initialization.

Discussion: The specification of default initialization is a facility that is not available for objects of intrinsic type.

The current proposal does not allow a pointer to have a default association status of defined, although there is no conceptual problem with doing this so long as the target has been declared with the TARGET and SAVE attributes.

Detailed Edits: [94-006r0 was integrated into these edits. The only collision was item 87 as noted below.]

[32:36+] add a new paragraph

> **Default initialization** is specified for a component of an object of derived type when initialization appears in the component declaration. The object will be initialized as specified in the derived type definition (14.7.3, 14.7.5) even if the definition is private or inaccessible. Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute. Default initialization does not apply to dummy arguments unless they have INTENT (OUT).

[33:29-30] replace with

R429 *component-decl* **is** *component-name* [(*component-array-spec*)]
[* *char-length*] [*component-initialization*]

R429a *component-initialization* **is** = *initialization-expr*
or => NULL ( )

[33:38+] add constraints

Constraint: If *component-initialization* appears, a double colon separator must appear before the *component-decl-list*.

Constraint: If => appears in *component-initialization*, the POINTER attribute must appear in the *component-attr-spec-list*. If = appears in *component-initialization*, the POINTER attribute must not appear in the *component-attr-spec-list*.

Constraint: If an object with default initialization is specified in a common block, the common block must be saved, and if an object with default initialization is specifed in the specification part of a module, the object must be saved.

Note that objects in named common may be initially defined only in a block data program unit (5.5.2.4). Objects with default initialization must not appear in blank common.

[34:1-2] replace with

The double colon separator in a *component-def-stmt* is required if the DIMENSION attribute, the POINTER attribute, or *component-initialization* is specified; otherwise, it is optional.

If *initialization-expr* appears, an object of the type becomes defined with the default initial value determined from *initialization-expr* unless the default initial value is overridden by explicit initialization. The *initialization-expr* is evaluated in the scoping unit of the type definition in accordance with the rules of intrinsic assignment (7.5.1.4).

[34:6+] add

Note that default initialization of an array component may be specified by a constant expression consisting of an array constructor, or of a single scalar that becomes the value of each array element.

A component is a pointer if its *component-attr-spec-list* contains the POINTER attribute. Pointers have an association status of “undefined”, “disassociated”, or “associated”. If no default initialization is specified, the initial status is “undefined”. **Pointer nullification** sets the status of a pointer to “disassociated”. To specify that the default initialization of a pointer component is to be “disassociated”, the pointer assignment symbol (=>) must be followed by a reference to the intrinsic function NULL( ) with no argument. No mechanism is provided to specify a default initial status of “associated”.

[35:30] insert after “component.”

The type definition may specify that in objects declared to be of this type, such a pointer is initially disassociated.

[35:33] replace with

```
TYPE(NODE), POINTER :: NEXT_NODE => NULL(  )
```

[35:35+] Add

It is not required that initialization be specified for each component of a derived type. For example:

```
TYPE DATE
  INTEGER DAY
  CHARACTER (5) MONTH
  INTEGER :: YEAR = 1994  ! Partial default initialization
END TYPE DATE
```

If a component is of intrinsic type and is not a pointer, a default initial value may be specified by an initialization expression. If the component is an array, the initialization expression must be conformable; it may be a scalar or an array constructor (4.5). If the component is a pointer, a limited form of pointer assignment (7.5.2) may be used to specify that in objects of the type, the pointer component is initially disassociated. If the component is of derived type and does not have the pointer attribute, its default initial value may be specified in the type definition for that derived type or by a structure constructor that may override any lower-level default initialization.

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)
```

The default initial value of a component of derived type may be overridden by a higher level default initialization. For example:

```
TYPE SINGLE_SCORE
  TYPE(DATE) :: PLAY_DAY = TODAY
  INTEGER SCORE
  TYPE(SINGLE_SCORE), POINTER :: NEXT => NULL( )
END TYPE SINGLE_SCORE

TYPE(SINGLE_SCORE) SETUP
```

The PLAY_DAY component of SETUP receives its initial value from TODAY overriding the lower-level initialization for the YEAR component.

Arrays of structures may be declared whose elements are partially or totally initialized by default. For example:

```
TYPE MEMBER
  CHARACTER (20) NAME
  INTEGER :: TEAM_NO, HANDICAP = 0
  TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL( )
END TYPE MEMBER
```

```
TYPE (MEMBER) LEAGUE (36)      ! Array of partially
                               !    initialized elements

TYPE (MEMBER):: ORGANIZER=MEMBER ("I. Manage",1,5,NULL( ))
```

ORGANIZER is explicitly initialized, overriding the default initialization for an object of type MEMBER. Allocated objects may also be initialized partially or totally. For example:

```
ALLOCATE (ORGANIZER % HISTORY) ! A partially initialized
                               !  object of type
                               !  SINGLE_SCORE is created.
```

[37:11] replace “constant expressions” with

constant expressions or pointer nullifications

[39:36-37] replace with

R504 *entity-decl* **is** *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]

[39:38+] add

R504a *initialization* **is** = *initialization-expr*
or => NULL ( )

[40:1] = *initialization-expr* -> *initialization*

[40:3] The = *initialization-expr* -> *initialization*

[40:5] delete “a pointer,”

[40:6+] add constraint

Constraint: If => appears in *initialization*, the object must have the POINTER attribute. If = appears in *initialization*, the object must not have the POINTER attribute.

[40:34] = *initialization-expr* -> *initialization*

[40:44] an = *initialization-expr* -> explicit initialization.

[41:8+] add paragraph

If *entity-decl* contains a reference to the NULL intrinsic function, *object-name* must be a pointer, and its initial association status is disassociated.

[41:9-10] change sentence to [includes edit from defect item 87]

The presence of *initialization* implies that *object-name* is saved, except for an *object-name* in a named common block or an *object-name* with the PARAMETER attribute.

[41:22+] add

```
TYPE(CHAIN), POINTER :: HEAD => NULL( )
```

[44:6+] add

```
TYPE(NODE), PARAMETER :: DEFAULT = NODE(0, NULL( ))
```

[44:30] add before period, “except for components of an object of derived type for which default initialization has been specified”.

[51:35+] add sentence, “If an object or subobject has been specified with default initialization in a type definition, it must not appear in a *data-stmt-object-list*.”

[76:22+] add

The intrinsic function NULL returns a disassociated pointer. A disassociated pointer has no shape but does have rank. The data type, type parameters, and rank of the result of the intrinsic function NULL when it appears without an argument are determined by the pointer that becomes associated with the result. If the intrinsic function appears on the right of a pointer assignment statement, the type, type parameters, and rank of the result are those of the pointer on the left. If the intrinsic function appears in *initialization* for an object in a type declaration statement, the type, type parameters, and rank of the result are those of the object. If the intrinsic function appears as a default initialization specification in a *component-decl*, the type, type parameters, and rank of the result are those of the component. If the intrinsic function appears in a structure constructor, the type, type parameters, and rank are determined by the corresponding pointer component. If the intrinsic function appears as an actual argument in a procedure reference, the type, type parameters, and rank of the result are those of the corresponding dummy argument.

The optional argument is required when the intrinsic function NULL appears as an actual argument in a reference to a generic procedure if the argument type, type parameters, and rank are required to resolve the reference. For example:

```
INTERFACE GEN
  SUBROUTINE S1 (J, PI)
    INTEGER J
    INTEGER, POINTER :: PI
  END SUBROUTINE S1
  SUBROUTINE S2 (K, PR)
    INTEGER K
    REAL, POINTER :: PR
  END SUBROUTINE S2
END INTERFACE

REAL, POINTER :: REAL_PTR
CALL GEN (7, NULL (REAL_PTR) )          ! Invokes S2
```

[77:23+] add new item to list

(5a) A reference to the transformational intrinsic function NULL,

[78:5] add new item to list

(5a) A reference to the transformational intrinsic function NULL,

[79:10+] add new item to list

(8a) The transformational NULL,

[92:17] Replace "If the *target* is a pointer that is disassociated," with "If the *target* is a pointer that is disassociated or a reference to the NULL intrinsic function," and delete "also".

[92:28+] add

PTR => NULL ( )

[187:32] change title to

13.8.10 Pointer association status functions

[187:33] The function -> The function NULL returns a disassociated pointer. The inquiry function

[192:13] change title to

13.10.20 Pointer association status functions

[192:14] Association status or comparison -> Association status inquiry or comparison

[192:15+] add

NULL (MOLD) Returns disassociated pointer
Optional MOLD

[225:19+] add

13.13.77a NULL (MOLD)

Optional Argument. MOLD

Description. Returns a disassociated pointer.

Class. Transformational function.

Argument. MOLD must be a pointer and may be of any type. Its pointer association status may be undefined, disassociated, or associated. If its status is associated, the target need not be defined with a value.

Result Type, Type Parameters, and Rank. Determined by context (7.1.4.1) if MOLD is not present; otherwise, the same as MOLD.

Result. The result is a pointer with disassociated association status.

Example. REAL, POINTER, DIMENSION(:) :: VEC => NULL( ) sets the initial association status of VEC to disassociated.

[243:13] add new paragraph

Note that if the intrinsic function NULL appears as an actual argument in a reference to a generic procedure, the argument MOLD may be required to resolve the reference (7.1.4.1).

[246:35] becomes -> is

[246:35+] add (and reletter remaining items)

(a) The pointer is explicitly initialized in a type declaration (5.1)

(b) Default initialization is specified for the pointer in a type definition (4.4)

[249:38+] add new item to the list and adjust the list appropriately

(3) Components of SAVEd variables for which default initialization is specified, and

[251:6+] add new items to the list and adjust the list appropriately

(17) Allocation of an object of a derived type, in which default initialization is specified for some components, causes those components of the object to become defined.

(19) Invocation of a procedure that contains a nonSAVEd local object that is not a dummy argument and is of a derived type in which default initialization is specified for some components, causes those components of the object to become defined.

(20) Invocation of a procedure that has an INTENT (OUT) dummy argument of a derived type that specifies default initialization for some components, causes those components of the dummy argument to become defined.

[256:10+] add new glossary item

default initialization (4.4) : If initialization is specified in a type definition, an object of the type will be automatically initialized. Nonpointer components may be initialized with values by default; pointer components may be initially disassociated by default. Default initialization is not provided for objects of intrinsic type.

[256:42+] add new glossary item

explicit initialization : Explicit initialization may be specified for objects of intrinsic or derived type in type declaration statements or DATA statements. An object of a derived type that specifies *default initialization* may not appear in a DATA statement.

[268:2-3] replace with

```
INTEGER :: VAL = 0
TYPE(CELL), POINTER :: NEXT_CELL => NULL( )
```

[268:5] replace with

```
TYPE(CELL), TARGET :: HEAD    ! Automatically initialized
```

[268:9] delete

[268:15] delete

Add to new Rationale Section:

4.4 Derived types

The prime motivation for adding a means to specify default initialization for objects of derived type is a need to eliminate situations in which dynamic memory becomes unavailable. This can occur in applications that manipulate objects of derived type with pointer components. Most memory leakage can be avoided if it is possible to specify that pointers be created with an initial status of disassociated. In order to allow the initial status of a pointer to be specified as disassociated in declarations, structure constructors, or type definitions, a new intrinsic function, NULL, with a single optional argument is provided. When the intrinsic function NULL appears in a type definition, the optional argument must not appear. When the intrinsic function NULL appears elsewhere, the argument is optional if the type, type parameters, and rank of the required disassociated pointer are apparent from the context. If this is not the case, a pointer object name must appear as the argument to indicate the type, type parameters, and rank of the required disassociated pointer. The argument acts as a mold. Such an argument may be necessary when a disassociated pointer is used as an actual argument in a generic procedure reference (7.1.4.1). An example of the initialization of a pointer as disassociated is:

```
REAL, POINTER, DIMENSION (:) :: VEC => NULL( )
```

The following code fragment illustrates the memory leakage problem:

```
MODULE CHARACTER_VARYING
  TYPE VARYING
    PRIVATE
    CHARACTER, POINTER :: CHARS(:)
  END TYPE VARYING

! Generic interfaces for assignment, concatenation, conversion, etc.
   ...

CONTAINS
```

```
  SUBROUTINE VS_ASS_VS (VAR, EXPR)       ! Assignment subroutine
    TYPE(VARYING), INTENT(OUT) :: VAR
    TYPE(VARYING), INTENT(IN) :: EXPR
    ALLOCATE (VAR % CHARS (1:SIZE(EXPR % CHARS)))
    VAR % CHARS = EXPR % CHARS
  END SUBROUTINE VS_ASS_VS
  ...
END MODULE CHARACTER_VARYING

MODULE VOCABULARY
  USE CHARACTER_VARYING
  PRIVATE
  TYPE(VARYING), PUBLIC :: WORDS (5000)   ! Words in random order
  WORDS(1) = "attack"
  WORDS(2) = "at"
  WORDS(3) = "dawn"
  ...
END MODULE VOCABULARY

PROGRAM DECIPHER_MESSAGE
  USE CHARACTER_VARYING
  USE VOCABULARY                     ! Get WORDS
  TYPE(VARYING) MSG                  ! To collect message
  INTEGER MSG_WDS(100)               ! Indices to WORDS
  INTEGER N, I                       ! No. of indices, loop index
  READ *, N, MSG_WDS
  MSG = ""                           ! Set to empty string
  DO I = 1, N

    MSG = MSG // WORDS(MSG_WDS(I))       ! Collect message
  END DO
  ...
END PROGRAM DECIPHER_MESSAGE
```

Suppose the string of indices read into MSG_WDS is 125, 2, 3005, 7, 333, 4002, 66, 222, 2, 6, 901 and at the conclusion of the loop, MSG contains "meeting at five on morning of may eighteen at abandoned mine". Dynamic memory will contain the following:

meeting
meeting at
meeting at five
meeting at five on
meeting at five on morning
...

Because the pointer VAR % CHARS in the subroutine VS_ASS_VS may be undefined, it cannot be tested for association so that space no longer needed can be released. In applications manipulating objects of derived type with pointer components, most memory leakage can be avoided if initialization of pointers in the derived type can be specified in the type definition. Objects of the type can then be initialized automatically when created by either declaration or allocation. If the module CHARACTER_VARYING can be specified as:

```
MODULE CHARACTER_VARYING
  TYPE VARYING
    PRIVATE
    CHARACTER, POINTER :: CHARS(:) => NULL( )
  END TYPE VARYING
```

```
! Generic interfaces for assignment, concatenation, conversion, etc.
        ...

      CONTAINS
        SUBROUTINE VS_ASS_VS (VAR, EXPR)       ! Assignment subroutine
          TYPE(VARYING), INTENT(OUT) :: VAR
          TYPE(VARYING), INTENT(IN) :: EXPR
          IF (ASSOCIATED (VAR % CHARS)) DEALLOCATE (VAR % CHARS)
          ALLOCATE (VAR % CHARS (1:SIZE(EXPR % CHARS)))
          VAR % CHARS = EXPR % CHARS
        END SUBROUTINE VS_ASS_VS
        ...
      END MODULE CHARACTER_VARYING
```

Then at the conclusion of the loop, dynamic memory would not be needlessly cluttered.

The previous standard allowed pointer assignment to occur when an object of a type containing a pointer is defined with a structure constructor; see the example in section 4.4.4 [37:17-26]. It is only a small extension to allow pointer nullification in derived-type constant expressions used for named constants or data initialization. For example:

```
TYPE LINK
   REAL :: VALUE
   TYPE(LINK), POINTER :: NEXT
END TYPE LINK

TYPE(LINK)                :: HEAD = LINK(0.0, NULL( ))
TYPE(LINK), PARAMETER     :: DEFAULT = LINK(0.0, NULL( ))
TYPE(LINK)                :: END_OF_CHAIN = DEFAULT
```

This language extension does not completely solve the memory leakage problem; for that, an automatic destructor is needed that would be invoked for local pointers and structures with pointer components when the procedure in which they are created terminates. Such a facility is not included in this standard; it could be provided automatically by a processor that strove to conserve dynamic memory.

For reasons of determinancy and portability, an object for which default initialization is specified is not allowed to appear in a DATA statement. In traditional implementations, a compiler passes along initialization information for nondynamic variables to a loader, which is frequently designed to handle object code from several different languages. There could thus be no guarantee that initialization in a DATA statement would override the default initialization specified in a type definition.

History: WG5-N930 Resolution of the Berchtesgaden WG5 Meeting
WG5-N932 Requirement for the Initialization of Pointers and Objects
X3J3/93-207 Pointer and Derived Type Initialization
X3J3/93-259 Proposal for Object Initialization
X3J3/94-031r2 Proposal for Object Initialization - moved to X3J3/009 with status “X3J3 consideration in progress” by unanimous consent
X3J3/94-138r3 Text for X3J3/009 re Object Initialization (B1) moved to X3J3/009 (replacing previous text) with status “Approved” by unanimous consent

---

Number: 007
Title: Language evolution (B9 Item B4.2)
Status: Incorporated in 94-007r2
Target date: 95
Last revision: May 94
X3J3 reference: x3j3.1994-310 email ballot

Introduction

At meeting 128, X3J3 approved the following two proposals.

1. The following Fortran 90 features shall be deleted from the revised language:
   - Real and double precision DO variables
   - Branching to END IF
   - PAUSE
   - ASSIGN, assigned GO TO, assigned formats
   - cH edit descriptor

2. The following Fortran 90 features shall be deemed obsolescent in the revised language:
   - Computed GO TO
   - Statement functions
   - DATA statements amongst executables
   - Assumed length character functions
   - Fixed form source
   - Assumed size arrays
   - CHARACTER* form of CHARACTER declaration

The following specifies the edits necessary to implement these decisions.

Proposed edits are represented using the form of the Technical Corrigenda, that is using location both relative to section and paragraph (for the benefit of those without line-numbered copies of Fortran 90) and by page and line number, shown in square brackets. The edits are ordered by topic, rather than sequentially overall, except that the replacement Annex B is presented as a whole, in order to facilitate consistency of style. Where comments are made, they are shown within square brackets.

Edits for the font of text describing obsolescent features have been proposed only for annexes A and C. It would be tedious in the extreme for Annex B, which discusses deleted and obsolescent features, to have every reference to an obsolescent feature in a different font.

Edits implementing deletion

D1. Edits relating to the deletion of real DO variables

In section 8.1.4.1.1, rule R821 [100:34-35]:
Change "scalar-numeric-expr" to "scalar-int-expr", three times.

In rule R822 [100:37]:
Change "scalar-variable" to "scalar-int-variable".

In rule R822, first constraint [100:38-39] :
Delete ", default real, or double precision real" ;

Delete the second constraint which follows rule R822 [100:40-41].

In section 8.1.4.4.1, third line, [102:22]:
Change "scalar-numeric-expr" to "scalar-int-expr", three times.

Replace the section numbered (1) [102:24-29] by:

"The initial parameter m1, the terminal parameter m2, and the incrementation parameter m3 are established by evaluating scalar-int-expr1, scalar-int-expr2, and scalar-int-expr3, respectively, including, if necessary, conversion to the kind type parameter of the do-variable according to the rules for numeric conversion (Table 7.9). If scalar-int-expr3 does not appear, m3 is of type default integer and its value is 1. The value m3 must not be zero."

In section 9.4.2, rule R918 [123:27-28]:
Change "scalar-numeric-expr" to "scalar-int-expr", three times.

In rule R918, second constraint [123:30] :
Delete ", default real, or double precision real" ;

Text for Annex B.1 is shown at B.1.1 below.

D2. Edits relating to the deletion of branching to END IF

In the second paragraph of section 8.1.2.2, second line [96:36]: Add "only" before "from within" and delete ", and also from outside the construct".

In the third paragraph of section 8.2, [107:7-8]:
Add "only" before "from within" and delete the text ", and also from outside the construct".
[This is then consistent with the wording for the CASE construct].

Text for Annex B.1 is shown at B.1.2 below.

D3. Edits related to deletion of the PAUSE statement

In section 2.1, rule R216 [9:31]:
Delete the line "or pause-stmt".

Delete all of section 8.5 [108:32-37].

Text for Annex B.1 is shown at B.1.3 below.

D4. Edits relating to deletion of ASSIGN, assigned GO TO, assigned formats

In section 2.1, rule R216 [9:29-30]:
Delete the lines "or assign-stmt" and "or assigned-go-to-stmt".

In section 8.1.4.1.2, rule R829, first constraint [101:20-21]: Add "or" before "an arithmetic-if-stmt" and delete "or an assigned-go-to-stmt".

In section 8.1.4.1.2, rule R833, first constraint [101:33-34]: Add "or" before "an arithmetic-if-stmt" and delete "or an assigned-go-to-stmt".

Delete section 8.2.4 [107:30-108:11] and renumber section 8.2.5 accordingly.

In section 9.4.1.1, rule R913 [121:5]:
Delete the line "or scalar-default-int-variable".
Delete the first paragraph following the constraint for rule R913 [121:8-9], that is delete, "The scalar-default-int-variable must have been assigned (8.2.4) the statement label of a FORMAT statement that appears in the same scoping unit as the format."

In section 14.7.5 [250:18-19]:
Delete the section numbered (6) and renumber accordingly.

In section 14.7.5 [250:33-34]:
In the section numbered (11) delete the text, ", except that variables associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed"

In section 14.7.6 [251:20-21]:
Delete the section numbered (2) and renumber accordingly.

Text for Annex B.1 is shown at B.1.4 below.

D5. Edits relating to the deletion of the cH edit descriptor

In section 10.2.1:
Delete the second line of rule R1016. [137:16]
Delete rule R1017 and the first three constraints following it. [137:17-20]
Delete the penultimate paragraph in the section, that is, "In the H edit descriptor, c specifies the number of characters following the H." [137:28]
In the last paragraph, delete the words, "except for the characters following the H in the H edit descriptor and". [137:30]

Delete section 10.7.2 [148:4-6].

Text for Annex B.1 is shown at B.1.5 below.

Edits implementing the obsolescence of features

O1. Edits related to making computed GO TO obsolescent

The following should be in obsolescent font:

In section 2.1, rule R216, the line "or computed-goto-stmt" [9:6];
Section 8.2.3 [107:21-29].

Text for Annex B.2 is shown at B.2.1 below.

O2. Edits related to making statement functions obsolescent

The following should be in obsolescent font:

In section 2.1, rule R207, the line "or stmt-function-stmt" [8:12];
In section 2.3.1, figure 2.1 the text "Statement Function Statements" [11:29];
In section 2.3.2, table 2.1, the text "Statement Function" [12:17];
In section 5.1, rule R505, second constraint, the text "or a statement function" [39:42];
In section 5.1.1.5, the paragraph, "The length specified for a character-valued statement function or statement function dummy argument of type character must be an integer constant expression." [43:6-7];
In section 11.3, rule 1106, second constraint, the text "a stmt-function-stmt," [157:21];
In section 11.3, penultimate paragraph, the text "statement function definitions," [157:27];
In section 12.1.2, the text ", or a statement function" [163:18];
In section 12.1.2.2.1, item numbered (2), the text "in a stmt-function-stmt," [164:4];
In section 12.1.2.2.1, item numbered (11), the text "or in a stmt-function-stmt," [164:15-16];
Section 12.1.2.4 [165:33-34];
In section 12.3.1, the sentence "The interface of a statement function is always implicit." [166:27];
In section 12.3.2.1, rule R1206, first constraint, the text ", or stmt-function-stmt" [167:29-30];
In section 12.4.1, rule R1214, fourth constraint, the text "or of a statement function" [172:7-8];
In section 12.5.2.4, first paragraph, the sentence "When a statement function is invoked, an instance of that statement function is created." [177:29 - added in corrigendum];
In section 12.5.2.4, second paragraph, the text "or statement function" (three times) [177:31-34];
In section 12.5.2.5, the entire paragraph "In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not appear in the expression of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement." [178:39-42];
Section 12.5.4 [182:1-32];

In section 14.1.2, item numbered (1), the text "statement functions," [241:24];
In section 14.1.2.4, item numbered (2)(b), the text "or statement function" [243:30-31];
In section 14.1.2.4.2, item numbered (3), the text "or statement function" [244:29];
In section 14.1.3, the entire first paragraph [245:16-18];
In Annex A, definition of dummy argument, the text ", or a statement function statement" [256:30];
In Annex A, definition of entity, the text "a statement function" [256:38];
In Annex A, definition of implicit interface, the text ", or a statement function" [257:33];
In Annex A, definition of procedure, the text ", or a statement function" [259:21];
In Annex A, all of the definition of statement function [260:21-22].

Text for Annex B.2 is shown at B.2.2 below.

O3. Edits relating to making obsolescent DATA statements among executables

The following should be in obsolescent font:

In section 2.1, rule 209, the line "or data-stmt" [8:17];
In section 2.3.1, figure 2.1, the text "DATA Statements" where it is alongside "Executable Constructs" [11:32-33];
In section 2.3.2, first paragraph, the text "DATA statement," [12:3].

[A paragraph in section 12.1.2.2.1 [164:23-25] also relates to positioning of DATA statements but does not need obsolescent font.]

Text for Annex B.2 is shown at B.2.3 below.

O4. Edits related to making obsolescent assumed length character functions

In section 5.1.1.5, rule R509, constraint, delete the text, "if the function is an internal or module function, array-valued, pointer-valued, or recursive" and insert, in obsolescent font, "unless it is an external or dummy function that is neither array-valued, pointer-valued, nor recursive" [42:36-37];

The following should be in obsolescent font:

In section 5.1.1.5, the paragraph numbered (3) [43:1-5];
In section 12.2.2, the last sentence, i.e. "If the length of a character data object is assumed, this is a characteristic." [166:16];
In section 12.3.1.1, paragraph (2)(d) [167:3-4];
In section 12.3.2.1.1, first paragraph, the text, "and the function result

must not have assumed character length" [169:18-19];
In Annex A, definition of characteristics item (5), the text ", and
whether the character length is assumed" [255:18];

Text for Annex B.2 is shown at B.2.4 below.

O5. Edits related to making fixed form source obsolescent

The following should be in obsolescent font:

In section 3.3, fourth paragraph, the texts "and fixed" and "and fixed
form" [21:39];
Section 3.3.2, including all subsections [23:23-24:14];
In section 4.3.2.1, in the definition of representable character, all of
the item numbered (1) [30:35-36];
In section 8.1, the pre-penultimate paragraph, all of the final sentence,
that is "In fixed source form, the name preceding the construct must be
placed after column 6." [95:20-21];
In section C.3.1, all of item (1) [264:28];
Section C.3.4 [265:5-33];
Section C.4.2, [265:37-40].
[the reasoning behind the last two is that if fixed source form did not exist,
the paragraphs would not exist; therefore the entire paragraphs should be in
obsolescent font.]

Text for Annex B.2 is shown at B.2.5 below.

O6. Edits related to making obsolescent assumed size arrays

The following should be in obsolescent font:

In section 5.1.2.4, rule R512, the line "or assumed-size-spec" [45:11];
Section 5.1.2.4.4 [47:1-29];
In section 6.2.1, third paragraph, second sentence [63:34-35];
In section 6.2.2, rule R621, second constraint [64:14-15];
In section 6.2.2.3.1, second paragraph [65:49];
In section 7.1.1.1, rule R702, second constraint [71:3];
In section 7.5.1.1, rule R735, constraint [89:12];
In section 9.4.2, rule R918, first constraint [123:29];
In section 12.2.1.1, last line, the text "size," [166:6];
In section 12.4.1.1, penultimate paragraph, the text "an assumed-size
array or" [173:27-28];
In section 12.4.1.4, the text "or assumed-size array" [174:16]; also the
last sentence [174:18:20]; [also change "assumed size" to "assumed-size"
as this is the only occurrence of the former.]
In section 13.8.1, first paragraph, last sentence [186:21-22];
In section 13.8.1, second paragraph, last sentence [186:24-25];
In section 13.13.95, definition of Argument, last sentence, [232:29-30];
In section 13.13.99, definition of argument ARRAY, last sentence,
[233:30-32];
In section 13.13.111, definition of argument ARRAY, last sentence,

[238:26-28];
In Annex A, all of the definition of assumed-size array [254:29-30];
In Annex A, definition of characteristics item (3), the text "size,"
[255:13].

Further, in the example in section 5.1.2.4, array S should be removed from the SUBROUTINE statement and the related REAL statement should be deleted since it is not policy to use obsolescent features in examples. [45:14 & 20]. Similarly the example in 5.2.5 [50:37] should be changed to be, say, ... C(:)

Text for Annex B.2 is shown at B.2.6 below.

O7. Edits related to making obsolescent CHARACTER*char-length

The following should be in obsolescent font:

In section 5.1.1.4, rule R507, the line "or * char-length[,]" [42:19].

[It appears that "*char-length" is used in section 5 where it is not clear without close study whether it refers to a type-spec or an entity-decl. However, there seems to be only the one place (rule R507) that requires obsolescent font.]

In section 5.1.1.5, in the last line, delete "*10" [43:12].

Text for Annex B.2 is shown at B.2.7 below.

Other related edits

In the Foreword, at the end of the second paragraph, [xii] add "The designation FORTRAN 66 is used to signify the first Fortran standard, ANS X3.9-1966 (published as USAS X3.9-1966) or ISO 1539-1972.";
In section C.9.7, last sentence, replace "ANSI X3.9-1966 (FORTRAN 66)" by "FORTRAN 66".

[This is because the term FORTRAN 66 is used in the Introduction (once), in Annex C several times, and is needed in the revised Annex B, but the only reference is hidden in C.9.7. Perhaps the bit about USAS is not needed - it is relevant only if a library has the standard catalogued under the title on the document. Fortran 66 was developed by an ASA committee but ASA was renamed USASI on August 24, 1966.]

In section 1.6, replace the first sentence [5:21-22] by "This International Standard protects the users' investment in existing software by including all but five of the language elements of Fortran 90 that are not processor independent.";
In the remainder of section 1.6 and subsections, replace "FORTRAN 77" by "Fortran 90" four times [5:24, 5:25, 5:28, 5:31];
In section 1.6, third sentence, replace "none" by "five" [5:23].

Delete the content of section B.1 [262:3-6] and replace by the following.

B.1 Deleted features

The deleted features are those features of Fortran 90 that are redundant and are considered largely unused. Section 1.6.1 describes the nature of the deleted features. The deleted features in this International Standard are:

(1) Real and double precision DO control variables
The ability present in FORTRAN 77, and for consistency also in Fortran 90, for a do-variable to be of type real or double precision in addition to type integer, has been deleted.

(2) Branching to an END IF statement from outside its block
In FORTRAN 77, and for consistency also in Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted.

(3) PAUSE statement
The PAUSE statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, has been deleted.

(4) ASSIGN and assigned GO TO statements and assigned format specifiers
The ASSIGN statement and the related assigned GO TO statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, present in FORTRAN 77 and Fortran 90, has been deleted.

(5) cH edit descriptor
In FORTRAN 77, and for consistency also in Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in FORTRAN 66; this has been deleted.

Recommendations are given in the following sections for those processors which extend the standard by implementing each of the deleted features.

[replacement text in subsections of B.1 below is copied from Fortran 90]

B.1.1 Real and Double Precision DO-Variables

Replace rules R821 and R822 in section 8.1.4.1.1 by the following:

"R821 loop-control is [,] do-variable = scalar-numeric-expr , n

n scalar-numeric-expr [, scalar-numeric-expr]

or [,] WHILE (scalar-logical-expr)

R822 do-variable is scalar-variable

Constraint: The do-variable must be a named scalar variable of type integer, default real, or double precision real.

Constraint: Each scalar-numeric-expr in loop-control must be of type integer, default real, or double precision real."

Replace the first part of section 8.1.4.4.1, up and excluding the paragraph numbered (2), by the following:

"When the DO statement is executed, the DO construct becomes active. If loop-control is

```
    [ , ] do-variable = scalar-numeric-expr1 , scalar-numeric-expr2
                    [ , scalar-numeric-expr3 ]
```

the following steps are performed in sequence:

(1) The initial parameter m1, the terminal parameter m2, and the incrementation parameter m3 are established by evaluating scalar-numeric-expr1, scalar-numeric-expr2, and scalar-numeric-expr3, respectively, including, if necessary, conversion to the type and kind type parameter of the do-variable according to the rules for numeric conversion (Table 7.9). If scalar-numeric-expr3 does not appear, m3 is of type default integer and its value is 1. The value m3 must not be zero."

B.1.2 Branching to an END IF statement from outside its IF block

In section 8.1.2.2, second paragraph, change the second sentence to be, "It is permissible to branch to an END IF statement from within the IF construct, and also from outside the construct." In section 8.2, change the third paragraph to read, "It is permissible to branch to an END IF statement from within its IF construct, and also from outside the construct."

B.1.3 PAUSE statement

The definition of the statement is:

```
        pause-stmt    is    PAUSE [ stop-code ]
```

Execution of a PAUSE statement causes a suspension of execution of the executable program. Execution must be resumable. At the time of suspension of execution, the stop code, if any, is available in a processor-dependent manner. Leading zero digits in the stop code are not significant. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed.

For completeness, "or pause-stmt" should be added to rule R216 in section 2.1.

B.1.4 ASSIGN and assigned GO TO statements and assigned FORMAT specifiers

The definitions of the ASSIGN and assigned GO TO statements are:

assign-stmt is ASSIGN label TO scalar-int-variable

Constraint: The label must be the statement label of a branch target statement or format-stmt that appears in the same scoping unit as the assign-stmt.

Constraint: scalar-int-variable must be named and of type default integer.

assigned-goto-stmt is GO TO scalar-int-variable [[,] (label-list)]

Constraint: Each label in label-list must be the statement label of a branch target statement that appears in the same scoping unit as the assigned-goto-stmt.

Constraint: scalar-int-variable must be named and of type default integer.

Execution of an ASSIGN statement causes a statement label to be assigned to an integer variable. While defined with a statement label value, the integer variable may be referenced only in the context of an assigned GO TO statement or as a format specifier in an input/output statement. An integer variable defined with a statement label value may be redefined with a statement label value or an integer value.

When an input/output statement containing the integer variable as a format specifier (9.4.1.1) is executed, the integer variable must be defined with the label of a FORMAT statement.

At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement label of a branch target statement that appears in the same scoping unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement in the same scoping unit as the assigned GO TO statement.

The execution of the assigned GO TO statement causes a transfer of control so that the branch target statement identified by the statement label currently assigned to the integer variable is executed next.

If the parenthesized list is present, the statement label assigned to the integer variable must be one of the statement labels in the list. A label may appear more than once in the label list of an assigned GOTO statement."

Further, "assigned-go-to-stmt" should be added to the lists of prohibited statements in the first constraints to rules R829 and R833 in section 8.4.1.1. For completeness, "assigned-stmt" and "assigned-go-to-stmt" should be added to rule R216 in section 2.1.

In section 14.7.5, the following section should be added: "Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value."

In section 14.7.5, the sentence in section (10???), "When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined." should have the following qualification added at the end, ", except that variables associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed"

In section 14.7.6, the following section should be added: "Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined."

Use of assigned integers as formats

In section 9.4.1.1 add to rule R913: "or scalar-default-int-variable" with the qualification that the scalar-default-int-variable must have been assigned the statement label of a FORMAT statement that appears in the same scoping unit as the format.

B.1.5 cH Edit Descriptor

In section 10.2.1, add the following line to rule R1016:
" or cH rep-char [rep-char] ..."

Add the following new rule with constraints, which logically follows rule R1016:

" c is int-literal-constant

Constraint: c must be positive.
Constraint: c must not have a kind parameter specified for it.
Constraint: The rep-char in the cH form must be of default character type."

In the H edit descriptor, c specifies the number of characters following the H.

If a processor is capable of representing letters in both upper and lower case, the edit descriptors are without regard to case except for the characters following the H in the H edit descriptor and the characters in the character constants.

Delete all of section B2, with subsections, [262:7-263:32] and replace by the following:

B.2 Obsolescent features

The obsolescent features are those features of Fortran 90 that are redundant and for which better methods are available in Fortran 90. Section 1.6.2 describes the nature of the obsolescent features. The obsolescent features in this International Standard are:

(1) Computed GO TO statement - see B.2.1
(2) Statement functions - see B.2.2
(3) DATA statements amongst executable statements - see B.2.3
(4) Assumed length character functions - see B.2.4
(5) Fixed form source - see B.2.5
(6) Assumed size arrays - see B.2.6
(7) CHARACTER* form of CHARACTER declaration - see B.2.7

B.2.1 Computed GO TO statement

The computed GO TO has been superseded by the CASE construct, which is a generalized, easier to use and more efficient means of expressing the same computation.

B.2.2 Statement functions

Statement functions are subject to a number of non-intuitive restrictions and are a potential source of error since their syntax is easily confused with that of an assignment statement.

The internal function is a more generalized form of the statement function and completely supersedes it.

B.2.3 DATA statements amongst executables

The statement ordering rules of FORTRAN 66, and hence of FORTRAN 77 and Fortran 90 for compatibility, allowed DATA statements to appear anywhere in a program unit after the specification statements. The ability to position DATA statements amongst executable statements is very rarely used, is unnecessary and is a potential source of error.

B.2.4 Assumed character length functions

Assumed character length for functions is an irregularity in the language since elsewhere in Fortran the philosophy is that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association.

This facility may be replaced by the use of a subroutine whose arguments correspond to the function result and the function arguments.

B.2.5 Fixed form source

Fixed form source was designed when the principal machine-readable input medium for new programs was punched cards. Now that new and amended programs are generally entered via keyboards with screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions 6, 7 or 72 on a line. Free form source was designed expressly for this more modern technology.

It is a simple matter for a software tool to convert from fixed to free form source.

B.2.6 Assumed-size arrays

Assumed-size arrays have been used for a wide variety of purposes in Fortran. The introduction of automatic arrays, assumed-shape arrays and deferred-shape arrays has given significantly increased functionality and allows the programmer to specify these requirements more directly and more securely.

B.2.7 CHARACTER* form of CHARACTER declaration

Fortran 90 had two different forms of specifying the length selector in CHARACTER declarations. The older form (CHARACTER*char-length) was an unnecessary redundancy.

Number: 008
Title: B9/B2 Conflicts with IEEE 754/854
Status: Incorporated in 94-007r3
Target date: 95
Last revision: May 94
X3J4 reference: 94-189r2

To: X3J3 94-189r2 page 1 of 2
From: Stan Whitlock
Subj: B9/B2 Conflicts with IEEE 754/854
Date: 13-May-1994

Rationale

These edits remove perceived conflicts in F90 with the IEC 559 (IEEE 754/854) standard.

Technical Description

If a processor supports IEC 559, -0. is distinct from +0. and certain computations like 0./0. or 0.**0. produce results, not errors. The edits below allow the SIGN intrinsic function to distinguish between -0. and +0., allow a -0 in formatted output, and do not disallow 0./0. or 0.**0..

Edits to the Standard

27:1-3 Delete this text.

27:11+ Add the text:

> The integer type includes a zero value, which is considered neither negative nor positive. The value of a signed integer zero is the same as the value of an unsigned integer zero.

28:29+ Add the text:

> The real type includes a zero value, which is generally considered neither negative nor positive. The value of a signed real zero is the same as the value of an unsigned real zero except, possibly, when used as the second argument to the SIGN intrinsic function.

80:1-3 Replace the first two sentences with:

> The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited.

80:1-3 Delete the word "also" from the third sentence.

139:23-24 Delete the sentence:

> However, the processor must not produce a negative signed zero in a formatted output record.

233:5 Replace the line with

> Result value.
> Case (i): If B > 0, the value of the result is |A|.
> Case (ii): If B < 0, the value of the result is -|A|.
> Case (iii): If B is of type integer and B=0, the value of the result is |A|.
> Case (iv): If the processor can distinguish between positive and negative real zero,
>
> If B is positive real zero, the value of the result is |A|.
> If B is negative real zero, the value of the result is -|A|.
>
> Otherwise, B is of type real, B=0, and the value of the result is |A|.

[Notes to Editor:

- "Result value." above is bold.
- "Case (i)", "Case (ii)", "Case (iii)", and "Case (iv)" above are italic.

]

3:3+ in the new section 1.4.1 (see 94-009 item #1), add a new item:

> (2) If the processor can distinguish between positive and negative real zero, the behavior of the SIGN intrinsic function when the second argument is negative real zero has been changed by this standard.

[End of proposal]

Number: 009
Title: CPU time
Status: Incorporated in 94-007r3
Target date: 95
Last revision: May 94
X3J3 reference: 94-170r5

To: X3J3 94/170r5
From: John Reid
Subject: B9/C1 - CPU time
Date: 19 May 1994

Rationale
This feature is provided to allow the assessment of what processor resources a piece of code consumed during execution.

Technical description
This feature is provided as an intrinsic subroutine that has a single argument in which the time in seconds is returned. A subroutine is chosen for consistency with DATE_AND_TIME and because all the intrinsic functions of Fortran 90 have a result that is uniquely determined by the values of the arguments.

Edits to the standard
188/4. Add: 'The subroutine CPU_TIME returns the processor time consumed during execution.'

192/16+ Add:
```
    CPU_TIME (TIME)           Processor time.
```

203/0+ Add:
13.13.24a CPU_TIME (TIME)

Description. Returns the processor time.

Class. Subroutine.

Argument. TIME must be scalar and of type real. It is an INTENT(OUT) argument that is set to a processor-dependent approximation to the processor time in seconds. If the processor cannot return a meaningful time, the value is set to zero.

Example.
```
    REAL T1,T2
     :
     CALL CPU_TIME(T1)
     :      ! Code to be timed.
     CALL CPU_TIME(T2)
     WRITE(*,*) 'Time taken by code was ',T2-T1,' seconds'
```
writes the processor time taken by a piece of code.

[Footnote: A processor for which a single result is inadequate (for example, a parallel processor) might choose to provide an additional

version for which TIME is an array.

The exact definition of the time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same computer or discover which parts of a calculation on a computer are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example.

Most computer systems have multiple concepts of time. One common concept is that of time expended by the processor for a given program. this may or may not include system overhead, and has no obvious connection to elapsed "wall clock" time.

End footnote]

Number: 010
Title: Nested WHERE
Status: Incorporated in 94-007r3
Target date: 95
Last revision: May 94
X3J3 reference: 94-154r2

To: X3J3
From: /parallel
Subject: A Second Proposal for Nested WHEREs.

---

1. Introduction.

The WHERE construct may contain only *assignment-stmts* in both both F90 and F95. It is the only F90 construct which cannot be named, (R215) and the only construct which cannot contain blocks.

F95 adds the FORALL construct in a similar category - it can be named, but it can't contain blocks. It can contain WHERE constructs and other FORALL constructs. This proposal allows WHERE constructs to be named and to contain FORALLs and other WHEREs.

The result is that the language is more regular, and that some nested IF constructs can be translated to array notation without forming complicated mask expressions..

2. Details.

2.1 Nested WHEREs

The proposal is to remove a restriction of Rule 739 that a *where-construct-stmt* may not contain a *where-construct-stmt.* That is a construct of the form

```
WHERE (mask1)
  vvv......
 WHERE (mask2)
    www....
  ELSEWHERE (mask3)
    xxx....
  ENDWHERE
ELSEWHERE
  zzz....
ENDWHERE
```

should be allowed. Thus, not only can WHERE constructs be nested but an ELSE WHERE, syntactically and semantically analogous to ELSE, has been introduced. The meaning may be understood by rewrite rules.

First rewrite every WHERE statement into a WHERE construct:

```
                                        WHERE(m)
WHERE (m) sss          becomes            sss
                                        END WHERE
```

Next eliminate all occurrences of ELSE WHERE(..)

```
WHERE (m1)                          WHERE (m1)
  xxx                                 xxx
ELSE WHERE (m2)        becomes      ELSE
  yyy                                   WHERE (m2)
END WHERE                                 yyy
                                        END WHERE
                                    END WHERE
```

note that xxx & yyy represent any sequences of statements, so long as the original WHERE , ELSEWHERE, and END WHERE match, and the ELSE WHERE is the first ELSE WHERE of the construct (that is yyy may contain additional ELSE WHERE(..) (or ELSEWHERE) statements of the construct.). Next, eliminate ELSEWHERE:

```
WHERE(m)                            temp  = m
  xxx                               WHERE (temp)
ELSE                   becomes          xxx
  yyy                               END WHERE
END WHERE                           WHERE (.NOT. temp)
                                      yyy
                                    END WHERE
```

Finally, eliminate nested WHERE constructs and statements:

```
WHERE (m1)                          temp = m1
  xxx                               WHERE (temp)
  WHERE (m2)                          xxx
    yyy                  becomes    END WHERE
  END WHERE                         WHERE (temp .AND. (m2))
  zzz                                 yyy
END WHERE                           END WHERE
                                    WHERE (temp)
                                      zzz
                                    END WHERE
```

2.2 WHEREs with nested FORALLs

A FORALL within a WHERE may be understood as a shorthand for an omitted enclosing FORALL which describes a mask to be applied to the inner FORALL. Thus, an example,

```
REAL  A(100,100), B(100,100),c(100)

WHERE (C .NE.0.0)
  C = 1.0/C
  FORALL (J=1:99) a(:,J) = (B(:,J) + B(J+1,:)) * C
END WHERE
```

is similar to

```
FORALL (J=1:100, C(I) .NE.0.0)
   C(I) = 1.0/C(I)
   FORALL (J=1:99) a(I,J) = (B(I,J) + B(J+1,I) * C(I)
END FORALL
```

The effect is to constrain all the *assignment-stmts* in the FORALL to be the same shape as the WHERE mask, just as usual. The arrays on the LHS of the FORALL will be of higher rank than the WHERE mask.

2.3 Edits to the Standard.

Page 93. Rules R739 through R743 are replaced by

R739a	*where-construct*	is	*where-construct-stmt* *[where-body-construct]...* *[masked-elsewhere-stmt* *[where-body-construct]...]...* *[elsewhere-stmt]* *[where-body-construct]...]...* *end-where-stmt*
R740	*where-construct-stmt*	is	*[where-construct-name:]* WHERE *(mask-expr)*
R740a	*where-body-construct*	is or or	*where-assignment-stmt* *where-stmt* *where-construct*
R740b	*where-assignment-stmt*	is or	*assignment-stmt* *forall-construct*
R741	*mask-expr*	is	*logical-expr*
R742a	*masked-elsewhere-stmt*	is	ELSEWHERE *(mask-expr)*
R742b	*elsewhere-stmt*	is	ELSEWHERE
R743	*end-where-stmt*	is	END WHERE *[where-construct-name]*

a rule number with the suffix 'a' or 'b' indicates a new or revised rule.

Page 93. 7.5.3.1, Add a new constraint

> ' If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding *end-where-stmt* must specify the same *where-construct-name*. If the *where-construct-stmt* is not identified by a *where-construct-name***,** the corresponding *end-where-stmt* must not specify a *where-construct-name*.'

Page 93. 7.5.3.2, 1st paragraph, (pink interpretation page) [93:21] add after '... arrays of the same shape.'

> ' If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt* or another *where-construct* then each *mask-expr* within the *where-construct* must be the same shape.'

Page 93. 7.5.3.2, 2nd paragraph [93:25] Replace the last sentence 'Each *assignment-stmt*...'with

> ' Each *where-construct-stmt* acts as if it has a controlling mask. Each statement within a *where-construct* is evaluated, in sequence, as if it were controlled by WHERE (control mask). The control mask for an outermost WHERE is the *mask-expr*.. The control mask of an inner WHERE is the conjunction of the *mask-expr* with the control mask of the enclosing *where-construct, masked-elsewhere-stmt, or elsewhere-stmt* . Within a *where-construct* if an ELSEWHERE is encountered the control mask becomes (.NOT. control mask) .AND. *mask-expr,* if any.'

page 93. 7.5.3.2, 5th paragraph. 1st sentence. Replace

'only a WHERE statement or a WHERE construct'

with

' only a WHERE statement or an outermost WHERE construct'

Page 93, 7.5.3.1 Throughout, replace

'*assignment-stmt'*

with

'*where-assignment-stmt*'

Page 93, 7.5.3.2 Throughout, replace

'*assignment-stmt'*

with

'*where-assignment-stmt*'

Add a footnote

' As the expressions within a *forall-construct* must conform with the *mask-expr* the array which receives the result of the forall will be of higher rank than the *mask-expr*.'

Number: 011
Title: Specification Functions
Status: Incorporated in 94-007r3
Target date: 95
Last revision: May 94
X3J3 reference: 94-173r2

94-173r2

To: X3J3
From: Len Moss
Subject: B9/C3 - Specification Functions

Note: This proposal makes use of the term "nonconstant specification expression" in the same way it is currently used in the standard (e.g., to distinguish automatic and nonautomatic data objects). However, this term is not explicitly defined in the standard and the /JOR subgroup believes that the implied definition, namely, a specification expression that is not also a constant expression, is incorrect in a number of contexts. As pointed out in 94-127, the formal definition of the term "constant expression" does not appear to serve any useful purpose: the critical definition is that for "initialization expression". As actually used in the document, the phrase "nonconstant specification expression" could better be defined as "a specification expression that is not also an initialization expression". The /JOR subgroup therefore recommends that one of the following changes be made: 1) provide somewhere (probably in 7.1.6.2) a formal definition of "nonconstant specification expression" in terms of initialization expressions; 2) change the term to something ("noninitialization specification expression"?) that better suggests the correct definition; or, 3) fix the concepts of constant and initialization expressions along the lines proposed in 94-127.

Proposal

Rationale:

A significant number of useful applications will be facilitated by the ability to perform more complicated calculations when specifying data objects than are permitted with Fortran 90. This will be very substantially achieved by allowing a restricted class of non-intrinsic functions in certain specification expressions.

Technical Description:

This feature permits a restricted class of user-defined functions, called "specification functions", to be referenced in nonconstant specification expressions. The restrictions are:

- the functions must be *pure*;
- the functions must not be internal;
- the functions must not be recursive;
- the arguments of such functions when used in specification expressions must be restricted expressions;
- the functions must not appear in any context which requires a constant specification expression, e.g., any specification not in a subprogram or an interface body, in a common or equivalence specification, or a component specification; and,
- any object whose specification depends on such a function is an automatic object and hence cannot be saved or initialized.

Edits:

5.1 [40/41]

Before "If" add the sentence,

> A specification expression is considered to be nonconstant if it involves a reference to a specification function.

7.1.6.2 [78/37+]

At the beginning of this section, add a new paragraph followed by a footnote:

> A function is a **specification function** if it is a *pure* function, is not an internal function and is not defined with the RECURSIVE keyword.
>
> {Note: Specification functions are nonintrinsic functions that can be used in specification expressions (7.1.6.2) to determine the attributes of data objects. The requirement that they be *pure* ensures that they cannot have side effects that could affect other objects being declared in the same specification-part. The requirement that they not be internal ensures that they cannot inquire, via host association, about other objects being declared in the same specification part. Some requirement against direct recursion is necessary: since specification expressions must be evaluated before the first executable statement, there would be no way to break such a recursion. Indirect recursion in specification functions appears to be possible but difficult to implement, and of little value to the user, and so there is a general prohibition against recursive specification functions.}

7.1.6.2 [79/16]

Insert before the last item in the numbered list (and renumber the last item) the following new item:

(10) A reference to a specification function where each argument is a restricted expression.

End Proposal

Number: 012
Title: Enable
Status: Incorporated in 94-007r2 but not approved
Target date: unspecified
Last revision: Aug 94
X3J3 reference: 94-258r4

To: X3J3 X3J3/94-258r4
From: John Reid
Subject: Enable proposal
Date: 19 August 1994

1. RATIONALE

If an operator invokes a process (for example, in hardware or in a procedure for a defined operator) and hits a problem with which it cannot deal, such as overflow, it needs to quit and ask the caller to do something else. A simple example of this proposal is

```
ENABLE (OVERFLOW)
   :
   ... = X*Y ...
HANDLE
   :
END ENABLE
```

If the multiply is intrinsic and an overflow occurs, a transfer of control is made to the block of code following the HANDLE statement. Similarly, if the multiply is a defined operator, it can be arranged that the OVERFLOW signals in comparable circumstances. The handle block may contain very carefully written code that is slow to execute but circumvents the problem, or may arrange for a graceful termination.

2. TECHNICAL SPECIFICATION

For dealing with exceptional events, this proposal involves the addition of integer-valued intrinsic conditions, a new construct, and some new statements. The intrinsic values are all positive, but negative values may be set by execution of a SIGNAL statement. For the definition of the conditions, see the proposed new section 15 at the end of this paper. Also, there are more examples in the proposed new sub-section 8.1.5.5.

The enable construct has the general form

```
enable statement
     [enable block]
[handle statement
     handle block]
end enable statement
```

Nesting of enable constructs is permitted. An enable or handle block may itself contain an enable-construct. Also, nesting with other constructs is permitted, subject to the usual rules for proper nesting of constructs.

The enable statement lists the names of the conditions to be signaled. If any of these conditions signals during the execution of the enable block, control is transferred to the handle block. A simple example is the following:

```
! Example A
ENABLE (OVERFLOW)
 ! First try a fast algorithm for inverting a matrix.
   :
HANDLE
 ! Fast algorithm failed; use slow one.
```

```
      :
   END ENABLE
```
Here, the code in the enable block takes no precautions against overflow and will usually execute correctly. Should it fail with overflow, the alternative algorithm is used instead.

The transfer to the handle block is imprecise in order to allow for optimizations such as vectorization. Any variable that is defined or redefined in a statement of the enable block becomes undefined. In Example A, a copy of the matrix itself would need to be available for the slow algorithm.

The transfer may be made more precise by adding within the enable block a nested enable construct without a handler. If any of the conditions is signaling when the inner enable statement is executed, control is transferred to the handle block. This reduces the imprecision to either the statements within the inner construct or those outside the inner construct. Adding such a construct to the code of Example A gives:

```
   ! Example B
   ENABLE (OVERFLOW)
   ! First try a fast algorithm for inverting a matrix.
   : ! Code that cannot signal overflow
      DO K = 1, N
         ENABLE
         :
         END ENABLE
      END DO
      ENABLE
      :
      END ENABLE
   HANDLE
   ! Alternative code which knows that K-1 steps have executed normally.
   :
   END ENABLE
```
Note that the enable, handle, and end-enable statements provide effective barriers to code migration by an optimizing compiler.

If there is no handler for a signaling condition (for example, if a condition signals outside any enable construct for the condition), a transfer of control as for a return statement takes place in a procedure or as for a stop statement takes place in a main program. The condition continues to signal.

When an enable statement is encountered, if any conditions that are enabled or are about to be enabled are signaling, a transfer of control to the next outer handler for a signaling condition (or a return or stop) takes place. This ensures that all enabled conditions are quiet on entering the enable block. Upon normal completion of the handle block, any of the handled conditions that is signaling is reset to quiet.

There is an option on the enable statement to specify that some of the conditions enabled are 'immediate'. Any <executable-construct> of the enable block that might signal one of the immediate conditions is treated as if it were followed by an enable construct with an empty body and no handler. An example of such an enable statement is
```
      ENABLE, IMMEDIATE (OVERFLOW)
```

For some conditions (mainly those that may require additional object code, for example, BOUND_ERROR), the processor is required to signal the condition only within the statements of the enable block. Whether such a condition signals outside any enable block for the condition is processor dependent. There is no requirement to signal such a condition in a procedure that is called from within an enable block.

There is an option on the handle statement to specify the handling of further

conditions. For example,

```
    HANDLE (ALL_CONDITIONS)
```

specifies that any condition that signals during the execution of the enable block be handled, including those that the processor handles outside enable blocks. These conditions, as well as those enabled, cause a transfer of control to an outer handler if they are signaling when an enable statement is encountered.

There is a facility for making a specified condition signal with a specified value. This is done with the SIGNAL statement. An example is

```
      SIGNAL(OVERFLOW, -3)
                ! Negative values of intrinsic conditions can be set this way.
```

It causes a transfer to the handler if in an enable block that has a handler for the condition; otherwise, it causes a return in a subprogram or a stop in a main program. This may also be used to set conditions quiet. For example,

```
   SIGNAL(ALL_CONDITIONS, 0)
```

sets all conditions quiet. In this case, there is no transfer of control.

In a handler, if it is desired to resignal the signaling conditions, this can be achieved with the pair of statements

```
   ENABLE
   END ENABLE
```

A transfer of control to the next outer handler for a signaling condition (or a return or stop) occurs without the values of the conditions changing.

There is a facility for finding the value of a condition. This is done with the CONDITION_INQUIRE statement. An example is

```
   CONDITION_INQUIRE(OVERFLOW, I)
```

which stores the value of the overflow condition in the variable I. Another form of the statement:

```
   CONDITION_INQUIRE(CHAR_ARRAY)
```

returns the names of the conditions that are signaling in the character array variable CHAR_ARRAY.

Each condition has a default integer value. The scoping rules for intrinsic conditions are as for intrinsic procedures. A future enhancement might allow the declaration of user conditions with scoping rules similar to those for variables.

If a condition is still signaling when the program stops, the processor must issue a warning on the default output unit.

Neither a handle statement nor an end-enable statement is permitted to be a branch target. A handle-block is intended for execution only following the signaling of a condition that it handles, and an end-enable statement is not a sensible target because it would permit skipping the handling of a condition.

Branching out of an enable construct is not permitted. This limits the extent of uncertainty over which statements have been executed when a handler is entered.

3. EDITS TO THE STANDARD

6/18+. Add

```
      IEC 559:1989, <Binary floating-point arithmetic for microprocessor
      systems>
      (also ANSI/IEEE 754-1985, IEEE standard for binary floating-point
      arithmetic).
```

.....

8/45+. Add

```
              <<or>>  <enable-construct>
```

.....

9/4-24. Add to R216 (in alphabetic positions) the lines
<<or>> <condition-inquire-stmt>
<<or>> <signal-stmt>
.....

12/50. After 'CASE constructs,', add 'ENABLE constructs,'.
.....

12/53+. Add:
(4) Execution of a signal statement (8.1.5.4) may change the execution sequence.
(5) Execution of an enable statement (8.1.5.1) may change the execution sequence.
.....

15/33+ Add

<<2.4.8 Condition>>

A <<condition>> is a default integer flag associated with the occurrence of an exceptional event. The value 0 corresponds to the quiet state and this is its initial value. Processor dependent nonzero values correspond to signaling states. Negative values can occur only through execution of the SIGNAL statement. The value may be found by execution of a CONDITION_INQUIRE statement.

[Footnote: The reason for specifying that conditions have integer values is that this leaves open the possibility of providing detailed information about the condition. This will be useful when a procedure (for example, in a library) signals a condition so that it can indicate the cause of the problem. The intrinsic values are forced to be positive so that a negative value can be seen to be created by source code and not by the system.]

[Footnote: Although multitasking is not part of Fortran 90, the interaction of this proposal with multitasking extensions has been considered. A model is that each virtual processor has a flag for each condition. For example, condition handling is permissible within a pure procedure. Enable, handle, and end-enable statements act as barriers at which the condition values are merged.]
.....

22/23+ Add to the Blanks Optional column:
END ENABLE
.....

67/39. After 'terminated', add 'unless the ALLOCATION_ERROR condition is enabled'.
.....

68/40. After 'terminated', add 'unless the DEALLOCATION_ERROR condition is enabled'.
.....

80/2. After 'program', add ', except in an enable block for a suitable condition'.
.....

95/10+ Add

(4) ENABLE construct
.....

95/19. Delete 'three'.

.....

107/0+. Add

<<8.1.5 Condition handling>>

A condition has a name with the same scoping rules as for intrinsic procedures and a value of type default integer. The value zero corresponds to the normal or 'quiet' state and nonzero values correspond to exceptional circumstances. All conditions have initial value zero. The processor is required to signal a condition if the associated circumstance occurs during execution of an intrinsic operation or an intrinsic procedure call specified in the scope of an enable block for the condition. Some conditions are also required to signal when the circumstance occurs outside an enable block, but whether other conditions signal outside an enable block is processor dependent. For the detailed specification, see Section 15. When the processor signals a condition, it has a positive value. The SIGNAL statement (8.1.5.4) may be used to give it a negative value.

[Footnote: For a condition whose signaling outside enable blocks is processor dependent, the control of whether the condition so signals is also processor dependent. There might be an option on the command line or there might be an intrinsic procedure that provides dynamic control. It is expected that by default the conditions UNDERFLOW and INEXACT will not signal except inside enable blocks.]

[Footnote: The proposal allows the in-lining of procedures with no change to the enable constructs. On some processors, this may cause a condition that does not signal outside enable blocks to signal.]

[Footnote: On many processors, it is expected that some conditions will cause no alteration to the flow of control when they signal and that they will be tested only when the enable block completes or another enable statement is encountered. Thus the overheads of testing the condition are confined precisely to the places where the programmer has requested a test. On other processors, this may be very expensive. They may instead cause a transfer of control to the handler (or a return or stop) as soon as the condition signals or soon thereafter.]

[Footnote: If additional code is needed (for example, to diagnose integer overflow), this is required only within the scope of the enable block.]

In a sequence of statements that contains no condition handling statements, if the execution of a process would cause a condition to signal but after execution of the sequence no value of a variable depends on the process, whether the condition signals is processor dependent. For example, when Y has the value zero, whether the code

```
          X = 1.0/Y
          X = 3.0
```

signals DIVIDE_BY_ZERO is processor dependent.

A condition must not signal if the signal could arise only during execution of a process not required by the standard. For example, the intrinsic LOG in the statement

```
          IF (F(X)>0.) Y = LOG(Z)
```

must not signal a condition when both F(X) and Z are negative and for the statement

```
          WHERE(A>0.) A = LOG (A)
```

negative elements of A must not cause signaling.

[Footnote: In general, it is intended that implementations be free within enable constructs to use the code motion techniques that they use outside enable constructs.]

<<8.1.5.1. The enable construct>>

The ENABLE construct specifies a (possibly empty) set of conditions, an enable block, and (optionally) a handle block with (optionally) a further set of conditions. The handle block is executed only if execution of the enable block leads to the signaling of one or more of the conditions.

```
R835a <enable-construct>    <<is>> <enable-stmt>
                                     [<enable-block>]
                                   [<handle-stmt>
                                       <handle-block>]
                                    <end-enable-stmt>

R835b <enable-stmt>         <<is>> [<enable-construct-name>:]          #
                                   # ENABLE [(<condition-name-list>)]   #
                                   # [,IMMEDIATE (<condition-name-list>)]

R835c <enable-block>        <<is>> <block>

R835d <handle-stmt>         <<is>> HANDLE [(<condition-name-list>)] #
                                   #  [<enable-construct-name>]

R835e <handle-block>        <<is>> <block>

R835f <end-enable-stmt>     <<is>> END ENABLE [<enable-construct-name>]
```

Constraint: If the <enable-stmt> of an <enable-construct> is identified by an <enable-construct-name>, the corresponding <end-enable-stmt> must specify the same <enable-construct-name>. If the <enable-stmt> of an <enable-construct> is not identified by an <enable-construct-name>, the corresponding <end-enable-stmt> must not specify an <enable-construct-name>. If the <handle-stmt> is identified by an <enable-construct-name>, the corresponding <enable-stmt> must specify the same <enable-construct-name>.

Constraint: A condition name must not appear more than once in an <enable-stmt>.

Constraint: A condition name must not appear more than once in a <handle-stmt>.

The conditions named on the enable statement are enabled during execution of the enable block. The set of conditions handled by the handle block consists of all those named on the enable statement or on the handle statement. If the enable construct is nested within an enable block, the conditions enabled for the outer block are also enabled for the inner block.

An <enable-stmt> may be a branch target statement (8.2).

[Footnote: Neither a handle statement nor an end-enable statement is permitted to be a branch target. A handle-block is intended for execution only following the signaling of a condition that it handles, and an end-enable statement is not a sensible target because it would permit skipping the handling of a condition.]

[Footnote: Nesting of enable constructs is permitted. An enable or handle block may itself contain an enable-construct. Also, nesting with other constructs is permitted, subject to the usual rules for proper nesting of constructs.]

Execution of an enable statement causes a transfer of control if a signaling condition is handled by the enable construct or any enable construct within

which it is nested. If the enable statement is nested in an enable block that has a handler for a signaling condition, the transfer is to the handler of the innermost such enable block. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program. The values of the conditions are not altered.

[Footnote: In an enable block, the pair of statements
```
       ENABLE
       END ENABLE
```
has a checking effect. If any handled condition is signaling, there will be a transfer of control to an outer handler (or a stop or return).The values of the conditions are not altered.]

[Footnote: Note that in a function subprogram it is very desirable to ensure that the function value is defined even if an error condition has been diagnosed and is expected to be handled in the calling subprogram. If the function value is not defined, further conditions will probably be signaled during the evaluation of the expression that gave rise to the function call, which may mask the condition that was the root cause.]

[Footnote: If a condition handled by a handler signals again during execution of the handler, this second signal will be indistinguishable from the first. If it is desired to handle it separately, it must be set to the quiet value and a nested enable must be provided.]

The value of each condition handled by the enable construct is set to the quiet value upon completion of execution of the <handle-block>.

<<8.1.5.2 Execution of an enable construct>>

Execution of an <enable-construct> begins with the first executable construct of the <enable- block>, and continues to the end of the block unless a handled condition is signaled. If a condition handled by the <enable-construct> signals outside any enable construct that handles the condition and is nested within the enable block, control is transferred to the <handle-block>. Transfer of control to the <handle-block> may take place on completion of execution of the enable-block or may take place sooner after the signaling of the condition. Any variable that might be defined or redefined by execution of a statement of the enable block outside any enable construct that handles the condition and is nested within the enable block is undefined, any pointer whose pointer association might be altered has undefined pointer association status, any allocatable array that might be allocated or deallocated may have been allocated or become unallocated, and the file position of any file specified in an input/output statement that might be executed is processor dependent.

[Footnote: The transfer to the handle block is imprecise in order to allow for optimizations such as vectorization. As a consequence, some variables become undefined. In Example 3 of 8.1.5.6, a copy of the matrix itself would need to be available for the slow algorithm.]

Branching out of an enable construct is not permitted. A CYCLE or EXIT statement is not permitted in an enable construct unless the do construct to which it belongs is nested within the enable construct. An alternate return specifier in an enable construct must not specify the label of a statement outside the construct. An ERR=, END=, or EOR= specifier in a statement in an enable construct must not be the label of a statement outside the construct. A RETURN or STOP statement is permitted in an enable construct. Conditions retain their values on execution of a RETURN or STOP statement.

[Footnote: The ban on branching out of an enable construct limits the extent of uncertainty over which statements have been executed when a handler is entered.]

Any <executable-construct> of the enable block that might signal one or more of the conditions in the immediate list on the enable statement is treated as if it were followed by an <enable-construct> with an empty enable block and no handler.

Execution of the <handle-block> completes the execution of the <enable-construct>.

If no condition handled by the enable construct is signaling on completion of execution of the <enable-block>, the execution of the entire construct is complete.

[Footnote: Nested enable constructs without handlers can be employed to reduce the imprecision of an interrupt. Note that enable, handle, and end-enable statements provide effective barriers to code migration by an optimizing compiler.]

<<8.1.5.3 Signaling conditions that are not enabled>>

A processor may signal a condition while executing a statement that is not in an enable block for the condition. If in a subprogram, a return is executed without alteration of the values of the conditions. If in a main program, a stop is executed and the processor must issue a warning on the default output unit.

[Footnote: On return to the caller, the condition will be signaling. If the invocation is within an enable block that has a handler for the condition, there will be a transfer to the handler (or a return or stop), but not necessarily until the execution of the block is complete. If the invocation is not within an enable block that has a handler for the condition, there may be a return (or stop) at once, or the processor may continue executing.]

<<8.1.5.4 Signal statement>>

R835g <signal-stmt> <<is>> SIGNAL (<condition-name>,<scalar-int-expr>)

Constraint: The <scalar-int-expr> must be of type default integer.

Constraint: If the condition name is that of a combination condition (15.7), the <scalar-int- expr> must be the literal constant 0.

The SIGNAL statement changes the value of the condition it names to that of the expression it contains. If the value is nonzero, it causes a transfer of control. If the statement is in an enable block of an enable construct that has a handler for the condition, the transfer is to the handler of the innermost such enable construct. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program.

[Footnote: In a handler, the pair of statements
```
       ENABLE
       END ENABLE
```
has a resignaling effect. If any handled condition is signaling, there will be a transfer of control to an outer handler (or a stop or return).The values of the conditions are not altered.]

<<8.1.5.5 Examples of ENABLE constructs>>

Example 1:

```
   MODULE MATRIX
! Module for matrix multiplication of real arrays of rank 2.
      INTERFACE OPERATOR(.mul.)
```

```
      MODULE PROCEDURE MULT
   END INTERFACE
CONTAINS
   FUNCTION MULT(A,B)
      REAL, INTENT(IN) :: A(:,:),B(:,:)
      REAL MULT(SIZE(A,1),SIZE(B,2)
      ENABLE (INTRINSIC, OVERFLOW)
         MULT = MATMUL(A, B)
      HANDLE
         SIGNAL(INEXACT, -1)
      END ENABLE
   END FUNCTION MULT
END MODULE MATRIX
```

This module provides matrix multiplication for real arrays of rank 2. Since the condition INSUFFICIENT_STORAGE signals outside enable blocks (see Section 15.1), if there is insufficient storage for the necessary temporary array, the module will signal the condition INSUFFICIENT_STORAGE. If an INTRINSIC or OVERFLOW condition occurs, the module will signal the condition INEXACT with value -1.

Example 2:

```
IO_CHECK: ENABLE (IO_ERROR, END_OF_FILE)
             :
             READ (*, '(I5)') I
             READ (*, '(I5)', END = 90) J
             :
       90    J = 0
          HANDLE
             CONDITION_INQUIRE(END_OF_FILE,K)
             IF (K/=0) THEN
                WRITE (*, *) 'Unexpected END-OF-FILE when reading ', &
                             'the real data for a finite element'
             ELSE
                CONDITION_INQUIRE(IO_ERROR,K)
                IF (K /= 0) WRITE (*, *) 'I/O error when reading ', &
                             'the real data for a finite element'
             END IF
             STOP
          END ENABLE IO_CHECK
```

In this example, if an input/output error occurs in either of the READ statements or if an end- of-file is encountered in the first READ statement, the appropriate condition will be signaled and the handler will receive control, print a message, and terminate the program. However, if an end-of-file is encountered in the second READ statement, no condition will be signaled and control will be transferred to the statement indicated in the END= specifier.

Example 3:

```
   ENABLE (USUAL)
    ! First try the "fast" algorithm for inverting a matrix:
      MATRIX1 = FAST_INV (MATRIX)
                        ! MATRIX is not altered during execution of FAST_INV.
   HANDLE
    ! "Fast" algorithm failed; try "slow" one:
      SIGNAL (USUAL, 0)
      ENABLE (USUAL)
         MATRIX1 = SLOW_INV (MATRIX)
      HANDLE
```

```
         WRITE (*, *) 'Cannot invert matrix'
         STOP
      END ENABLE
   END ENABLE
```

In this example, the function FAST_INV may cause a condition to signal. If it does, another try is made with SLOW_INV. If this still fails, a message is printed and the program stops. Note the use of nested enable constructs. Note, also, that it is important to set the signals to 'quiet' before the inner enable. If this is not done, a condition will still be signaling when the inner ENABLE is encountered, which will cause an immediate transfer to an outer handler (or a stop or return).

Example 4:

```
   ENABLE (OVERFLOW)
   ! First try a fast algorithm for inverting a matrix.
   : ! Code that cannot signal overflow
   DO K = 1, N
      ENABLE
      :
      END ENABLE
   END DO
      ENABLE
      :
      END ENABLE
   HANDLE
   ! Alternative code which knows that K-1 steps have executed normally.
   :
   END ENABLE
```

Here the code for matrix inversion is in line and the transfer is made more precise by adding to the enable block two enable constructs without handlers.

Example 5:

The following subroutine finds a zero of <f(x)> on an interval [<a,b>]. It is limited to take one second of real time as measured by the system clock. If it fails to obtain the requested accuracy after this time, the condition INEXACT signals with the value -1.

```
   SUBROUTINE ZERO_SOLVER (A, B, X, TOLERANCE, F)
      REAL A, B, X, TOLERANCE
      INTERFACE; REAL FUNCTION F(X); REAL X; END INTERFACE

      INTEGER  COUNT, RATE, START ! Local variables
      CALL SYSTEM_CLOCK(START, RATE)
      :
   ! The following code is executed every iteration
      CALL SYSTEM_CLOCK(COUNT)
   ! If time has run out, return, signaling condition INEXACT.
      IF (COUNT > START+RATE) SIGNAL (INEXACT,-1)
      :
   END SUBROUTINE ZERO_SOLVER
```

The application code handles the exception in a way that only it knows. An example is:

```
   :
   ENABLE
```

```
       CALL ZERO_SOLVER (A, B, X, TOLERANCE, F)
   HANDLE (INEXACT)

   ! Exceeded time limit. Fix up and go on.
      :
   END ENABLE
   :
```

Example 6:

```
      REAL FUNCTION CABS (Z)
        COMPLEX Z
! Calculate the complex absolute value, using a scaled algorithm
!   if  the straightforward calculation underflows or overflows. Set the
!   overflow condition to the value -1 if the result is too large to
!   be representable.

        REAL S, ZI, ZR
        INTRINSIC REAL, AIMAG, SQRT, ABS, MAX

        ZR = REAL(Z)
        ZI = AIMAG(Z)

quick: ENABLE(OVERFLOW, UNDERFLOW)

!         This is the quick and usual calculation.
          CABS = SQRT(ZR**2 + ZI**2)

        HANDLE quick

!         Will try again using a scaled equivalent method.
          S = MAX(ABS(ZR),ABS(ZI))
          SIGNAL (OVERFLOW,0) ; SIGNAL (UNDERFLOW,0)
  slow: ENABLE(OVERFLOW, UNDERFLOW)
            CABS = S*SQRT( (ZR/S)**2 + (ZI/S)**2 )
          HANDLE slow
            CONDITION_INQUIRE(OVERFLOW,K)
            IF (K/= 0) THEN
!             The result is too large to be representable.
              SIGNAL(OVERFLOW, -1)
            ELSE
              CONDITION_INQUIRE(UNDERFLOW,K)
              IF (K/= 0)  CABS = S
            END IF
          END ENABLE slow

        END ENABLE quick

      END FUNCTION CABS
```

This illustrates the setting of a special condition value when the problem really has a result that overflows.

Example 7:

```
      MODULE LIBRARY
      ...
      CONTAINS
         SUBROUTINE B
            ...
            X = Y*Z(I) ! No condition enabled.
```

```
            IF(X>10.)SIGNAL(OVERFLOW, 1)
            ...
         END SUBROUTINE B
      END MODULE LIBRARY

      SUBROUTINE A
         USE LIBRARY
         ENABLE
            CALL B
         HANDLE (OVERFLOW)
            ...
         END ENABLE
      END SUBROUTINE A
```

This illustrates the use of a library module that may signal the condition OVERFLOW. The signal statement causes a transfer to the handler in the calling subroutine A.

This also illustrates the effect of an intrinsic condition that is not enabled. An overflow in Y*Z(I) would cause OVERFLOW to signal and hence a transfer to the handler in the calling subroutine A. An out-of-range subscript value I might or might not signal BOUND_ERROR, but it would not be handled by subroutine A.

Example 8:

```
   ENABLE, IMMEDIATE (OVERFLOW)
          A = B*C
          WHERE(RAINING)
               X(:) = X(:)*A
          ELSEWHERE
               Y(:) = Y(:)*A
          END WHERE
    HANDLE
       .....
    END ENABLE
```

This illustrates the use of IMMEDIATE. The enable construct is equivalent to

```
      ENABLE (OVERFLOW)
          A = B*C
          ENABLE
          END ENABLE
          WHERE(RAINING)
               X(:) = X(:)*A
          ELSEWHERE
               Y(:) = Y(:)*A
          END WHERE
          ENABLE
          END ENABLE
    HANDLE
       .....
    END ENABLE
```

Note that the statements of a WHERE construct are not tested separately.

Example 9:

```
   SUBROUTINE LONG
      REAL, ALLOCATABLE A(:), B(:,:)
      : ! Other specifications
      ENABLE
          :
```

```
          ! Lots of code, including many procedure calls
          :
      HANDLE (ALL_CONDITIONS)
          ! Fix-up, including deallocation of any allocated arrays
          IF(ALLOCATED(A)) DEALLOCATE (A)
          IF(ALLOCATED(B)) DEALLOCATE (B)
          :
      END ENABLE
   END SUBROUTINE LONG
```

This illustrates the use of a handle statement with additional conditions. Here the enable block enables no conditions because fast execution is desired, but if anything goes wrong (for example, in one of the procedure invoked), fix-ups are performed, including deallocation of any local allocated arrays.

.....

107/5. After '<end-do-stmt,>' add 'an <enable-stmt>,'.
.....

122/17-18. Replace sentence by
   If an error condition (9.4.3) occurs during execution of an input/output statement that lies in an enable block for the IO_ERROR condition or contains an ERR= specifier:
.....

122/25. After 'continues with' add 'the handle block or'
.....

122/27-28. Replace sentence by
   If an end-of-file condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of an input/output statement that lies in an enable block for the END_OF_FILE condition or contains an END= specifier.
.....

122/34. After 'continues with' add 'the handle block or'
.....

122/37-38. Replace sentence by

If an end-of-record condition (9.4.3) occurs and no error condition (9.4.3) occurs during condition of an input/output statement that lies in an enable block for the END_OF_RECORD condition or contains an EOR= specifier:
 .....

123/6. After 'continues with' add 'the handle block or'
.....

125/10. Before 'contains' add 'is not in a enable block for the IO_ERROR condition and '.
.....

125/11. Before 'contains' add 'is not in a enable block for the END_OF_FILE condition and '.
.....

125/13. Before 'contains' add 'is not in a enable block for the END_OF_RECORD condition and '.
.....

241/25. After 'procedures,' add 'intrinsic conditions,'.
.....

241/35. After 'procedure,' add 'or condition'.

.....

<<15. CONDITIONS>>

In this section, the conditions supported by the standard and a statement for obtaining the value of a condition are specified.

The CONDITION_INQUIRE statement returns the value of a condition.

R835i <condition-inquire-stmt> <<is>> CONDITION_INQUIRE (<condition-name>, #
[STAT=]<scalar-default-int-variable>)
<<or>> CONDITION_INQUIRE (<conditions-array>)

835j <conditions-array> <<is>> <default-char-variable>

Constraint: The condition name must not be that of a combination condition (Section 15.7).

Constraint: The <conditions-array> must be a rank-one array that is not of assumed size.

The STAT= variable is defined with the value 0 if the condition named is quiet and a nonzero value otherwise. Negative values can occur only following execution of a SIGNAL statement.

The <conditions-array> is defined with the names of signaling conditions and blanks according to the rules of default assignment. If there are <s> conditions signaling, the first <s> elements are defined with the names of these conditions and the remaining elements are given the value blank. If the processor provides additional conditions, the names of the conditions defined by the standard must precede the names of any such additional intrinsics. If there are more signaling conditions than the size of the array, all elements are defined with condition names and which are chosen is processor dependent.

[Footnote: An array size 20 will always be adequate to return the names of all the conditions defined by the standard. If the final element of the character array has the value blank, the names of all signaling conditions will have been returned. If it is not blank, the user may set the conditions named quiet with SIGNAL statements and call CONDITION_INQUIRE again.]

<<15.1 Storage and addressing conditions>>

ALLOCATION_ERROR
This occurs when the processor is unable to perform an allocation requested by an ALLOCATE statement (6.3.1) containing no STAT= specifier. It is not signaled by an ALLOCATE statement containing a STAT= specifier. The signaling values are the same as the STAT values. Whether it signals outside enable blocks is processor dependent.

DEALLOCATION_ERROR
This occurs when the processor detects an error when executing a DEALLOCATE statement (6.3.1) containing no STAT= specifier. It is not signaled when executing a DEALLOCATE statement containing a STAT= specifier. The signaling values are the same as the STAT values. Whether it signals outside enable blocks is processor dependent.

INSUFFICIENT_STORAGE
This indicates that the processor is unable to find sufficient storage to continue execution. It may occur prior to the execution of the first executable statement of a main program or procedure and it may occur during the execution of an executable statement. It need not signal if ALLOCATION_ERROR signals. It signals outside enable blocks.

BOUND_ERROR
This occurs when an array subscript, array section subscript, or substring range violates its bounds. This does not include violations of the requirements derived from the size of an assumed-size array. Whether it signals outside enable blocks is processor dependent.

SHAPE
This occurs when an array operation or assignment does not conform in shape. Whether it signals outside enable blocks is processor dependent.

MANY_ONE
This occurs when a many-one array section (6.2.2.3.2) appears on the left of the equals in an assignment statement or as an input item in a READ statement. Whether it signals outside enable blocks is processor dependent.

NOT_PRESENT
This occurs when a dummy argument that is not present is accessed as if it were present; that is, when one of the restrictions of 12.5.2.8 is violated. Whether it signals outside enable blocks is processor dependent.

UNDEFINED
This occurs when a value that is required for an operation is detected by the processor to be undefined. Whether it signals outside enable blocks is processor dependent.

[Footnote: This wording is intended to allow the processor to be as thorough as it chooses with respect to the detection of undefined values.]

<<15.2 Input/output conditions>>

IO_ERROR
This occurs when an input/output error (9.4.3) is encountered in an input/output statement containing no IOSTAT= or ERR= specifier. It is not signaled when executing an input/output statement containing an IOSTAT= or ERR= specifier. The signaling values are the same as the IOSTAT values. Whether it signals outside enable blocks is processor dependent.

END_OF_FILE
This occurs when an end-of-file condition (9.4.3) is encountered in an input statement containing no IOSTAT= or END= specifier. It is not signaled when executing an input statement containing an IOSTAT= or END= specifier. Whether it signals outside enable blocks is processor dependent.

END_OF_RECORD
This occurs when an end-of-record condition (9.4.3) is encountered in an input statement containing no IOSTAT= or EOR= specifier. It is not signaled when executing an input statement containing an IOSTAT= or EOR= specifier. Whether it signals outside enable blocks is processor dependent.

<<15.3 Floating-point conditions>>

OVERFLOW
This condition occurs when the result for an intrinsic real or complex operation has a very large processor-dependent absolute value. Whether it signals outside enable blocks is processor dependent.

UNDERFLOW
This condition occurs when the result for an intrinsic real or complex operation has a very small processor-dependent absolute value. A processor that does not conform to IEC 559:1989 is required to set this condition when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

DIVIDE_BY_ZERO
This condition occurs when a real or complex division has a nonzero numerator and a zero denominator. Whether it signals outside enable blocks is processor dependent.

INEXACT
This condition occurs when the result of a real or complex operation is not exact. A processor that does not conform to IEC 559:1989 is required to set this condition when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

INVALID
This condition occurs when a real or complex operation is invalid. A processor that does not conform to IEC 559:1989 is required to set this condition for real or complex division of zero by zero and when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

[Footnote: It is expected that by default the conditions UNDERFLOW and INEXACT will not signal except inside enable blocks.]

<<15.4 Integer conditions>>

INTEGER_OVERFLOW
This condition occurs when the result for an intrinsic integer operation has a very large processor-dependent absolute value. Whether it signals outside enable blocks is processor dependent.

INTEGER_DIVIDE_BY_ZERO
This condition occurs when an integer division has a zero denominator. Whether it signals outside enable blocks is processor dependent.

<<15.5 Intrinsic procedure condition>>

INTRINSIC
This condition indicates that an intrinsic procedure or operation has been unsuccessful. An unsuccessful intrinsic procedure may signal other conditions instead of INTRINSIC. Whether it signals outside enable blocks is processor dependent. If an intrinsic procedure is an actual argument in a procedure call within an enable block for the INTRINSIC condition, the condition must signal if the procedure is invoked through the argument association.

<<15.6 System error conditions>>

SYSTEM_ERROR
This condition occurs as a result of a system error. Whether it signals outside enable blocks is processor dependent.

<<15.7 Combination conditions>>

Each of the following conditions may be specified on an enable, handle, or signal statement and is equivalent to specifying a list of conditions.

STORAGE
This condition is equivalent to the list: ALLOCATION_ERROR, DEALLOCATION_ERROR, and INSUFFICIENT_STORAGE.

IO
This condition is equivalent to listing all the input/output conditions.

FLOATING
This condition is equivalent to the list: OVERFLOW, INVALID, and DIVIDE_BY_ZERO.

INTEGER
This condition is equivalent to listing the two integer conditions.

USUAL
This condition is equivalent to the list: STORAGE, IO, FLOATING, and INTRINSIC.

ALL_CONDITIONS
This condition is equivalent to listing all the conditions.
signal if ALLOCATION_ERROR signals. It signals outside enable blocks.

Number: 013
Title: ELEMENTAL procedures
Status: Incorporated in 94-007r2
Target date: 95
Last revision: Aug 94
X3J3 reference: 94-245r3

X3J3/94-245r3

Elemental references to pure procedures
Revision of X3J3/94-245

Based on discussions at meeting 130, restrictions were added to non-intrinsic elemental functions that prohibit passing dummy procedures as arguments to elemental functions, prohibit passing elemental functions as dummy procedures, prohibit recursive elemental procedures, and added an example to chapter 14.

Rationale
Elemental functions provide the programmer with expressive power and the processor with additional opportunities for efficient parallelization.

Extending the concept of elemental procedures from intrinsic to both intrinsic and user-defined procedures is very much analogous to, but simpler than, extending the concept of generic procedures from intrinsic to both intrinsic and user-defined procedures. Generic procedures were introduced to intrinsic procedures in Fortran 77 and extended to user-defined procedures in Fortran 90. Elemental procedures were introduced to intrinsic procedures in Fortran 90 and, especially because of their usefulness in parallel processing, it is quite natural that they be extended in Fortran 95 to user-defined procedures.

Technical Description
The extension of elemental to user-defined procedures is straightforward. A minimal facility is proposed here, that involves a procedure's arguments being elemental.

A user-defined elemental procedure must be a pure procedure, having both the PURE keyword, and the ELEMENTAL keyword. All dummy arguments must be scalar and must not be pointers.

The actual arguments in a reference to an elemental procedure must all be conformable. Note that a scalar is conformable to any shape array, and thus any actual argument may be scalar; an actual argument must be scalar if it is associated with a dummy argument used in a specification expression in the procedure definition.

To be referenced elementally, the procedure must have an explicit interface. This allows there to be no change in the rules for how specific procedures differ and a simple change (addition) to the rules for resolving generic overloads: if there is no specific match the processor looks for an elemental match (see section 14.1.2.4.1).

One way of extending this feature in the future is to specify an ELEMENTAL attribute for any individual dummy argument(s) in lieu of the ELEMENTAL procedure keyword; the ELEMENTAL keyword included here may be considered to automatically give each dummy argument the elemental attribute.

Detailed Edits
section 7.1.3 - add a fifth paragraph
> A defined elemental operation is a defined operation for which the function is elemental (12.yyyy).

section 7.1.5 - change title

change "intrinsic" to "elemental"
section 7.1.5 - new first sentence
An elemental operation is an intrinsic operation or a defined elemental operation.
section 7.1.5 - second paragraph
change "intrinsic" to "elemental"
section 7.1.7 - penultimate paragraph
change "intrinsic binary" to "elemental binary"
section 7.1.7 - last paragraph
change "intrinsic unary" to "elemental unary"
section 7.3.1 - in item (5) replace "The" with
(a) The function is elemental, or
(b) The

section 7.3.2 - in item (5) replace "The" with
(a) The function is elemental and x1 and x2 are conformable, or
(b) The
section 7.5.1.3 - add the following sentence to the paragraph
A defined elemental assignment statement is a defined assignment statement for which the subroutine is elemental (12.yyyy).
section 7.5.1.6 - in item (5) replace "The" with
(a) The subroutine is elemental and either x1 and x2 have the same shape or
x2 is scalar, or
(b) The
section 7.5.1.6 - add as a last paragraph
If the defined assignment is an elemental assignment and the variable in the assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of variable and expr. If expr is a scalar it is treated as if it were an array of the same shape as variable with every element of the array equal to the scalar value of expr.
section 7.5.3.2 - third paragraph
change "elemental intrinsic operation" to "elemental operation"
section 8.1.1.2 - last sentence
change "12.4.2, 12.4.3, 12.4.4, 12.4.5" to "12.4.2, 12.4.4, 12.yyyy"
section 12.1.1 - last paragraph
replace "an intrinsic" with "a",
and replace the references with "12.yyyy"
section 12.2 - first paragraph (as modified by 94-149r2)
after "whether or not it is pure," add "whether or not it is elemental,"
section 12.3.1.1 - Add to list in part 2
f) The ELEMENTAL keyword
section 12.4.1.1 - third paragraph
change "12.4.3, 12.4.5" to "12.yyyy"
section 12.4.1.1 - insert phrase in last sentence of fourth-from-last paragraph
change "procedure is referenced" to "procedure is nonelemental and is referenced"
section 12.4.1.1 - insert phrase in last sentence
change "dummy argument" to "dummy argument of a nonelemental procedure"
section 12.4.2 - add to the paragraph
A reference to an elemental function (12.yyyy) is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.
section 12.4.3 - delete entire section
section 12.4.4 - add to the paragraph
A reference to an elemental subroutine (12.yyyy) is an elemental reference if all actual arguments corresponding to INTENT(OUT) and INTENT(INOUT) dummy arguments are arrays that have the same shape and the remaining actual arguments are conformable with them.
section 12.4.5 - delete entire section
section 12.5.2.2 - add to syntax rule R1217a (as modified by 94-149r2)

or ELEMENTAL
Constraint: If ELEMENTAL is present, PURE must be present.

Constraint: If ELEMENTAL is present, RECURSIVE must not be present.

section 13.1 - first paragraph
change "elemental function" with "elemental intrinsic function"

section 13.2 - replace sections 13.2.1 and 13.2.2 with
Elemental intrinsic procedures behave as described in 12.yyyy.

section 14.1.2.4.1 - insert a new rule (2), and renumber accordingly
(2) If (1) does not apply, if the reference is consistent with an elemental reference to one of the specific interfaces of an interface block that has that name and either is contained in the scoping unit in which the reference appears or is made accessible by a USE statement contained in the scoping unit, the reference is to the specific elemental procedure in that interface block that provides that interface. Note that the rules in 14.1.2.3 ensure that there can be at most one such specific interface.

[footnote 1]

[footnote 1. These rules allow specific instances of a generic function to be used for specific array ranks and a general elemental version to be used for other ranks. Given an interface block such as:

```
INTERFACE  RANF

      ELEMENTAL  FUNCTION SCALAR_RANF(X)
      REAL  X
      END

      FUNCTION VECTOR_RANDOM(X)
      REAL  X(:)
      REAL  VECTOR_RANDOM(SIZE(X))
      END

END  INTERFACE  RANF
```

and a declaration such as:

```
REAL  A(10,10), AA(10,10)
```

then the statement

```
A = RANF(AA)
```

is an elemental reference to SCALAR_RANF. The statement

```
A(1,1:5) = RANF(AA(6:10,2))
```

is a non-elemental reference to VECTOR_RANDOM.]

section 14.1.2.4.1 - new item (3)
change ""If (1) does" to "If (1) and (2) do"

section 14.1.2.4.1 - new item (4)
change ""If (1) and (2) do" to "If (1), (2), and (3) do"

section 14.1.2.4.1 - new item (5)
change ""If (1), (2), and (3) do" to "If (1), (2), (3), and (4) do"

annex A - elemental
remove the word "intrinsic"
and replace the references with "12.yyyy"

section 12.yyyy - immediately after 12.xxxx (Pure procedures)
12.yyyy Elemental procedures

12.yyyy.1 Elemental procedure declaration and interface

An elemental procedure is an elemental intrinsic procedure or a procedure that is defined with the prefix-spec ELEMENTAL.

Procedures defined with the keyword ELEMENTAL must satisfy the additional constraints:

Constraint: All dummy arguments must be scalar and must not have the POINTER attribute.

Constraint: For a function, the result must be scalar and must not have the POINTER attribute.

Constraint: A dummy-arg must not be *.

Constraint: A dummy-arg must not be a dummy procedure.

Constraint: A procedure with the elemental keyword must not be used as an actual argument.

Note that an elemental procedure is a pure procedure and all of the constraints for pure procedures also apply.

12.yyyy.2 Elemental function arguments and results

<this paragraph is largely taken from what was originally section 13.2.1>
If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the argument with the greatest rank. If the arguments are all scalar, the result is scalar. For those elemental functions that have more than one argument, all arguments must be conformable. In the array-valued case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar-valued function had been applied separately, in any order, to corresponding elements of each argument. For an intrinsic function, an argument called KIND must be specified as a scalar integer initialization expression and must specify a representation method for the function result that exists on the processor. For a non-intrinsic function, an actual argument must be scalar if it is associated with a dummy argument that is used in a specification expression.

<this example was in the original section 12.4.3>
An example of an elemental reference to the intrinsic function MAX:
if X and Y are arrays of shape (m, n),
MAX (X, 0.0, Y)
is an array expression of shape (m, n) whose elements have values
MAX (X (i, j), 0.0, Y (i, j)), i = 1, 2, ..., m, j = 1, 2, ..., n

12.yyyy.3 Elemental subroutine arguments

<this paragraph was originally section 13.2.1, with "intrinsic" deleted>
An elemental subroutine is one that is specified for scalar arguments, but in a generic reference may be applied to array arguments. In a reference to an elemental subroutine, either all actual arguments must be scalar, or all INTENT (OUT) and INTENT (INOUT) arguments must be arrays of the same shape and the remaining arguments must be conformable with them. In the case that the INTENT (OUT) and INTENT (INOUT) arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained if the subroutine with scalar arguments were applied separately, in any order, to corresponding elements of each argument.

Number: 014
Title: Automatic deallocation
Status: Incorporated in 94-007r2
Target date: 95
Last revision: Aug 94
X3J3 reference: 94-270r3

X3J3/94-270r3

Subject: Automatic deallocation of ALLOCATABLE objects
References: WG5-N930, Resolutions of the Berchtesgaden WG5 Meeting, B9
WG5-N931, Requirements for Allowing Allocatable derived-type Components

Revision History: 94-270r1, examples added, initial status text moved here from 94-269
94-270, specific edits added
94-211, original presentation, May 1994 (meeting 129)

Requirement Title: B9/B3 Allocatable derived-type components

Status: For consideration

Technical Description:
Require automatic deallocation of unSAVEd allocatable objects on scope exit.

Motivation:
Currently, the standard does not provide for automatic deallocation of allocatable objects, even when they are local non-static variables. When such objects go out of scope a memory leak can occur.

The general handling of ALLOCATABLE is being regularised; removing surprises is part of the task. Further, the general thrust of the Fortran committee is to work to eliminate memory leakage opportunities.

The main reason for the current situation appears to be a concern about performance. The addition of automatic (i.e. managed by the processor rather than the user) deallocation will not have a major performance impact. The actual deallocation is what takes time, and that must occur in any event if a memory leak is to be avoided. The proposed change merely adds a check on whether the deallocation is required -- a few instructions at most, compared with dozens of instructions for the actual deallocation.

Note further that we have already accepted the "overhead" of explicit "initialisation" of the allocation state of allocatable variables. This initialisation was necessary to bring a semblance of order into the allocation status. By adding automatic deallocation, we not only remove a source of user annoyance and surprise, but also simplify the language definition. Currently, there must be a third ("undefined") allocation state for allocatable objects and fairly confusing words about the consequences of going out of scope with a non- deallocated object.

Finally, as the language currently stands it is impossible to create opaque data types which need a variable amount of storage without the possibility of leaking memory. In conjunction with the allocatable component proposal (94- 269r2) automatic deallocation of allocatable objects provides the user with this capability in a form which is safe to use.

Detailed EDITS:

Subclause 6.3.3.1
[69:12-15] {allocation status} {{the third and fourth paragraphs of 6.3.3.1}}
{{specify we lose the old allocatable array on each new instantiation if not SAVEd}}

replace with:
"Any other allocated allocatable array that is a local variable of a procedure, is not a subobject of a pointer, is not accessed by use association, is not part of a dummy argument, is not part of a function return value, and is not a variable accessed from the host scoping unit is deallocated (as if by a DEALLOCATE statement)."

[69:13+] {allocation status}
{{allow modules to always SAVE their allocatable arrays, or to always initialise them (e.g. if they are implemented via overlay segments), but do not penalise the user by forbidding subsequent access}}
add a new paragraph:
"The allocation status of any other allocatable array that is a local variable of a module is processor-defined; the ALLOCATED intrinsic may be used to determine whether the array is still currently allocated or has been deallocated."

[69:15+] {deallocation action}
{{Here we define deallocation of an object containing allocatable components to deallocate any of these which happen to be allocated. Since this is described recursively, a nested tree of allocatable components will be deallocated bottom-up. Note that a system which stores allocatable objects on a stack or which performs automatic garbage collection already satisfies this definition.}}
insert new paragraph:
"When a derived-type object is deallocated, any ultimate allocatable components that are currently allocated are deallocated.

[Footnote: in the following example:

```
    MODULE USER
        TYPE, PRIVATE :: VARYING_STRING
            CHARACTER,ALLOCATABLE :: VALUE(:)
        END TYPE
        TYPE USER
            PRIVATE
            TYPE(VARYING_STRING) NAME
            TYPE(VARYING_STRING),POINTER :: DETAILS(:)
        END TYPE
        ...
    END MODULE
    SUBROUTINE PROCESS_ONE_USER
        USE USER
        TYPE(USER) X
         CALL READ_USER(X)    ! Read user details into X
        ...                  ! Process the user
    END SUBROUTINE
```

on return from PROCESS_ONE_USER, X does not have the SAVE attribute and so X%NAME%VALUE is deallocated. However, X%DETAILS has the POINTER attribute and so its target and any of the target~s components are not automatically deallocated.]

Subclause 14.8
[252:37-253:4] {{remove undefined allocation status from the possibilities, to avoid never-never land}}
replace existing text with:
"(1) Not currently allocated. An allocatable object with this status must not be referenced, defined, or deallocated; it may be allocated with the ALLOCATE statement. The ALLOCATED intrinsic returns .FALSE. for such an object.
(2) Currently allocated. An allocatable object with this status may be referenced, defined, or deallocated; it must not be allocated. The ALLOCATED intrinsic returns .TRUE. for such an object.

An allocatable array with the SAVE attribute has an initial status of not currently allocated. An ALLOCATE statement changes this status to currently allocated; it then remains currently allocated until execution of a DEALLOCATE statement.

An allocatable array component of a derived-type variable that has the POINTER attribute, or is a subobject of a pointer component, has no initial allocation status, because it does not exist until brought into existence by the ALLOCATE statement (or referred to by the pointer assignment statement).

Any other allocatable array without the SAVE attribute that is a local variable of a procedure, is not accessed by use association, is not part of a dummy argument, and is not a variable accessed from the host scoping unit has a status of not currently allocated at the beginning of each invocation of the procedure. During execution of the procedure its status may be changed by execution of ALLOCATE and DEALLOCATE statements. On exit from the procedure by execution of a RETURN or END statement, if such an allocatable array is not part of a function return value and has the status of currently allocated, it is deallocated (as if by a DEALLOCATE statement).

Any other allocatable array without the SAVE attribute that is a local variable of a module has an initial status of not currently allocated. If the array has an allocation status of currently allocated on execution of a RETURN or END statement resulting in no executing scoping unit having access to the module it is processor-defined whether the array~s allocation status remains currently allocated or the array is deallocated (6.3.3.1) as if by a DEALLOCATE statement."

add to Rationale Section:

6.3.3.1 Deallocation of allocatable arrays

Automatic deallocation of allocatable arrays (including allocatable array components) provides the user with a safe method of creating opaque data types of variable size which do not leak memory. It also removes the burden of manual storage deallocation both for simple allocatable arrays and for allocatable components of non-opaque types.

The "undefined" allocation status of Fortran 90 meant that an allocatable array could easily get into a state where it could not be further used in any way whatsoever, it could not be allocated, deallocated, referenced, defined, or even used as the argument to the ALLOCATED function. Removal of this status provides the user with a safe way of handling allocatable arrays which he does not desire to be SAVEd, permitting use of the ALLOCATED intrinsic function to discover the current allocation status of the array at any time.

Number: 015
Title: Allocatable components
Status: Incorporated in 94-007r2 but not approved
Target date: U
Last revision: Aug 94
X3J3 reference: 94-269r2

X3J3/94-269r2

Subject:Text for X3J3/009 re: Allocatable Components in Structures (B3)
References:WG5-N930, Resolutions of the Berchtesgaden WG5 Meeting, B9
WG5-N931, Requirements for Allowing Allocatable derived-type
Components

Revision history: 94-269r2, resolve comments
94-269r1, move most initial status to 94-270, fix object init
conflict, editorial fixes
94-269, specify initial status, use "ultimate component"
classification
WG5-N1040, revised to resolve received comments
94-202r2 remove allocatable entities from COMMON, do not
specify a storage unit; clarify/simplify constructors
(revised post-meeting per meeting 129 comments)
94-202r1 provide restriction for entity-decls, add constructors
94-202 original presentation, May 1994 (meeting 129)

Requirement Title: B9/B3 Allocatable derived-type components

Status:X3J3 consideration in progress

Technical Description:
Currently, the ALLOCATABLE attribute is limited to local named array data entities. One cannot have allocatable components of derived-type objects, nor allocatable dummy arguments or function return values. There is a requirement from WG5 that these restrictions be removed, that ALLOCATABLE be extended and regularized.
In addition, if an ALLOCATABLE object is still allocated when it goes out of scope, its allocation status becomes undefined. There is a desire that the deallocation be provided "automatically" by the processor. That is addressed in a separate paper (X3J3/94-270r2).
The current paper provides a major step towards ALLOCATABLE arrays as full, first-class entities. It provides for them to appear as components of derived types, as per WG5-N931, "Requirements for Allowing Allocatable Derived-type Components".
Still outstanding from the general request:
- ALLOCATABLE dummy arguments and function return values
- ALLOCATABLE strings
- ALLOCATABLE derived types (needs Parameterized Derived Types)
While these are mostly fairly straightforward, a large number of edits appear to be required. Due to the shortage of time, we recommend that this development be undertaken for Fortran 2000.
Per direction from X3J3, this proposal does not provide for allowing ALLOCATABLE objects, nor derived type objects with ALLOCATABLE components in COMMON or EQUIVALENCE.

Motivation:
ALLOCATABLE provides functionality which shares much with that of the POINTER features. Indeed, the underlying implementation is probably nearly identical, with one important difference. The address pointer implicit in ALLOCATABLE objects is not accessible to the user. ALLOCATABLE objects, therefore, cannot be aliased with other objects as a result of pointer

assignments. For this reason, they behave much better than pointers with respect to optimization.
Given the benefits of ALLOCATABLE, it appears advisable to extend the facility to handle derived type components as well as normal variables. The extension introduces little additional complexity into either the language definition or implementation; in fact, one can argue that the generalization serves to remove an arbitrary restriction from the language.
The duties of the implementor follow directly from current practice.
If a local object has the ALLOCATABLE attribute, the implementation must arrange to set the object to non-allocated status at the time that the object is created. This duty is now extended to components of derived type objects.
In addition, if a derived type object with an ALLOCATABLE component is itself created via an ALLOCATE statement, the component must be set to non-allocated status as part of the creation.
There exists some complication with regards to constructors. There is no "null" value for an allocatable object, so the constructor must specify an actual value set for an allocatable component. (This will generally be an array constructor with an implied DO to provide the right number of elements.)
(Note that, even though "assignment semantics" apply, this value cannot be given as a scalar, as there is no size given in the type definition.)
The value of the constructor will include the allocatable component in an "allocated" state with the indicated contents. Note that the rules of assignment for allocatable arrays involve copying the contents, not the implicit pointer. Thus, if the constructor is used in an assignment statement, the allocatable components of the target variable must already be allocated.
Lastly, because allocation cannot be done prior to execution, allocatable components cannot be initialized in DATA statements or with = initialization.
(Note that some of these conditions may be changed slightly if we approve the proposal for constructors with keyword and optional fields.)

Detailed EDITS:
{{note: edits are marked with a brief comment about the particular
                subject area being addressed. In some cases, a subject area
                is relevant but requires no edits; in such cases, a null edit
        (plus comment) is given. }}

[throughout]{general nomenclature}
{no problem. The Standard references the term "allocatable array"
         throughout. There is no problem, in general, with this terminology
         applying equitably to either data entities or to components; so no
         pervasive change is required.}

Subclause 4.4
[32:28] {make allocatable components ultimate}
after: "Ultimately, a derived type is resolved into ultimate
               components that are"
change:"either of intrinsic type"
to:"of intrinsic type, are allocatable,"

Subclause 4.4.1
[33:20] {definition of component-attr-spec}
after:"R427 component-attr-spec   is   POINTER"
add new line:
"   or ALLOCATABLE"

[33:26]
add a new constraint following the third constraint of R427:
"Constraint: POINTER and ALLOCATABLE must not both appear in
                             the same component-def-stmt."

[33:26+] {{WG5: Since allocatable components prohibit appearance in
                storage-associated contexts, do not allow sequence types to

contain allocatable components.}}
add a new constraint following the above:
"Constraint: The ALLOCATABLE attribute must not be specified if
SEQUENCE is present in the derived-type-def."

[33:31-32] {WG5(reworded):dimension info} in the first constraint of R429,
change:"If the POINTER attribute is not specified"
to: "If neither the POINTER attribute nor the ALLOCATABLE attribute is
specified"

[33:33] {WG5(reworded):dimension info}
in the second constraint of R429,
change: "POINTER attribute" to "POINTER attribute or the ALLOCATABLE
attribute"

[33:38+] {default initialization, per 94-138}
after:"the POINTER attribute must not appear"
add:"and the ALLOCATABLE attribute must not appear"

[33:39] {sequence types}
{{WG5: No changes needed. A sequence type cannot have allocatable
components.}}

[34:2] {when one needs to have "::" in a component definition}
change:"or both"
to"the ALLOCATABLE attribute, or any combination thereof"

[35:35+] {edits to object initialization proposal in 94-009.006}
in paragraph 3 (immediately following the first example in this edit),
after:"If a component is of intrinsic type and is not"
insert:"allocatable or"
{{before "a pointer, a default initial value may be specified..."}}
later in the same paragraph,
after:"If the component is of derived type and does not have the"
replace:"pointer attribute"
with:"ALLOCATABLE attribute or POINTER attribute"

[35:35+] {examples}
add new paragraphs:
"A derived type may have a component that is allocatable. For example:

```
TYPE STACK
INTEGER :: INDEX
INTEGER, ALLOCATABLE, DIMENSION(:) :: CONTENTS
END TYPE STACK
```

For each variable of type STACK, the shape of the component CONTENTS
is determined by execution of an ALLOCATE statement."

Subclause 4.4.4
[37:28] {constructors}{{WG5: Interaction with 94-009.006 in last sentence of
this addition}}
Add, at the end of this subclause, new paragraphs:
"Where a component of the derived type is an allocatable array, the corresponding constructor expression must evaluate to an array. The value of the constructor will have a component that is allocated (has an allocation status of allocated), and with contents as given by the constructor expression. Note that the allocation status of the allocatable component is available to the user program if the constructor is associated with a dummy argument, but generally not for other uses. Note, also, that when the constructor value is used in assignment, the corresponding component of the target must already be allocated.

Where a derived type contains an ultimate allocatable component, the

constructor must not appear as a data-stmt-constant in a DATA statement
(5.2.9), as an initialization-expr in an entity-decl (5.1), or as an
initialization-expr in a component-initialization (4.4.1)."

Subclause 5.1
[40:5] {=initialization}
after:"an allocatable array,"
add:"a derived-type object containing an ultimate allocatable
                component,"

[40:20] {addition to constraint so it is clear they cannot be initialised at
         compile time}
after:"allocatable arrays,"
add: "derived-type objects containing an ultimate allocatable
              component,"

Subclause 5.1.2.3
[44:24+] {edit to interpretation edit, paper 94-274r1}
change:"a stat-variable"
to:"an allocate-object or stat-variable"

Subclause 5.1.2.4.3
[46:8] {allocatable arrays are no longer necessarily named}
in the first line of the second paragraph, change: "a named array" to
        "an array"

[46:10] {shape of an allocatable component can also be determined by a
         structure constructor}
add to end of sentence:
"or evaluation of a structure constructor for a derived type with an
         allocatable component"

[46:11] {where & how you can declare ALLOCATABLEs}
after:"in a type declaration statement"
add:", a component definition statement,"

[46:13] {ditto}
after:"in a type declaration statement,"
add:"a component definition statement,"

[46:15] {ditto}
after:"type declaration statement"
add:"or a component definition statement"

Subclause 5.1.2.9
[48:24] {storage units, sequence}
{WG5: This change has been deleted as unnecessary since derived type
              objects containing allocatable components cannot appear in
              COMMON or EQUIVALENCE}.

Subclause 5.2.9
[52:34] {DATA statement restrictions}
change:"or an allocatable array"
to: "an allocatable array, or a derived-type object containing an
                ultimate allocatable component"

Subclause 5.4
[56:11] {WG5: do not allow allocatable components in NAMELIST}
change:", or an allocatable array"
to:", an allocatable array, or a derived-type object containing
                  an ultimate allocatable component"
Subclause 5.5.1
[57:1] {WG5: do not allow allocatable components in EQUIVALENCE}
after:"an allocatable array,"

```
add: "a derived-type object containing an ultimate allocatable
              component,"

Subclause 5.5.2
[58:30] {{WG5: allocatable components are not allowed in COMMON since they are
               excluded from sequence types}}

Subclause 5.5.2.3
[59:42+] {Common Association}
{{no edit needed; not allowed in COMMON}}

Subclause 6.1.2
[63:8] {corresponding restriction to that preventing array ops on pointer
        components}
after:"attribute"
add:"and must not have the ALLOCATABLE attribute"

Subclause 6.3.1
[67:16] {allocate statement}
{ok - because POINTER components were allowed}

Subclause 6.3.1.1
[68:1+] {allocation of allocatable arrays - initial status}
{ok, existing text appears to be sufficient}

[68:7+] {{WG5: Specify initial state of allocatable components after ALLOCATE}}
Add new paragraph at the end of this subclause:
"If an object of derived type is created by an ALLOCATE statement, any
ultimate allocatable components have an allocation status of not currently
allocated."

Subclause 6.3.3.1
[69:8] {deallocation}
add footnote: "Allocatable array components of derived-type objects
                       with the SAVE attribute also retain their allocation
                       status."
{{note: similar issue for 69:29, for pointers }}

[69:9] add list items and renumber
"(2) An allocatable array that is part of the return value of a
             function,
 (3) An allocatable array that is part of a dummy argument,"

Subclause 7.1.6.1
[77:40] {{remove derived-type constructors with allocatable array components
         from initialization exprs}}
after: "(3) A structure constructor where each component is an
                     initialization expression"
add:"and no component has the ALLOCATABLE attribute"

Subclause  7.5.1.5
[91:20] {interpretation of intrinsic assignment}
after:"nonpointer components."

add: "Note that for allocatable components the component in the
              variable being defined must already be allocated, and the shapes
              of the corresponding allocatable components of the variable
              being defined and the expression must be the same."

Subclause 9.4.2
[124:17] {we do not allow derived-types with pointer components in i/o
          statements, neither should we allow ones with allocatable components}
after:"If a derived type ultimately contains a pointer component"
add:"or allocatable component"
```

Annex A
[254:12-13] {Fix glossary copy of the definition of an allocatable array}
change:"A named"
to:"An"
append to end of definition:
"An allocatable array may be a named array or a structure
component"

[261:21-22] {Fix glossary copy of the definition of ultimate component}
after:"a component that is of intrinsic type"
add:", has the ALLOCATABLE attribute,"

[261:23]
after:"does not have the"
add:"ALLOCATABLE attribute or"

add to Rationale Section:

5.1.2.9 ALLOCATABLE attribute

Allocatable arrays provide a degree of flexibility intermediate between that of automatic arrays and pointers. The bounds of an automatic array can be calculated on non-constant values, but only at scope entry. Often, this is not sufficiently powerful; nor does it provide for SAVEd arrays whose size is not given by a constant expression. Pointers do provide the facility to arbitrarily determine the array bounds by execution of an ALLOCATE statement. Because of the pointer assignment statement, pointers are a little less closely controlled than allocatable arrays. More than one pointer may be associated with a given object, a condition known as "aliasing" which can negatively impact optimizing compilers. Pointers also allow memory leaks if not properly deallocated.
Array variables, and components of derived types, can be specified as ALLOCATABLE when it is desired to calculate their size as a function of some value determined during execution.
Explicit initialization of allocatable arrays is not permitted for SAVEd variables, because such initialization may take place prior to execution, but the ALLOCATE action only occurs during execution.
Input/output of derived types containing allocatable components is not permitted. This is because the storage of an allocatable component is typically far removed from that of the other derived type components, so it would be a non-trivial burden to the implementor for unformatted i/o. Also, for derived types containing allocatable components, as for derived types containing pointer components, the i/o action desired by the user is unlikely to be that which can be provided automatically. User-defined i/o of derived types is not part of this standard but we wish to avoid adding a facility which will be little used and likely to conflict with the addition of the proper facility in a later revision of the standard (or as a vendor extension).
Allocatable arrays were not permitted in storage-associated contexts (NAMELIST, COMMON, EQUIVALENCE) in Fortran 90. This restriction still holds, and thus objects of derived type containing allocatable components are also not permitted in such contexts. For this reason, and because the primary purpose of SEQUENCE is to allow derived-type objects to appear in such contexts, allocatable components are not permitted in sequence derived types.