ISO IEC JTC 1 SC 22 WG 23 N0625
19 January 2016

Proposed top Fortran specific guidance:

1. Never use implicit typing. Always declare all variables. Use `implicit none` to enforce this.
2. Use explicit conversion intrinsics for conversions of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.
3. Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used. Similarly, use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
4. Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
5. Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
6. Use default initialization in the declarations of pointer components.
7. Specify `pure` (or `elemental`) for procedures where possible for greater clarity of the programmer's intentions.
8. Code a status variable for all statements that support one, and examine its value prior to continuing execution for faults that cause termination, provide a message to users of the program, perhaps with the help of the error message generated by the statement whose execution generated the error.
9. Avoid the use of common and equivalence. Use modules instead of common to share data. Use allocatable data instead of equivalence.
10. Supply an explicit interface to specify the `external` attribute for all external procedures invoked.

For reference, here are all of the Fortran specific guidance from 24773 (N0461):

- Use kind values based on the needed range for integer types via the `selected_int_kind` intrinsic procedure, and based on the range and precision needed for real and complex types via the `selected_real_kind` intrinsic procedure.
- Use explicit conversion intrinsics for conversions of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.

- Use inquiry intrinsic procedures to learn the limits of a variable's representation and thereby take care to avoid exceeding those limits.
- Use derived types to avoid implicit conversions.
- Use compiler options when available to detect during execution when a significant loss of information occurs.
- Use compiler options when available to detect during execution when an integer value overflows.
- Use the intrinsic procedure `bit_size` to determine the size of the bit model supported by the kind of integer in use.
- Be aware that the Fortran standard uses the term "left-most" to refer to the highest-order bit, and the term "left" to mean towards (as in `shiftl`), or from (as in `maskl`), the highest-order bit.
- Be aware that the Fortran standard uses the term "right-most" to refer to the lowest-order bit, and the term "right" to mean towards (as in `shiftr`), or from (as in `maskr`), the lowest-order bit.
- Avoid bit constants made by adding integer powers of two in favour of those created by the bit intrinsic procedures or encoded by BOZ constants.
- Use bit intrinsic procedures to operate on individual bits and bit fields, especially those that occupy more than one storage unit. Choose shift intrinsic procedures cognizant of the need to affect the sign bit, or not.
- Create objects of derived type to hide use of bit intrinsic procedures within defined operators and to separate those objects subject to arithmetic operations from those objects subject to bit operations.
- Use procedures from a trusted library to perform calculations where floating-point accuracy is needed. Understand the use of the library procedures and test the diagnostic status values returned to ensure the calculation proceeded as expected.
- Avoid creating a logical value from a test for equality or inequality between two floating-point expressions. Use compiler options where available to detect such usage.
- Do not use floating-point variables as loop indices; use integer variables instead. (This relies on a deleted feature.) A floating-point value can be computed from the integer loop variable as needed.
- Use intrinsic inquiry procedures to determine the limits of the representation in use when needed.
- Avoid the use of bit operations to get or to set the parts of a floating point quantity. Use intrinsic procedures to provide the functionality when needed.
- Use the intrinsic module procedures to determine the limits of the processor's conformance to IEEE 754, and to determine the limits of the representation in use, where the IEEE intrinsic modules and the IEEE real kinds are in use.
- Use the intrinsic module procedures to detect and control the available rounding modes and exception flags, where the IEEE intrinsic modules are in use.
- Use enumeration values in Fortran only when interoperating with C procedures that have

enumerations as formal parameters and/or return enumeration values as function results.

- Ensure the interoperability of the C and Fortran definitions of every enum type used.
- Ensure that the correct companion processor has been identified, including any companion processor options that affect enum definitions.
- Do not use variables assigned enumeration values in arithmetic operations, or to receive the results of arithmetic operations if subsequent use will be as an enumerator.
- Use the kind selection intrinsic procedures to select sizes of variables supporting the required operations and values.
- Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used.
- Use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
- When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
- Use derived types and put checks in the applicable defined assignment procedures.
- Use static analysis to identify whether numeric conversion will lose information.
- Use compiler options when available to detect during execution when a significant loss of information occurs.
- Use compiler options when available to detect during execution when an integer value overflows.
- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable
- dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local automatic arrays.
- Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Use allocatable character variables where assignment of strings of widely-varying sizes is expected so the left-hand side character variable is reallocated as needed.
- Use intrinsic assignment rather than explicit loops to assign data to statically-sized character variables so the truncate-or-blank-fill semantic protects against storing outside the assigned variable.
- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking throughout development of a code. Disable bounds checking during

production runs only for program units that are critical for performance.

- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable arrays as procedure dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local automatic arrays.
- Use allocatable arrays where arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Declare the lower bound of each array extent to fit the problem, thus minimizing the use of subscript arithmetic.
- Arrays can be declared in modules which makes their bounds information available wherever the array is available.
- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.
- Use allocatable arrays where arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Avoid C interoperability features in programs that do not interoperate with other languages.
- Avoid use of sequence types.
- Use compiler options where available to enable pointer checking during development of a code throughout. Disable pointer checking during production runs only for program units that are critical for performance.
- Use the associated intrinsic procedure before referencing a target through the pointer if there is any possibility of it being disassociated.
- Associate pointers before referencing them.
- Use default initialization in the declarations of pointer components.
- Use initialization in the declarations of all pointers that have the save attribute.
- Use allocatable objects in preference to pointer objects whenever the facilities of allocatable objects are sufficient.
- Use compiler options where available to detect dangling references.
- Use compiler options where available to enable pointer checking throughout development of a code. Disable pointer checking during production runs only for program units that are critical for performance.
- Do not pointer-assign a pointer to a target if the pointer might have a longer lifetime than the target or the target attribute of the target. Check actual arguments that are argument

associated with dummy arguments that are given the target attribute within the referenced procedure.

- Check for successful deallocation when deallocating a pointer by using the `stat=` specifier.
- Use the intrinsic procedure `selected_int_kind` to select an integer kind value that will be adequate for all anticipated needs.
- Use compiler options where available to detect during execution when an integer value overflows.
- Separate integer variables into those on which bit operations are performed and those on which integer arithmetic is performed.
- Do not use shift intrinsics where integer multiplication or division is intended.
- Declare all variables and use `implicit none` to enforce this.
- Do not attempt to distinguish names by case only.
- Do not use consecutive underscores in a name.
- Use a compiler, or other analysis tool, that provides a warning for this.
- Use the volatile attribute where a variable is assigned a value to communicate with a device or process unknown to the processor.
- Do not use similar names in nested scopes.
- Use a processor that can detect a variable that is declared but not used and enable the processor's option to do so at all times.
- Use processor options where available or a static analysis to detect variables to which a value is assigned but are not referenced.
- Do not reuse a name within a nested scope.
- Clearly comment the distinction between similarly-named variables, wherever they occur in nested scopes.
- Never use implicit typing. Always declare all variables. Use `implicit none` to enforce this.
- Use a global `private` statement in all modules to require explicit specification of the `public` attribute.
- Use an `only` clause on every use statement.
- Use renaming when needed to avoid name collisions.
- Favor explicit initialization for objects of intrinsic type and default initialization for objects of derived type. When providing default initialization, provide default values for all components.
- Use type value constructors to provide values for all components.
- Use compiler options, where available, to find instances of use of uninitialized variables.
- Use other tools, for example, a debugger or flow analyzer, to detect instances of the use of uninitialized variables.
- Use parentheses and partial-result variables within expressions to avoid any reliance on a precedence that is not well known.
- Replace any function with a side effect by a subroutine so that its place in the sequence of computation is certain.
- Assign function values to temporary variables and use the temporary variables in the original

expression.

- Declare a function as `pure` whenever possible.
- Use an automatic tool to simplify expressions.
- Check for assignment versus pointer assignment carefully when assigning to names having the pointer attribute.
- Use dummy argument intents to assist the processor's ability to detect such occurrences.
- Use a compiler, or other tool, that can detect dead or deactivated code.
- Use a coverage tool to check that the test suite causes every statement to be executed.
- Use an editor or other tool that can transform a block of code to comments to do so with dead or deactivated code.
- Use a version control tool to maintain older versions of code when needed to preserve development history.
- Cover cases that are expected never to occur with a case default clause to ensure that unexpected cases are detected and processed, perhaps emitting an error message.
- Avoid the use of computed `go to` statements.
- Use the block form of the do-loop, together with cycle and exit statements, rather than the non-block do-loop.
- Use the `if` construct or `select case` construct whenever possible, rather than statements that rely on labels, that is, the arithmetic `if` and `go to` statements.
- Use names on block constructs to provide matching of initial statement and end statement for each construct.
- Ensure that the value of the iteration variable is not changed other than by the loop control mechanism during the execution of a `do` loop.
- Verify that where the iteration variable is an actual argument, it is associated with an `intent(in)` or a value dummy argument.
- Declare array bounds to fit the natural bounds of the problem.
- Declare interoperable arrays with the lower bound 0 so that the subscript values correspond between languages, where doing so reduces the overall amount of explicit subscript arithmetic.
- Use a tool to automatically refactor unstructured code.
- Replace unstructured code manually with modern structured alternatives only where automatic tools are unable to do so.
- Use the compiler or other tool to detect archaic usage.
- Specify explicit interfaces by placing procedures in modules where the procedure is to be used in more than one scope, or by using internal procedures where the procedure is to be used in one scope only.
- Specify argument intents to allow further checking of argument usage.
- Specify `pure` (or `elemental`) for procedures where possible for greater clarity of the programmer's intentions.
- Use a compiler or other tool to automatically create explicit interfaces for external procedures.
- Do not pointer-assign a pointer to a target if the pointer association might have a longer lifetime

than the target or the `target` attribute of the target.

- Use allocatable variables in preference to pointers wherever they provide sufficient functionality.
- Use explicit interfaces, preferably by placing procedures inside a module or another procedure.
- Use a processor that checks all interfaces, especially if this can be checked during compilation with no execution overhead.
- Use a processor or other tool to create explicit interface bodies for external procedures.
- Prefer iteration to recursion, unless it can be proved that the depth of recursion can never be large.
- Code a status variable for all statements that support one, and examine its value prior to continuing execution for faults that cause termination, provide a message to users of the program, perhaps with the help of the error message generated by the statement whose execution generated the error.
- Appropriately treat all status values that might be returned by an intrinsic procedure or by a library procedure.
- Decide upon a strategy for handling errors, and consistently use it across all portions of the program.
- Use `stop` or `error stop` as appropriate.
- Do not use common to share data. Use modules instead.
- Do not use equivalence to save storage space. Use allocatable data instead.
- Avoid use of the transfer intrinsic unless its use is unavoidable, and then document the use carefully.
- Use compiler options where available to detect violation of the rules for common and equivalence.
- Use allocatable data items rather than pointer data items whenever possible.
- Use final routines to free memory resources allocated to a data item of derived type.
- Use a tool during testing to detect memory leaks.
- Declare a type-bound procedure to be `non overridable` when necessary to ensure that it is not overridden.
- Provide a private component to store the version control identifier of the derived type, together with an accessor routine.
- Specify that an intrinsic or external procedure has the `intrinsic` or `external` attribute, respectively, in the scope where the reference occurs.
- Use compiler options to detect use of non-standard intrinsic procedures.
- Use libraries from reputable sources with reliable documentation and understand the documentation to appreciate the range of acceptable input.
- Verify arguments to library procedures when their validity is in doubt.
- Use condition constructs such as `if` and `where` to prevent invocation of a library procedure with invalid arguments.
- Provide explicit interfaces for library procedures. If the library provides a module containing

interface bodies, use the module.

- Correctly identify the companion processor, including any options affecting its types.
- Use the `iso_c_binding` module, and use the correct constants therein to specify the type kind values needed.
- Use the `value` attribute as needed for dummy arguments.
- Use compiler options to effect a static link.
- Use explicit interfaces for the library code if they are available. Avoid libraries that do not provide explicit interfaces.
- Carefully construct explicit interfaces for the library procedures where library modules are not provided.
- Prefer libraries that provide procedures as module procedures rather than as external procedures.
- Check any return flags present and, if an error is indicated, take appropriate actions when calling a library procedure.
- Avoid use of the C pre-processor `cpp`.
- Avoid pre-processors generally. Where deemed necessary, a Fortran mode should be set.
- Use processor-specific modules in place of pre-processing wherever possible.
- Use all run-time checks that are available during development.
- Use all run-time checks that are available during production running, except where performance is critical.
- Use several processors during development to check as many conditions as possible.
- Provide an explicit interface for each external procedure or replace the procedure by an internal or module procedure.
- Avoid the use of the intrinsic function transfer.
- Avoid the use of common and equivalence.
- Use the compiler or other automatic tool for checking the types of the arguments in calls between Fortran and C, make use of them during development and in production running except where performance would be severely affected.
- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.
- Avoid the use of common and equivalence.
- Specify the save attribute when supplying an initial value.
- Use implicit none to require explicit declarations.
- Use processor options to detect and report use of non-standard features.
- Obtain diagnostics from more than one source, for example, use code checking tools.
- Supply an explicit interface to specify the `external` attribute for all external procedures invoked.
- Avoid use of non-standard intrinsic procedures.
- Specific the `intrinsic` attribute for all non-standard intrinsic procedures.
- Do not rely on the behaviour of processor dependencies. See Annex A.2 of ISO/IEC 1539-1

(2010) for a complete list.

- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.