Proposed top Ada specific guidance:

1. Do not use features explicitly identified as unsafe, such as Unchecked_Deallocation or Unchecked_Conversion.
2. Handle all Exceptions raised by type and subtype-conversions.
3. Protect all data shared between tasks within a protected object or mark the data Atomic.
4. Use pragma Atomic and **pragma** Atomic_Components to ensure that all updates to objects and components happen atomically.
5. Use pragma Volatile and **pragma** Volatile_Components to notify the compiler that objects and components must be read immediately before use as other devices or systems may be updating them between accesses of the program.
6. Rather than using predefined types, such as Float and Long_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, digits 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
7. Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as 'Exponent).
8. For **case** statements and aggregates, do not use the **others** choice.
9. Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.

For reference, here is all of the Ada specific guidance from 24773:

- The predefined 'Valid attribute for a given subtype may be applied to any value to ascertain if the value is a valid value of the subtype. This is especially useful when interfacing with type-less systems or after Unchecked_Conversion.
- A conceivable measure to prevent incorrect unit conversions is to restrict explicit conversions to the bodies of user-provided conversion functions that are then used as the only means to effect the transition between unit systems. These bodies are to be critically reviewed for proper conversion factors.
- Exceptions raised by type and subtype-conversions shall be handled.
- The use of record and array types with the appropriate representation specifications added so that the objects are accessed by their logical structure rather than their physical representation. These representation specifications may address: order, position, and size of data components and fields.
- The use of pragma Atomic and **pragma** Atomic_Components to ensure that all updates to objects and components happen atomically.
- The use of pragma Volatile and **pragma** Volatile_Components to notify the compiler that objects and components must be read immediately before use as other devices or systems may be updating them between accesses of the program.
- The default object layout chosen by the compiler may be queried by the programmer to determine the expected behaviour of the final representation.
- Rather than using predefined types, such as Float and Long_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision

(for example, digits 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.

- Avoid comparing floating-point values for equality. Instead, use comparisons that account for the approximate results of computations. Consult a numeric analyst when appropriate.
- Make use of static arithmetic expressions and static constant declarations when possible, since static expressions in Ada are computed at compile time with exact precision.
- Use Ada's standardized numeric libraries (for example, Generic_Elementary_Functions) for common mathematical operations (trigonometric operations, logarithms, and others).
- Use an Ada implementation that supports Annex G (Numerics) of the Ada standard, and employ the "strict mode" of that Annex in cases where additional accuracy requirements must be met by floating-point arithmetic and the operations of predefined numerics packages, as defined and guaranteed by the Annex.
- Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as 'Exponent).
- In cases where absolute precision is needed, consider replacement of floating-point types and operations with fixed-point types and operations.
- For **case** statements and aggregates, do not use the **others** choice.
- For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.
- Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets.
- Use range checks on conversions involving scalar types and subtypes to prevent generation of invalid data.
- Use static analysis tools during program development to verify that conversions cannot violate the range of their target.
- Do not suppress the checks provided by the language.
- Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.
- Write explicit bounds tests to prevent exceptions for indexing out of bounds.
- This vulnerability can be avoided in Ada by not using the features explicitly identified as unsafe.
- Use ‘Access which is always type safe.
- Use local access types where possible.
- Do not use Unchecked_Deallocation.
- Use Controlled types and reference counting.
- Avoid the use of similar names to denote different objects of the same type.
- Adopt a project convention for dealing with similar names
- See the Ada Quality and Style Guide.
- Use Ada compilers that detect and generate compiler warnings for unused variables or use static analysis tools to detect such problems.
- Do not declare variables of the same type with similar names. Use distinctive identifiers and the strong typing of Ada (for example through declaring specific types such as Pig_Counter **is range** 0 .. 1000; rather than just Pig: Integer;) to reduce the number of variables of the same type.
- Use Ada compilers that detect and generate compiler warnings for unused variables.
- Use static analysis tools to detect dead stores.
- Use *expanded names* whenever confusion may arise.

- Use Ada compilers that generate compile time warnings for declarations in inner scopes that hide declarations in outer scopes.
- Use static analysis tools that detect the same problem.
- If the compiler has a mode that detects use before initialization, then this mode should be enabled and any such warnings should be treated as errors.
- Where appropriate, explicit initializations or default initializations can be specified.
- The pragma Normalize_Scalars can be used to cause out-of-range default initializations for scalar variables.
- The 'Valid attribute can be used to identify out-of-range values caused by the use of uninitialized variables, without incurring the raising of an exception.
- Make use of one or more programming guidelines which prohibit functions that modify global state, and can be enforced by static analysis.
- Keep expressions simple. Complicated code is prone to error and difficult to maintain.
- Always use brackets to indicate order of evaluation of operators of the same precedence level.
- Compilers and other static analysis tools can detect some cases (such as the preceding example).
- Developers may also choose to use short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), but this makes it more difficult for the author to express the distinction between the cases where short-circuited evaluation is known to be needed (either for correctness or for performance) and those where it is not.
- Compilers and other static analysis tools can detect some cases (such as the preceding example).
- Developers may also choose to use short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), but this makes it more difficult for the author to express the distinction between the cases where short-circuited evaluation is known to be needed (either for correctness or for performance) and those where it is not.
- Implementation specific mechanisms may be provided to support the elimination of dead code. In some cases, **pragma**s such as Restrictions, Suppress, or Discard_Names may be used to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example, given the following:

  ```
  package Pkg is
    type Enum is (Aaa, Bbb, Ccc);
    pragma Discard_Names( Enum );
  end Pkg;
  ```

  If Pkg.Enum'Image and related attributes (for example, Value, Wide_Image) of the type are never used, and if the implementation normally builds a table, then the **pragma** allows the elimination of the table.

- For **case** statements and aggregates, avoid the use of the **others** choice.
- For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.
- Whenever possible, a **for loop** should be used instead of a **while loop**.
- Whenever possible, the 'First, 'Last, and 'Range attributes should be used for loop termination. If the 'Length attribute must be used, then extra care should be taken to ensure that the length expression considers the starting index value for the array.

- Avoid the use of **goto**, **loop exit** statements, **return** statements in **procedure**s and more than one **return** statement in a **function** If not following this guidance caused the function code to be clearer – short of appropriate restructuring – then multiple exit points should be used.
- Only use 'Address attribute on static objects (for example, a register address).
- Do not use 'Address to provide indirect untyped access to an object.
- Do not use conversion between Address and access types.
- Use access types in all circumstances when indirect access is needed.
- Do not suppress accessibility checks.
- Avoid use of the attribute Unchecked_Access.
- Use 'Access attribute in preference to 'Address.
- Do not use default expressions for formal parameters.
- Interfaces between Ada program units and program units in other languages can be managed using **pragma** Import to specify subprograms that are defined externally and **pragma** Export to specify subprograms that are used externally. These **pragma**s specify the imported and exported aspects of the subprograms, this includes the calling convention. Like subprogram calls, all parameters need to be specified when using **pragma** Import and **pragma** Export.
- The **pragma** Convention may be used to identify when an Ada entity should use the calling conventions of a different programming language facilitating the correct usage of the execution stack when interfacing with other programming languages.
- In addition, the Valid attribute may be used to check if an object that is part of an interface with another language has a valid value and type.
- If recursion is used, then a Storage_Error exception handler may be used to handle insufficient storage due to recurring execution.
- Alternatively, the asynchronous control construct may be used to time the execution of a recurring call and to terminate the call if the time limit is exceeded.
- In Ada, the **pragma** Restrictions may be invoked with the parameter No_Recursion. In this case, the compiler will ensure that as part of the execution of a subprogram the same subprogram is not invoked.
- In addition to the mitigations defined in the main text, values delivered to an Ada program from an external device may be checked for validity prior to being used. This is achieved by testing the Valid attribute.
- Include exception handlers for every task, so that their unexpected termination can be handled and possibly communicated to the execution environment.
- Use objects of controlled types to ensure that resources are properly released if a task terminates unexpectedly.
- The **abort** statement should be used sparingly, if at all.
- For high-integrity systems, exception handling is usually forbidden. However, a top-level exception handler can be used to restore the overall system to a coherent state.
- Define interrupt handlers to handle signals that come from the hardware or the operating system. This mechanism can also be used to add robustness to a concurrent program.
- Annex C of the Ada Reference Manual (Systems Programming) defines the package Ada.Task_Termination to be used to monitor task termination and its causes.
- Annex H of the Ada Reference Manual (High Integrity Systems) describes several **pragma**, restrictions, and other language features to be used when writing systems for high-reliability applications. For example, the **pragma** Detect_Blocking forces an implementation to detect a potentially blocking operation within a protected operation, and to raise an exception in that case.

- The fact that Unchecked_Conversion is a generic function that must be instantiated explicitly (and given a meaningful name) hinders its undisciplined use, and places a loud marker in the code wherever it is used. Well-written Ada code will have a small set of instantiations of Unchecked_Conversion.
- Most implementations require the source and target types to have the same size in bits, to prevent accidental truncation or sign extension.
- Unchecked_Union should only be used in multi-language programs that need to communicate data between Ada and C or C++. Otherwise the use of discriminated types prevents "punning" between values of two distinct types that happen to share storage.
- Using address clauses to obtain overlays should be avoided. If the types of the objects are the same, then a renaming declaration is preferable. Otherwise, the **pragma** Import should be used to inhibit the initialization of one of the entities so that it does not interfere with the initialization of the other one.
- Use storage pools where possible.
- Use controlled types and reference counting to implement explicit storage management systems that cannot have storage leaks.
- Use a completely static model where all storage is allocated from global memory and explicitly managed under program control.
- Use the overriding indicators on potentially inherited subprograms to ensure that the intended contract is obeyed, thus preventing the accidental redefinition or failure to redefine an operation of the parent.
- Exploit the type and subtype system of Ada to express preconditions (and postconditions) on the values of parameters.
- Document all other preconditions and ensure by guidelines that either callers or callees are responsible for checking the preconditions (and postconditions). Wrapper subprograms for that purpose are particularly advisable.
- Library providers should specify the response to invalid values.
- Use the inter-language methods and syntax specified by the Ada Reference Manual when the routines to be called are written in languages that the ARM specifies an interface with.
- Use interfaces to the C programming language where the other language system(s) are not covered by the ARM, but the other language systems have interfacing to C.
- Make explicit checks on all return values from foreign system code artifacts, for example by using the 'Valid attribute or by performing explicit tests to ensure that values returned by inter-language calls conform to the expected representation and semantics of the Ada application.
- Ensure that the interfaces with libraries written in other languages are compatible in the naming and generation of exceptions.
- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler **when others** =>.
- Document any exceptions that may be raised by any Ada units being used as library routines.
- Do not suppress language defined checks.
- If language-defined checks must be suppressed, use static analysis to prove that the code is correct for all combinations of inputs.
- If language-defined checks must be suppressed, use explicit checks at appropriate places in the code to ensure that errors are detected before any processing that relies on the correct values.
- The **pragma** Restrictions can be used to prevent the use of certain features of the language. Thus, if a program should not use feature X, then writing **pragma** Restrictions (No_X); ensures that any attempt to use feature X prevents the program from compiling.

Similarly, features in a Specialized Needs Annex should not be used unless the application area concerned is well-understood by the programmer.

- For situations where order of evaluation or number of evaluations is unspecified, using only operations with no side-effects, or idempotent behaviour, will avoid the vulnerability;
- For situations involving generic formal subprograms, care should be taken that the actual subprogram satisfies all of the stated expectations;
- For situations involving unspecified values, care should be taken not to depend on equality between potentially distinct values;
- For situations involving bounded errors, care should be taken to avoid the situation completely, by ensuring in other ways that all requirements for correct operation are satisfied before invoking an operation that might result in a bounded error. See the Ada Annex section on Initialization of Variables [LAV] for a discussion of uninitialized variables in Ada, a common cause of a bounded error.
- All data shared between tasks should be within a protected object or marked Atomic, whenever practical;
- Any use of Unchecked_Deallocation should be carefully checked to be sure that there are no remaining references to the object;
- **pragma** Suppress should be used sparingly, and only after the code has undergone extensive verification.
- The other errors that can lead to erroneous execution are less common, but clearly in any given Ada application, care must be taken when using features such as:
  - abort;
  - Unchecked_Conversion;
  - Address_To_Access_Conversions;
  - The results of imported subprograms;
  - Discriminant-changing assignments to global variables.
- Many implementation-defined limits have associated constants declared in language-defined packages, generally **package** System. In particular, the maximum range of integers is given by System.Min_Int .. System.Max_Int, and other limits are indicated by constants such as System.Max_Binary_Modulus, System.Memory_Size, System.Max_Mantissa, and similar. Other implementation-defined limits are implicit in normal ‘First and ‘Last attributes of language-defined (sub) types, such as System.Priority’First and System.Priority’Last. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.
  - Programmers should be aware of the contents of Annex M of the Ada Standard and avoid implementation-defined behaviour whenever possible.
  - Programmers should make use of the constants and subtype attributes provided in package System and elsewhere to avoid exceeding implementation-defined limits.
  - Programmers should minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, they should declare their own numeric types to match their particular application needs.

When there are implementation-defined formats for strings, such as Exception_ Information, any necessary processing should be localized in packages with implementation-specific variants.

- Use **pragma** Restrictions (No_Obsolescent_Features) to prevent the use of any obsolescent features.

- Refer to Annex J of the Ada reference manual to determine if a feature is obsolescent.