GNAT Pro User's Guide

Supplement for High-Integrity Edition Platforms

The GNAT Ada Compiler
GNAT GPL Edition, Version 2012
Document revision level 246224
Date: 2012/02/29

AdaCore

About This Guide

This guide describes the GNAT Pro High-Integrity Edition, an Ada tool suite designed especially for applications that need to be certified for compliance with safety standards such as DO-178B.

GNAT Pro implements Ada 95 and Ada 2005, and it may also be invoked in Ada 83 compatibility mode. By default, GNAT Pro assumes Ada 95, but you can override with a compiler switch to explicitly specify the language version. (Please refer to the section "Compiling Different Versions of Ada", in *GNAT Pro User's Guide*, for details on these switches.) Throughout this manual, references to "Ada" without a year suffix apply to both the Ada 95 and Ada 2005 versions of the language.

A major feature of the GNAT Pro High-Integrity Edition is the support for both predefined and user-specified *profiles*. A profile is a compiler-enforced Ada language subset with a corresponding (possibly empty) run-time library. Thus selecting a profile has two effects:

- The compiler will reject any source file that uses features outside the chosen subset.
- The run-time library (if any) bound with the program will contain support only for the features in the chosen subset

The profiles predefined by the GNAT Pro High-Integrity Edition, depending on the target, are as follows:

- The *Zero Footprint Profile*, an Ada subset requiring no run-time support;
- The *Cert Profile*, comprising the features in the Zero Footprint profile together with a restricted set of thread-safe features, in particular exception propagation;
- The *Ravenscar Profiles*, comprising the features in the Zero Footprint profile (Ravenscar SFP) or the Cert profile (Ravenscar Cert) together with a restricted set of tasking features;
- The *Full-Runtime Profile*, comprising the complete Ada language.

The Zero Footprint Profile, the Cert Profile and the Ravenscar Profiles are collectively known as the *High-Integrity Profiles*, since they are designed to be used in applications that need to be certified for safety-critical use. A *High-Integrity Profile program* is a program built with one of the High-Integrity Profiles.

These profiles are applications of a general technology implemented by the GNAT Pro High-Integrity Edition: user-defined profiles. The programmer can select a particular subset of language features and obtain compiler enforcement of this subset, with a corresponding specially configured run-time library containing only what is required for the chosen features. This provides additional flexibility and control, beyond the predefined profiles.

This guide explains the various GNAT Pro features and tool options that may be useful for high-integrity applications, defines the High-Integrity Profiles and the run-time configurability mechanism. Target specific topics are treated in the *GNAT Pro User's Guide, Supplement for Cross Platforms*.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [The High Integrity Philosophy], page 5, gives an overview of the product.
- Chapter 2 [Using GNAT Pro Features Relevant to High-Integrity], page 7, describes how to take advantage of some features of this product in an High-Integrity context.
- Chapter 3 [The Predefined Profiles], page 25, presents the different profiles that you may encounter using the GNAT Pro High-Integrity Edition, depending on the system you are targeting.
- Chapter 4 [The GNAT Configurable Run Time Facility], page 37, explains how to configure the GNAT run-time library / define a specific Ada feature profile based on specific application requirements.

What You Should Know Before Reading This Guide

This guide assumes a basic understanding of the Ada 95 language and familiarity with the *GNAT Pro User's Guide*, in particular the material related to cross-compilation environments. It does not require knowledge of the new features introduced by Ada 2005, (officially known as ISO/IEC 8652:1995 with Technical Corrigendum 1 and Amendment 1). Both Ada reference manuals are included in the GNAT Pro documentation package.

Related Information

For further information about GNAT and Ada, please refer to the following documents:

- *GNAT Pro User's Guide*, which provides information on how to use the GNAT Pro toolset on native platforms.
- *GNAT Pro User's Guide Supplement for Cross Platforms*, which provides information on how to use GNAT Pro on cross-compilation platforms.
- *GNAT Pro Reference Manual*, which contains reference material for the GNAT Pro implementation of Ada.
- *Ada 95 Reference Manual*, which defines the Ada 95 language standard.
- *Ada 2005 Reference Manual*, which defines the Ada 2005 language standard.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `Subprogram names` and `package names`.
- '`Option flags`'.
- '`File Names`'.
- *`Variable names`* and *`field names`*.
- *Emphasis*.
- [optional information or parameters]
- Examples are described by text

```
and then shown this way.
```

Commands that are entered by the user are preceded in this manual by the characters "`$ `" (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the "`$ `" replaced by the relevant prompt string.

1 The High Integrity Philosophy

The High-Integrity Edition of GNAT Pro is intended to reduce costs and risks in meeting safety certification standards such as DO-178B for applications written in Ada.

The main way this goal is met is through the use of language profiles: either one of the High-Integrity profiles supplied with the product, or an appropriately constrained Ada subset selected by the programmer. This will restrict Ada construct usage so that either no run-time library, or else a simple certifiable library, is used by a conforming application. In the first case (the Zero Footprint Profile) there is no need to certify any object code outside of the application code. Note that there still are some run-time source files, which provide definitions used by the compiler, but these files do not generate any object code. Depending on the certification protocol, it may still be necessary to include tests for correct access to these definitions in these run-time files.

In the second case (simple certifiable library, either for a predefined High-Integrity profile or for a user-defined configuration), the library is designed to expedite the certification process for an application that needs to use its features (e.g. concurrency, exception propagation...) For example, it may be simpler and less expensive to certify a concurrent program built on the Ravenscar Profile than to certify a sequential program with no run-time library (should concurrency need to be simulated in application code).

Although limited in terms of dynamic Ada semantics, all High-Integrity profiles fully support static Ada constructs such as generic templates and child units. These profiles also allow the use of tagged types (at library level) and other Object-Oriented Programming features, but you can prohibit the general use of dynamic dispatching at the application level through `pragma Restrictions`.

In addition to the High-Integrity Profiles, the GNAT Pro High-Integrity Edition may also support a Full-Runtime Profile thus allowing usage of the complete Ada language. However, the availability of this profile is target dependent, and certification materials for the full run-time profile are not available.

A traditional problem with predefined profiles is their inflexibility: if you need to use a feature that is outside a given profile, then it is your responsibility to address the certification issues deriving from its use. The GNAT Pro High-Integrity Edition accommodates this need by allowing you to define a profile for the specific set of features you will use. Typically this will be for features with run-time libraries that require associated certification materials. Thus your program will have a tailored run-time library supporting only those features that you have specified. Indeed, the High-Integrity profiles themselves are implemented in this manner, using the product's run-time configurability mechanism.

2 Using GNAT Pro Features Relevant to High-Integrity

The GNAT Pro High-Integrity Edition contains a number of features especially useful for safety-critical programming:

- With the `-gnatD` or `-gnatG` options, the compiler generates a low-level version of the source program in an Ada-like format.

  This serves as an intermediate form between the original source program and the generated object code, thus supporting traceability requirements. This intermediate representation may be used as a reference point for verifying that the object code matches the source code. This expanded low-level generated code can also be used as the reference point for run-time debugging.
- With the `-gnatR` option, the compiler generates information about the choice of data representations.
- The compiler produces an extensive set of warning messages to diagnose situations that are likely to be errors, even though they do not correspond to illegalities in Ada Reference Manual terms.
- Two restriction identifiers for `pragma Restrictions` prohibit constructs that would generate implicit loops (`No_Implicit_Loops`) or implicit conditionals (`No_Implicit_Conditionals`). If these restrictions are specified, then either alternative code is generated without the implicit code, or the construct in question is rejected, forcing the programmer to make the loop or conditional explicit in the source. This is desirable in ensuring full testing of conditionals. See Section 2.6 [Controlling Implicit Conditionals and Loops], page 14.
- A restriction identifier for `pragma Restrictions` can be used to restrict the form of explicit conditionals. Using the restriction identifier `No_Direct_Boolean_Operators` prohibits the use of `and` and `or` operators, forcing instead the use of `and then` and `or else`. This can be useful in some certification procedures in order to reduce the number of test cases needed to fully exercise compound conditions.
- With the `-gnaty`*x* options, the compiler performs various style checks that can be used to enforce rigorous coding standards, easing the verification process by ensuring that the source is formatted in a uniform manner.
- Annex H of the *Ada Reference Manual* is fully implemented, and all implementation-dependent characteristics of the GNAT Pro implementation are defined in the *GNAT Pro Reference Manual*, further supporting the goal of reviewable object code.

2.1 Exceptions and the High-Integrity Profiles

The predefined profiles implement two different levels of support for exception handling. The ZFP and Ravenscar SFP (Small Footprint) profiles implement the scenario where `pragma Restrictions (No_Exception_Propagation)` is implicitly applied to an application (see below). The Cert and Ravenscar Cert profiles implement full Ada 83 exception handling, plus limited use of Ada 95 / Ada 2005 exception occurrences. Both implementations provide a last chance handler capability to deal with unhandled exceptions. Details are described in the sections on exceptions in the chapters specific to the individual profiles.

The restriction `No_Exception_Propagation`, which is the default mode for the ZFP and Ravenscar SFP profiles, allows exceptions to be raised and handled only if the handler is in the same subprogram (more generally in the same scope not counting packages and blocks). This limits the handling of exceptions to cases where raising the exception corresponds to a simple goto to the exception handler. This is especially useful for predefined exceptions. For example, the following is allowed in the ZFP or Ravenscar SFP profiles:

```
begin
   X := Y + Z;
exception
   when Constraint_Error =>
      ... result of addition outside range of X
end;
```

With this restriction in place, handlers are allowed, but can only be entered if the raise is local to the scope with the handler. The handler may not have a choice parameter, use of GNAT.Current_Exception is not permitted, and use of reraise statements (raise with no operand) is not permitted.

Warnings are given if an implicit or explicit exception raise is not covered by a local handler, or if an exception handler does not cover a case of a local raise. The following example shows these warnings in action:

```
1. pragma Restrictions (No_Exception_Propagation);
2. procedure p (C : in out Natural) is
3. begin
4.    begin
5.       C := C - 1;
6.    exception
7.       when Constraint_Error =>
8.          null;
9.       when Tasking_Error =>
         |
   >>> warning: pragma Restrictions
       (No_Exception_Propagation) in effect, this
       handler can never be entered, and has been
       removed
```

```
10.           null;
11.     end;
12.
13.     begin
14.        C := C - 1;
                  |
     >>> warning: pragma Restrictions
         (No_Exception_Propagation) in effect,
         "Constraint_Error" may call
         Last_Chance_Handler

15.     end;
16.
17.     begin
18.        raise Program_Error;
           |
     >>> warning: pragma Restrictions
         (No_Exception_Propagation) in effect,
         Last_Chance_Handler will be called
         on exception

19.     end;
20. end p;
```

These warnings may be turned off globally using the switch -gnatw.X, or by using pragma Warnings (Off) locally.

As shown by the warnings above, if an exception handler If any other exception is raised, it is treated as unhandled, and causes the last chance handler to be entered.

In a High-Integrity program, you can forbid exception handling entirely, but still allow the raising of exceptions by using:

```
pragma Restrictions (No_Exception_Handlers);
```

No handlers are permitted in a program if this restriction is specified, so exceptions can never be handled in the usual Ada way. If run time checking is enabled, then it is possible for the predefined exceptions `Constraint_Error`, `Program_Error`, or `Storage_Error` to be raised at run time.

When such an exception is raised, a call is made to the routine designated by the symbol `__gnat_last_chance_handler`. This routine has distinct parameters for the ZFP and Ravenscar SFP profiles vs. the Cert and Ravenscar Cert profiles, as detailed in the profile-specific sections.

If your program may raise `Constraint_Error`, `Program_Error`, or `Storage_Error`, then the application must include an appropriate “last chance handler” to deal with the fatal errors represented by these exception occurrences. This handler can be written in C or in Ada using the profile-specific parameters. All

last chance handler implementations must terminate or suspend the thread that executes the handler.

Exception declarations and `raise` statements are still permitted under this restriction. A `raise` statement is compiled into a call of `__gnat_last_chance_handler`.

To suppress all run-time error checking and generation of implicit calls to the last chance handler, and to disallow all `raise` statements, you may use:

```
pragma Restrictions (No_Exceptions);
```

The following switch is not relevant if the program has `pragma Restrictions (No_Exception_Handlers)` or `pragma Restrictions (No_Exception_Propagation)`:

'`-E`' This switch causes traceback information to be stored with exception occurrences and is only applicable when there are exception handlers.

The following switches are not relevant if the program has `pragma Restrictions (No_Exceptions)` or `pragma Restrictions (No_Exception_Propagation)`:

'`-E`' See above.

'`-gnatE`' This switch enables run-time checks for "Access-Before-Elaboration" errors.

2.2 Allocators and the High-Integrity Profiles

Allocators and unchecked deallocation are permitted in a High-Integrity Profile program. Use of these features will generate calls on one of the following C convention functions:

```
void *__gnat_malloc (size_t size);
void __gnat_free (void *ptr);
```

The corresponding Ada subprogram declarations are:

```
type Pointer is access Character;
-- This is really a void pointer type, the result from
-- Gnat_Malloc will always be maximally aligned.

function Gnat_Malloc (size : Interfaces.C.size_t)
  return Pointer;
pragma Export (C, Gnat_Malloc, "__gnat_malloc");

procedure Gnat_Free (Ptr : Pointer);
pragma Export (C, Gnat_Free, "__gnat_free");
```

These functions are part of the run-time library for some High Integrity Profiles, and must be provided by the user otherwise. If included in the run-time library,

they appear in the file '`s-memory.adb`'. To know if a given profile provides this feature, see the relevant section in Chapter 3 [The Predefined Profiles], page 25.

If these functions are not provided, then the user must define `__gnat_malloc` either in C or in Ada, using the above declarations; otherwise the program will fail to bind. Analogously, if the program uses `Unchecked_Deallocation`, then `__gnat_free` must be defined.

For example, on VxWorks DO-178B, one approach (if no deallocation is to be allowed) is for the user to implement `Gnat_Malloc` through calls on `memLib` routines. The `memNoMoreAllocations` function can be invoked to prevent further allocations, for example at the end of package elaboration.

To prohibit the use of allocators or unchecked deallocation, you can use `pragma Restrictions` with the following restriction identifiers (these are defined in the *Ada Reference Manual*):

`pragma Restrictions (No_Local_Allocators);`
: This prohibits the use of allocators except at the library level (thus allocations occur only at elaboration time, and not after the invocation of the main subprogram).

`pragma Restrictions (No_Allocators);`
: This prohibits all explicit use of allocators, thus preventing allocators both at the local and library level.

`pragma Restrictions (No_Implicit_Heap_Allocations);`
: This prohibits implicit allocations (for example an array with non-static subscript bounds declared at library level).

`pragma Restrictions (No_Unchecked_Deallocation);`
: This prohibits all use of the generic procedure `Ada.Unchecked_Deallocation`.

If any or all of these pragmas appear in the '`gnat.adc`' file, the corresponding construct(s) will be forbidden throughout the application. If all four of the above restrictions are in place, then no calls to either `__gnat_malloc` or `__gnat_free` will be generated.

2.3 Array and Record Assignments and the High-Integrity Profiles

The use of assignments of arrays and records is permitted in a High-Integrity Profile program. However, on some targets such constructs may generate calls on the C library functions `memcpy`, `memmove` or `bcopy`. There are two ways to deal with this issue.

First, such assignments can be avoided at the source code level. You can replace an array assignment by an explicit loop, and a record assignment by

a series of assignments to individual components. You can encapsulate such statements in a procedure if many such assignments occur.

Second, you can reuse or define an appropriate `memcpy` (and/or `memove` and/or `bcopy`) routine.

For example, if certification protocols permit, you can use the `memcpy`, `memmove` or `bcopy` routine(s) from the C library. Otherwise, the following Ada procedure will supply the needed `memcpy` functionality:

```
with System; use System;
with Interfaces.C; use Interfaces.C;
function memcpy (dest, src : Address;
                 n         : size_t) return Address;
pragma Export (C, memcpy, "memcpy");

with Ada.Unchecked_Conversion;
function memcpy (dest, src : Address;
                 n         : size_t) return Address is
   subtype mem is char_array (size_t);
   type memptr is access mem;
   function to_memptr is
      new Ada.Unchecked_Conversion (address, memptr);
   dest_p : constant memptr := to_memptr (dest);
   src_p  : constant memptr := to_memptr (src);

begin
  if n > 0 then  -- need to guard against n=0 since size_t is a modular type
    for J in 0 .. n - 1 loop
       dest_p (J) := src_p (J);
    end loop;
  end if;

  return dest;
end memcpy;
```

The above `memcpy` routine provides the minimal required functionality. A more elaborate version that deals with alignments and moves by words rather than bytes where possible would improve performance. On some targets there may be hardware instructions (e.g. `rep movsb` on the `x86` architecture) that can be used to provide the needed functionality.

Note that `memcpy` is not required to handle the overlap case, and the GNAT Pro compiler ensures that any call to `memcpy` meets the requirement that the operands do not overlap.

On the other hand, `memmove` and `bcopy` are required to handle overlaps. Note that the order of the arguments of `bcopy` differs from `memcpy` and `memmove`. A user-supplied version of `bcopy` should take into account these differences.

2.4 Object-Oriented Programming and the High-Integrity Profiles

The High-Integrity Profiles support a large part of Ada's object-oriented programming facilities.

Objects of tagged and class-wide types may be declared. Dispatching and class-wide subprograms are allowed.

Restrictions are:

- Tagged types must be declared at the library level.
- No controlled types.
- `Ada.Tags.Internal_Tag` and `Ada.Tags.Tag_Error` are not available in the default implementations of these profiles.

Ada 2005 object-operation notation is fully supported under ZFP profile. Ada 2005 generic dispatching constructors are not supported under ZFP profile, and a subset of Ada 2005 interface types is supported under ZFP profile. Restrictions on interfaces are:

- No task interfaces, protected interfaces or synchronized interfaces.
- No dynamic membership test applied to interfaces (only those cases in which the evaluation can be performed at compile-time are supported).
- No class-wide interface conversions.
- No declaration of tagged type covering interfaces in which its parent type has variable-size components.
- 'Address not supported on objects whose visible type is a class-wide interface.

Note: Dispatch tables are used to implement dynamic dispatching: each tagged type has an associated data structure including a table containing the address of each of its primitive operations. The format of dispatch tables is compatible with the format described in the Itanium C++ ABI. It means, in particular, that the dispatching mechanism is deterministic and bounded in time, with a performance similar to an indirect call. Furthermore, dispatch tables are generated by the compiler as static data that is placed in a read-only data section of the object code. Appropriate linker scripts can be used to ensure that such sections are placed in ROM. Placing such tables in ROM is recommended because it offers some level of robustness to the dispatching mechanism: it prevents the possibility of unintended changes in such tables that could affect the control flow of the code. In addition, declarations of tagged types in this profile are statically elaborable and thus can be used in the presence of the restriction No_Elaboration_Code.

In order for the debugger to be able to print the complete type description of a tagged type, the program must have Ada.Tags as part of its closure. Otherwise, the debugger will not be able to print the name of the type from which the tagged

type has been derived. This limitation can easily be addressed by introducing an artificial dependency on the Ada.Tags unit. Another option is to compile one unit that declares a tagged type with -fno-eliminate-unused-debug-types. This does not affect the ability to print the value of tagged objects, which should work without problem regardless.

2.5 Functions Returning Unconstrained Objects

By default, the High-Integrity Profiles allow functions returning unconstrained objects such as unconstrained arrays or discriminated records without default initializations for discriminants. To implement this capability, the compiler generates references to a secondary stack mechanism that requires run-time support. If you need to use this capability, you should refer to the relevant section in Chapter 3 [The Predefined Profiles], page 25, to see if it is already provided in the run-time library or if you are responsible for providing an appropriate implementation of this unit for a given profile. The specification of the secondary stack mechanism is described in the Ada package `System.Secondary_Stack`, which is in the file '`s-secsta.ads`'.

If you wish to disable this capability, you can use the `pragma Restrictions (No_Secondary_Stack)`; that will generate an error at compile time for each call to a function returning an unconstrained object.

2.6 Controlling Implicit Conditionals and Loops

Certain complex constructs in Ada result in generated code that contains implicit conditionals, or implicit `for` loops. ("Implicit" means that the conditionals/loops are not present in the Ada source code.) For example, slice assignments result in both kinds of generated code.

In some certification protocols, conditionals and loops require special treatment. For example, in the case of a conditional, it may be necessary to ensure that the test suite contains cases that branch in both directions for a given conditional. A question arises as to whether implicit conditionals and loops generated by the compiler are subject to the same verification requirements.

To address this issue, the GNAT Pro High-Integrity Edition defines two restriction identifiers that control whether the compiler is permitted to generate implicit conditionals and loops :

```
pragma Restrictions (No_Implicit_Conditionals);
pragma Restrictions (No_Implicit_Loops);
```

These are partition-wide restrictions that ensure that the generated code respectively contains no conditionals and no loops. This is achieved in one of two ways. Either the compiler generates alternative code to avoid the implicit construct (possibly with some loss of efficiency) or, if it cannot find an equivalent code sequence, it rejects the program and flags the offending construct. In the

latter situation, the programmer will need to revise the source program to avoid the implicit conditional or loop.

As an example, consider the slice assignment:

```
Data (J .. K) := Data (R .. S);
```

Ada language semantics requires that a slice assignment of this type be performed nondestructively, as though the right hand side were computed first into a temporary, with the value then assigned to the left hand side. In practice it is more efficient to use a single loop, but the direction of the loop needs to depend on the values of `J` and `R`. If `J` is less than `R` them the move can take place left to right, otherwise it needs to be done right to left. The normal code generated by GNAT Pro reflects this requirement:

```
if J < R then
   for L in J .. K loop
      Data (L) := Data (L - J + R);
   end loop;
else
   for L in reverse J .. K loop
      Data (L) := Data (L - J + R);
   end loop;
end if;
```

This code clearly contains both implicit conditionals and implicit loops. If the restriction `No_Implicit_Conditionals` is active, then the effect is to generate code that uses a temporary:

```
for L in R .. S loop
   Temp (L - R) := Data (L);
end loop;
for L in J .. K loop
   Data (L) := Temp (L - J);
end loop;
```

This code avoids an implicit conditional at the expense of doing twice as many moves. If the restriction `No_Implicit_Loops` is also specified, then the slice assignment above would not be permitted, and would be rejected as illegal. This means that the programmer would need to modify the source program to have an explicit loop (in the appropriate direction). This loop could then be treated in whatever manner is required by the certification protocols in use.

The following constructs are not permitted in the presence of `No_Implicit_Conditionals` (note that some are in any event excluded from the High-Integrity Profiles):

- Comparison of record or array values
- Array concatenation
- Controlled types
- Protected types

- Asynchronous select
- Conditional entry call
- Delay statement
- Selective accept
- Timed entry call
- Tagged types
- Distributed System Annex features (shared passive partitions, etc.)
- `Width` attribute applied to real type
- Absolute value operator if checks are on
- `rem`, `mod`, and division operators if checks are on
- `Is_Negative` intrinsic function
- Dynamic elaboration check

The following constructs are not permitted in the presence of `No_Implicit_Loops` (note that entry families are in any event excluded from some or all High-Integrity Profiles):

- Array aggregate with `others` clause
- Array types with implicit component initialization
- Array equality test
- Array concatenation
- Logical operations on array types
- Array assignments
- Controlled array types
- Entry families
- Default array stream attributes

2.7 Controlling Use of Conditional Operators

Some testing procedures for certification can be more efficient if application programs avoid the explicit use of `and`/`or` and instead use `and then`/`or else`. This can facilitate meeting the requirement for MCDC ("Modified Condition / Decision Coverage") testing. The net effect of using the short-circuit versions is a significant reduction in the number of test cases needed to demonstrate code and condition coverage.

The GNAT Pro High-Integrity Edition provides an additional restriction identifier that addresses this issue by controlling the presence of direct boolean conditional operators:

```
pragma Restrictions (No_Direct_Boolean_Operators);
```

These are partition-wide restrictions that ensure that the source code does not contain any instances of the direct boolean operators `and` or `or`. This will alert the programmer to the requirement to use `and then` or `or else` as appropriate

2.8 Avoiding Elaboration Code

Ada allows constructs (e.g., variables with implicit initializations) for which so-called *elaboration code* must be generated. In a certification context the need to certify elaboration code will increase costs, as it will be necessary to address questions such as why the compiler implicitly generated the elaboration code, which Ada requirement it met, which test cases are needed.

The GNAT Pro High-Integrity Edition provides the pragma `Restrictions (No_Elaboration_Code)`, which alerts you to constructs for which elaboration code would be generated by the compiler. When this pragma is specified for a compilation unit, the compiler outputs an error message whenever it needs to generate elaboration code. You must then revise the program so that no elaboration code is generated. As an example consider the following:

```
package List is
   type Elmt;
   type Temperature is range 0.0 ... 1_000.0;
   type Elmt_Ptr is access all Elmt;
   type Elmt is record
      T    : Temperature;
      Next : Elmt_Ptr;
   end record;
end List;

pragma Restrictions (No_Elaboration_Code);
with List;
procedure Client is
   The_List : List.Elmt;
begin
   null;
end Client;
```

When compiling unit `Client`, the compiler will generate the error message:

```
client.adb:4:04: violation of restriction "No_Elaboration_Code" at line 1
```

In this example GNAT Pro needs to generate elaboration code for object `The_List` because Ada requires access values to be **null** initialized (unless they have explicit initializations). To see the elaboration code that would be generated you can remove the `No_Elaboration_Code` restriction and use the '`-gnatG`' switch to view the low-level version of the Ada code generated by GNAT Pro. In this case we obtain:

```
package list is
```

```
      type list__elmt;
      type list__temperature is new float range 0.0E0 .. (16384000.0*2**(-14));
      type list__elmt_ptr is access all list__elmt;
      type list__elmt is record
         t : list__temperature;
         next : list__elmt_ptr;
      end record;
      freeze list__elmt [
         procedure list__elmtIP (_init : in out list__elmt) is
         begin
            _init.next := null;
            return;
         end list__elmtIP;
      ]
   end list;

   with list; use list;
   procedure client is
      the_list : list.list__elmt;
      list__elmtIP (the_list);
   begin
      null;
      return;
   end client;
```

Elaboration code is generated inside procedure `Client` to **null**-initialize the access value inside `The_List` object, by calling the initialization procedure for the type, namely `elmtIP`.

To avoid generating elaboration code, you can add explicit initialization as follows:

```
pragma Restrictions (No_Elaboration_Code);
with List; use List;
procedure Client is
   The_List : List.Elmt := (0.0, null);
begin
   null;
end Client;
```

Since the initialization is now explicit, it becomes part of the requirements mapping and application design. In a certification context it is preferable to certify code that you write explicitly rather than code that gets generated implicitly.

2.9 Removal of Deactivated Code

Deactivated code in the executable requires specific treatment with standards such ED12/DO-178b (see 6.4.4.3d). This treatment can be simplified by diminishing the footprint of deactivated code in the final executable. This section

summarizes the various features that can be used to minimize the footprint of deactivated code in the final executable program.

- Automatic Removal The compiler automatically removes local nested subprograms that are never used at all levels of optimization. It also removes unused library-level subprograms declared within package bodies at optimization level -O1 or higher.
- Static Boolean conditionals The compiler eliminates automatically all code protected by conditionals, if and case statements, that are statically detected to be false. For instance, with those declarations:

  ```
  type Configs is (config1, config2, config3);
  ...
  C : constant Configs := config2;
  ```

  the following code

  ```
  if C = config1 then
     some_code0;
  end if;
  case C is
     when config1 => some_code1;
     when config2 => some_code2;
     when config3 => some_code3;
  end case;
  ```

  is transformed into:

  ```
  some_code2;
  ```

  The elimination done by the compiler can be traced thanks to a compiler optional warning enabled with `-gnatwt`.

- Pragma Eliminate

  The user can prevent code generation for subprograms that are known to be deactivated by placing pragmas eliminate in the configuration pragmas file:

  ```
  pragma Eliminate (Unit, Unused_Subp, Source_Location => "unit.adb:19")
  ```

  The compiler will issue an error if an attempt is made to call an eliminated subprogram. A separate tool, `gnatelim` can produce automatically the list of pragmas Eliminate that can be used for the construction of a given executable.

- Linker Level Removal

  Some configurations support linker level removal of unused subprograms and data. The source code needs to be compiled with the options `-ffunction-sections` and `-fdata-sections` and the linking must be performed with option `-Wl,--gc-sections`. Traceability of the code removal is provided through a chapter called "Discarded Input sections" in the map file that can be produced by the linker with option `-Wl,-M`.

These methods can be used to reduce the amount of code in the final executable corresponding to deactivated source code. The last three offer means of tracing the removed code and thus can be used for traceability purposes or as documentation of the deactivation mechanism. The last method can also remove code added by the compiler such as the implicit initialization procedures associated with composite types when they are not used. It therefore simplifies the reverse traceability from object to source.

2.10 Traceability from Source Code to Object Code

During the build process, the GNAT Pro toolchain manipulates several program representations, in particular:

- The initial source code;
- The expanded code, which is low level Ada pseudo-code;
- Assembler code;
- Object code.

In order to help the traceability process, GNAT Pro gives access to the intermediate format, through the following flags:

'`-gnatD`' This switch causes a listing of a pseudo-Ada low-level version of the compilation unit to be directed to a file.

'`-gnatG`' This switch produces a listing of a pseudo-Ada low-level version of the compilation unit.

'`-gnatL`' This switch enhances traceability of the expanded code by adding the original source code as comments just before the corresponding expansion. It has to be used in conjunction with `-gnatD` or `-gnatG`.

'`-S`' This switch causes generation of an assembly language version of the compilation unit, instead of an object file.

'`-save-temps`'
This switch causes the compiler to save temporary files (in particular, the '`.s`' file) while still generating '`.o`' and '`ALI`' files.

'`-fverbose-asm`'
This switch causes the compiler to decorate the assembly output with comments containing the names of the entities (e.g. local variables) being manipulated by the current assembly instruction.

'`-mregnames`'
This switch causes the compiler to emit symbolic names for register in the assembly output. This switch is specific to PowerPC and even though it is not directly related to traceability, it is worth mentioning because it greatly improves assembly code readability.

'`-Wa,-adhl`'
: This switch instructs the GNU assembler to produce a text listing of the generated code containing the high-level source and the assembly while excluding debugger directives (for readability reasons). The listing goes to standard output and can be redirected to a file using the syntax `-Wa,-adhl=file.lst` which save it in the file `file.lst`.

'`-gnatR`'
: This switch causes representation information to be generated for declared types and objects.

These options can be combined in various ways. A fairly complete source to object traceability is provided by:

```
$ <target>-gcc -c -gnatDL -Wa,-adhl -fverbose-asm -mregnames prg.adb
```

2.11 Optimization issues

The '`-On`' compiler switch sets the optimization level to *n*, where *n* is an integer between 0 and 3. If *n* is omitted, it is set to 1.

Generally you should use the '`-O`' switch to enable optimization, because this is the level at which you can most easily track the correspondence between source code and object code. Although for safety-critical programs optimizations are often regarded with suspicion, the fundamental reason for this concern is traceability. At the '`-O0`' level, traceability is in fact more difficult due to the large amount of naive code that is generated. The optimizations performed at level '`-O1`' eliminate redundant code, but avoid the kind of code-shuffling transformations that can obscure the correspondence between source and object code.

If you consider coverage, it can be useful to disable optimizations on conditional statements. The option '`-fno-short-circuit-optimize`' specifically disables boolean operator optimizations (those which transform `or else` into `or` or `and then` into `and`), and the options '`-fno-if-conversion`' and '`-fno-if-conversion2`' disable optimization passes that try to remove conditional jumps for if statements and use conditional move instead. These switches help to preserve the original jump structure implied by the source, thus helping source-to-object traceability, and facilitating certification and coverage analysis at the object level.

2.12 Other useful features

The '`-gnaty`' compiler switches direct the compiler to enforce style consistency checks, which can be useful in ensuring a uniform lexical appearance (and easing program readability) across a project.

Annex H in the *Ada Reference Manual* defines a set of restrictions identifiers, many of which are relevant for code that needs to be certified. If you provide a `pragma Restrictions` with any of these identifiers, the compiler will prohibit your program from using the corresponding construct.

2.13 Compilation options for the GNAT Pro High-Integrity Tool Chain

For the most part, you use the GNAT Pro tools in the High-Integrity edition in the same way, and with the same set of switches, as in a full Ada environment. However, certain switches are not relevant for the High-Integrity Profiles. This section lists this set of switches; a complete description of the compiler switches appears in the *GNAT Pro User's Guide* and, in the case of general gcc switches, the *Using GNU GCC* manual.

2.13.1 Compiler Switches

The following switches are not relevant in certain High-Integrity profiles, since they are associated with features that are outside these profiles:

'`-fstack-check`'
: This switch enables stack overflow checking, and is allowed for the Cert and Ravenscar Cert profiles on VxWorks Cert 6, VxWorks 653 and VxWorks MILS according to availability, but not for ZFP or Ravenscar SFP.

'`-gnata`'
: This switch enables pragma `Assert`. This is allowed in the Cert and Ravenscar Cert profiles, but not in the ZFP or Ravenscar SFP profiles.

'`-gnato`'
: This switch enables run-time checks for integer overflow, but is subject to exception propagation and handling restrictions in ZFP and Ravenscar SFP.

'`-gnatP`'
: This switch enables polling in a tasking program.

'`-gnatT`'
: This switch sets the timeslice in a tasking program.

'`-gnatz`'
: This switch is used for distributed Ada programs.

2.13.2 `gnatbind` Switches

The following switches are not relevant in Zero Footprint, Cert or Ravenscar modes:

'`-static`'
: This switch specifies linking with a static GNAT run-time library.

'`-shared`'
: This switch specifies linking with a shared GNAT run-time library.

‘-T’ This switch sets the timestamp for a tasking program.

In addition, the ‘-t’ switch overrides standard Ada unit consistency checks and would generally not be used for high-integrity applications.

The ‘-f’ switch for elaboration order control should not be used; see the discussion of elaboration order issues in *GNAT Pro User’s Guide*.

3 The Predefined Profiles

This chapter describes what is prohibited and what is permitted in the different predefined profiles.

3.1 Choosing a Predefined Profile

You choose a particular profile by setting the GNAT Pro configuration mode; this is done through the '`--RTS=`' switch. You need to pass this switch to the compiler, to `gnatbind`, to `gnatlink`, to `gnatmake`, and in general to any tool that needs to be aware of the run-time configuration. The values for the switch for the high-integrity profiles are `zfp`, `ravenscar-sfp`, `cert`, `ravenscar-cert` and `ravenscar-cert-rtp`, corresponding to the establishment of Zero Footprint, Ravenscar Small Footprint, Cert, Ravenscar Cert Kernel, and Ravenscar Cert RTP profiles, respectively. Depending on the targeted environment, there may be other supported full Ada alternatives. These are not described here - see *The GNAT Pro User's Guide Supplement for Cross Platforms*. The switch settings are not case sensitive.

The set of available profiles depends on the target.

Generic Cross
: Zero Footprint is supported. For LEON and ERC 32 targets, Ravenscar SFP is also supported.

VxWorks DO-178B 6.x
: Zero Footprint, Ravenscar Cert and Full-runtime are supported.

VxWorks 653
: Zero Footprint, Cert, Ravenscar Cert and Full-Runtime are supported.

VxWorks MILS
: Zero Footprint, Cert, Ravenscar Cert and Full-Runtime are supported for the VxWorks Guest OS. Zero Footprint is supported for the High Assurance Environment.

Development Host
: A native version of the Zero Footprint run-time library is provided for all high integrity development hosts to facilitate native platform testing. It should be installed into the same location as the corresponding native development system. It is selected as described above. Note that this is a distinct download available on the GNATtracker page for the high integrity cross-development system.

3.2 The Zero Footprint Profile

With the Zero Footprint Profile the generated object modules usually contain no references to the GNAT run-time library. There are a few exceptions related to floating point attributes or for support of software emulated floating point operations. The absence of references to the run-time allows the construction of a standalone program that has no code other than that corresponding to the original source code, together with the elaboration routine generated by the binder. The elaboration routine generated by the binder also avoids any reference to runtime routines or data.

3.2.1 The '`-nostdlib`' Switch

In a certification context, it may be important to guarantee the enforcement of a strict Zero Footprint Profile where one can ensure that no external library is linked with the final image. This can be achieved by using the builder switch '`-nostdlib`'. The GNAT builders (`gnatmake`, `gprbuild`) will pass this switch to the binder and to the linker, with the effect of removing the run-time libraries such as `libgnat.a` and `libgcc.a` from the link path.

In order to use the '`-nostdlib`' switch, the program must meet a number of restrictions in addition to those listed in Section 3.2.2 [Ada Restrictions in the Zero Footprint Profile], page 26:

- No references to entities from runtime packages, e.g., `Ada.External_Tag`;
- No use of pragma `Assert`;
- No use of the following floating-point attributes: `Adjacent`, `Ceiling`, `Compose`, `Copy_Sign`, `Exponent`, `Floor`, `Fraction`, `Leading_Part`, `Machine`, `Machine_Rounding`, `Model`, `Pred`, `Remainder`, `Rounding`, `Scaling`, `Succ`, `Truncation`, `Unbiased_Rounding` (with non-static arguments), and `Valid` and `Unaligned_Valid` (with any argument);
- No use of a secondary stack (this can be enforced by including pragma `Restrictions(No_Secondary_Stack)`; and
- No use of floating-point or other arithmetic operations that are too complex for inline expansion and that therefore require calls to a support library.

3.2.2 Ada Restrictions in the Zero Footprint Profile

The following features are excluded from the Zero Footprint Profile:

- All constructs defined in Section 9 of the *Ada Reference Manual* (i.e., all tasking features)
- Exception propagation. See Section 2.1 [Exceptions and the High-Integrity Profiles], page 8; note that `raise` statements and local exception handling are permitted.
- Packed arrays with a component size other than 1, 2, 4, 8 or 16 bits

- The exponentiation operator, unless one of the following conditions is met:
  - The exponentiation expression is a static expression as defined in the *Ada Reference Manual*, section 4.9
  - The right operand is a static expression as defined in the *Ada Reference Manual*, section 4.9, and is an integral value in the range `0 .. 4`
- 64-bit integer (the type `Long_Long_Integer`) and fixed-point types
- Boolean operations on packed arrays (individual elements may be accessed)
- Comparison operations on arrays of discrete components, with the exception that equality and inequality operators are supported for such arrays
- The attributes `Image`, `Value`, `Body_Version`, `Version`, `Width`, and `Mantissa`
- Controlled types
- The attributes `Address`, `Access`, `Unchecked_Access`, `Unrestricted_Access` when applied to a non library-level subprogram
- Non library-level tagged types
- Annex E features (Distributed Systems)

Note that explicit `with`s of library units are still permitted but the programmer needs to ensure that the GNAT library modules that come with the compiler that is being used in Zero Footprint mode satisfy the requirements of the target hardware. In particular, in an environment where the code needs to be certified, the user takes responsibility for ensuring that the referenced library unit meets the certification conditions.

3.2.3 Predefined Packages in the Zero Footprint Profile

In general, the predefined Ada environment (Annex A), the interfacing packages (Annex B), and the units in the Specialized Needs Annexes may be implemented with the full Ada language and thus will not be appropriate in applications that need to adhere to the Zero Footprint Profile. The only predefined Ada units that are permitted (i.e., that may be `with`ed) in a Zero Footprint Profile program are as follows:

- Package `System`
- Package `Ada.Tags`
- Generic function `Ada.Unchecked_Conversion`
- Generic procedure `Ada.Unchecked_Deallocation`
- Package `Interfaces`, except that references to the types `Integer_64` and `Unsigned_64` are prohibited
- Package `Interfaces.C`

- Package `System.Storage_Elements`
- Generic package `System.Address_To_Access_Conversions`
- Package `System.Machine_Code`

Additionally, the following GNAT packages are permitted since all their subprograms are either intrinsic or are expanded inline:

- Package `GNAT.Source_Info`
- Package `GNAT.IO`, which provides basic input / output capabilities

A sample implementation of the secondary stack is provided that can be used and tailored (in the file '`s-secsta.adb`'). However, since no certification material is provided for this sample implementation, it is your responsibility to certify this implementation. See Section 2.5 [Functions Returning Unconstrained Objects], page 14, for further details.

'`s-memory.adb`' is not provided. If you want to use allocators and unchecked deallocation, you will have to define __gnat_malloc and __gnat_free. See Section 2.2 [Allocators and the High-Integrity Profiles], page 10, for further details.

3.2.4 Thread Registration Issues

The Zero Footprint profile does not provide run-time facilities that require per-thread data. Therefore, thread registration is not relevant to this profile.

3.2.5 Pragmas Automatically Enabled in the Zero Footprint Profile

The following pragmas are automatically enabled when the Zero Footprint Profile is in effect:

- `pragma Restrictions(No_Exception_Handlers)`
- `pragma Discard_Names`

3.2.6 Exceptions and the Last Chance Handler - ZFP and Ravenscar SFP

In this profile, all applications are treated as though `pragma Restrictions (No_Exception_Propagation)` has been applied. It is possible to raise the predefined Ada exceptions, as well as user-defined exceptions and handle them locally.

When such an exception is raised and not handled locally, a function with the following C-convention prototype is called (note that the function name begins with two underscore characters):

```
void __gnat_last_chance_handler (char *source_location,
                                 int line);
```

The corresponding Ada subprogram declaration is:

```
procedure Last_Chance_Handler
  (Source_Location : System.Address; Line : Integer);
pragma Export (C, Last_Chance_Handler,
               "__gnat_last_chance_handler");
```

The `Source_Location` parameter is a C null-terminated string representing the source location of the `raise` statement, as generated by the compiler, or a zero-length string if `pragma Suppress_Exception_Locations` is used. The `Line` parameter (when nonzero) represents the line number in the source. When `Line` is zero, the line number information is provided in `Source_Location` itself.

If your program may raise `Constraint_Error`, `Program_Error`, or `Storage_Error`, then you must explicitly supply an appropriate last chance handler to deal with the fatal errors represented by these exception occurrences. This handler can be written in C with the above prototype, or in Ada using the corresponding procedure declaration. You must write the body of this handler so that it terminates the program.

3.3 The Cert Profile

The Cert profile is a superset of the Zero Footprint Profile, and thus Section 3.2.3 [Predefined Packages in the Zero Footprint Profile], page 27, is relevant. This section describes the additional capabilities provided by the Cert profile.

The Cert profile is based on an implicitly threaded run-time library. The current implementation of this profile is for use on Wind River Systems VxWorks 653 partition OS and the VxWorks MILS VxWorks Guest OS with APEX processes. Although there are no Ada tasks, there are different threads (VxWorks 653/MILS vThreads) executing the code and accessing data structures. For example, The following data items must be per-thread:

- secondary stack,
- thread-accessible pointer to jump buffer,
- thread-accessible pointer to secondary stack location.

By "thread-accessible", we mean that a thread-specific data item is needed to implement the behavior defined by the subset.

3.3.1 Exceptions and the Last Chance Handler - Cert and Ravenscar Cert Profiles

The full semantics of Ada 83 exceptions is supported; limited support for Ada 95 / Ada 2005 exception occurrences is included. The run-time library supports propagation of exceptions and handlers for multiple threads, provided that the threads have been registered with the Ada run-time library: see Section 3.3.3 [Thread Registration], page 30, for details.

The run-time libraries provided by these profiles (both available in the GNAT Pro for VxWorks 653 High-Integrity Edition, and Ravenscar Cert in VxWorks

Cert 6.x High-Integrity Edition) support limited Ada 95 / Ada 2005 exception occurrences, `Ada.Exceptions.Exception_Name`, and a non-symbolic stack trace capability `Ada.Exceptions.Exception_Traces`.

As with the zero footprint case, the application developer may provide a body for a last chance handler, to deal with unhandled Ada exceptions. The declaration of the last chance handler for these profiles is:

```
procedure Ada.Exceptions.Last_Chance_Handler (Except : Exception_Occurrence);
  pragma Export (C,
                 Last_Chance_Handler,
                 "__gnat_last_chance_handler");
  pragma No_Return (Last_Chance_Handler);
```

`Except` is an Ada 95 / Ada 2005 exception occurrence that can be used to identify the exception and the circumstances under which it occurred. A default body is provided in the '`rts-cert/adainclude`' directory of the restricted run-time library as file '`a-elchha.adb`'. Its object module '`a-elchha.o`' is archived in '`rts-cert/adalib/libgnat.a`' along with a partially linked object module '`libgnat.o`' containing the remainder of the Cert profile run-time library. This example handler generates a non-symbolic traceback that can be used on the development host to obtain a symbolic traceback via the vxaddr2line utility. It also propagates an event to the VxWorks 653 health monitor on VxWorks 653 (but not VxWorks MILS).

Note that the last chance handler can have any name, and can be written in C. However, it must obey the parameter profile and export the symbol `__gnat_last_chance_handler` as its entry point.

If you do not supply a last chance handler, the default version will be linked into your application.

The last chance handler for the Cert profile on VxWorks 653 and VxWorks MILS is expected to suspend the vThread that calls it.

See the *GNAT Pro User's Guide Supplement for Cross Platforms* for the details of providing an application-specific last chance handler.

3.3.2 Secondary Stack

The secondary stack supports functions returning objects of unconstrained types (e.g., unconstrained array types) and of variant record types. The default secondary stack size (for the environment task) can be set using the gnatbind "-D" switch, that specifies the value in Kbytes. For APEX processes, the value defaults to 1/4 the requested primary stack size, and is added to the requested allocation from APEX. The size can also be specified in the Apex_Processes.Create_Process routine.

3.3.3 Thread Registration

The thread registration mechanism allows applications to use system threads with Ada code. Specifically, the mechanism supports the Ada exception semantics outside of an Ada tasking context, so that Ada code that uses exceptions can be called by a non-Ada thread. The mechanism is also necessary to support unconstrained function results in Ada code called by system threads.

The Ada/APEX bindings provided with GNAT Pro High Integrity Edition for VxWorks 653 and VxWorks MILS take care of all thread registration issues when using APEX processes. The bindings are provided as part of the source and object paths, just like the remainder of the Ada run-time library.

Ravenscar Cert does not support the use of foreign threads (eg APEX processes).

3.3.4 Dynamic Allocation

A limited dynamic allocation model is supported by this profile. The package `System.Memory` is provided, as described in Section 2.2 [Allocators and the High-Integrity Profiles], page 10. The package provides a simple version of function `gnat_malloc` that simply calls the underlying `malloc` routine. Deallocation is prohibited on VxWorks 653 in certified partition operating system (POS) application partitions.

Note that in VxWorks 653 partitions that use APEX facilities, dynamic allocation is prohibited once the partition has been set into normal mode.

3.3.5 Exponentiation

The cert subset supports all standard Ada forms of exponentiation.

3.3.6 64-bit Numeric Types

64-bit fixed point and integer types are fully supported.

3.3.7 Utility Routines

The following are provided in addition to the library packages available with the Zero Footprint Profile:

- `GNAT.Debug_Utilities` provides formatting routines for use in I/O.
- `Ada.Exceptions.Exception_Traces` provides a non-symbolic traceback capability that can be used with '`vxaddr2line`' to generate symbolic tracebacks. See *The GNAT User's Guide Supplement for Cross Platforms*.
- `Pragma Assert` is supported.
- `Ada.Numerics`, `Ada.Numerics.Generic_Elementary_Functions`, `Ada.Numerics.Elementary_Functions`, `Ada.Numerics.Long_`

`Elementary_Functions` and `Ada.Numerics.Long_Long_Elementary_Functions` are supported, except for the hyperbolic trig functions.

Although the elementary functions packages are very similar to those defined by the Ada standard, they are not intended to fully implement the standard. Rather, the intent is to provide the functionality of the underlying C math library to Ada developers in a convenient way. In particular, this package does not implement Annex G, Numerics, and thus does not implement strict mode. Precision is dependent on the underlying C math library.

One particular implication of the above is that, in contrast to libraries adhering to the Ada standard:

- `Sqrt(1.0 - Cos(X) * Cos(X))` might raise an exception, because `Cos(X)` might be slightly larger than `1.0`
- `Sin(X)` might be slightly larger than `X`
- `Sqrt(X)` might be negative for small `X`

This is not to say that these situations will occur - just that there is no requirement that explicitly states that they will not occur.

3.4 The Ravenscar Profiles

The Ravenscar Profile, named for the venue of the 1997 International Real-Time Ada Workshop where it was conceived, defines a simple set of tasking features for high-performance and high-integrity real-time Ada programs. The feature subset is designed to be small enough to allow efficient and certifiable run-time support, but large enough to permit programming styles needed in practice for real-time systems.

The Ravenscar Profile is intended for applications comprising a fixed number of tasks (the number can be established at elaboration time) where each task body is a loop. The number of loop iterations may be unbounded, and there is a single "invocation event" at each iteration. The invocation event may be either a timeout from a `delay until` or the execution of a protected entry call. Inter-task communication is either through protected objects or via accesses to shared data marked with pragmas `Atomic` or `Volatile`. Task entry calls are therefore forbidden.

3.4.1 Ada Restrictions in the Ravenscar Profiles

The tasking restrictions defined by the Ravenscar Profile are itemized in the following list; details on the rationale and implications may be found in *The Ravenscar Profile for High-Integrity Real-Time Programs* by A. Burns, B. Dobbing and G. Romanski, in *Reliable Software Technologies – Ada Europe '98*, Springer-Verlag Lecture Notes in Computer Science, Number 1411.

- No locally-declared task objects or task types

- No dynamic allocation (and hence no unchecked deallocation) of task objects or protected objects
- No task termination
- No locally-declared protected objects or protected types
- At most one entry for each protected object/type
- Each entry barrier expression must be a single `Boolean` variable
- At most one task at a time may be queued on an entry
- No `requeue` statements
- No `abort` statements or Asynchronous Transfer of Control
- No `select` statements
- No task entries (and thus no `accept` statements)
- No relative `delay` statements
- No references to package `Ada.Calendar`
- No user-defined task attributes
- No use of dynamic priorities

3.4.2 Ada Features Permitted in the Ravenscar Profiles

The Ravenscar profile allows the following tasking features:

- Task type and task object declarations in library-level packages
- Protected type and protected object declarations in library-level packages
- Absolute delay (`delay until`) statements
- References to package `Ada.Real_Time`
- Pragmas `Atomic` and `Volatile` for shared data
- `Count` attribute (but not in a barrier expression)
- References to package `Ada.Task_Identification` (but no `Abort_Task` or task attribute functions `Callable` or `Terminated`)
- Discriminants for protected types and task types
- Protected procedures as interrupt handlers
- `FIFO_Within_Priority` dispatching policy
- `Ceiling_Locking` locking policy

Implementations may also support non-preemptive or cooperative dispatching policies.

The original Ravenscar Profile from the 1997 International Real-Time Ada Workshop defined only a tasking subset and was silent about restrictions on sequential features. In the context of the GNAT Pro High-Integrity Edition, the term *Ravenscar Profile* means not only the tasking subset just presented, but

also the sequential feature subset defined either by the Zero Footprint Profile (Ravenscar SFP) or the Cert Profile (Ravenscar Cert). Thus the Ravenscar SFP Profile is a proper superset of the Zero Footprint Profile and the Ravenscar Cert Profile is a proper superset of the Cert Profile. A user specifying `ravenscar-sfp` as the '`-RTS=`' option will therefore be able to use the permitted features in the Zero Footprint Profile. A user specifying `ravenscar-cert` or `ravenscar-cert` as the '`-RTS=`' option will be able to use the permitted features in the Cert Profile.

3.4.3 Pragmas Automatically Enabled in the Ravenscar Profiles

The same pragmas automatically enabled in the Zero Footprint profile (see Section 3.2.5 [Pragmas Automatically Enabled in the Zero Footprint Profile], page 28) are also automatically enabled in the Ravenscar SFP profile:

- `pragma Restrictions(No_Exception_Handlers)`

  The Ravenscar SFP Profile uses the same last chance handler scheme as the Zero Footprint Profile. The Ravenscar Cert Profile uses the same scheme as the Cert Profile. The last chance handler is expected to terminate the whole application in both cases.

- `pragma Discard_Names`

  This pragma is enabled for both Ravenscar profiles.

3.4.4 Non-tasking Predefined Packages in the Ravenscar Profiles

The Ravenscar SFP Profile is a superset of the Zero Footprint Profile. So Section 3.2.3 [Predefined Packages in the Zero Footprint Profile], page 27, is still relevant to the Ravenscar Profile, with one difference: '`s-secsta.adb`' is provided. See Section 2.5 [Functions Returning Unconstrained Objects], page 14 for further details.

The Ravenscar Cert Profile is a superset of the Cert Profile. It provides the same packages as the Cert Profile, except for `Ada.Calendar` and `Calendar`, which are disallowed by the Ravenscar profile definition.

3.4.5 Interrupt Handling in the Ravenscar Profiles

Interrupt handling is supported in the Ravenscar Profiles except for the VxWorks 653 target. The mechanism is as defined in Annex D of the *Ada Reference Manual*; the handler is supplied as a parameterless protected procedure, pragma `Attach_Handler` provides static attachment, and `Interrupt_Priority` establishes the priority of the handler.

In the GNAT Pro High-Integrity Edition for VxWorks DO-178B, you can also use the `intConnect()` routine directly, for dynamic attachment of handlers to interrupts.

In either case there is no task switching; the interrupt handler is executed on the stack of the task that was running when the interrupt occurred, at the priority given by the `Interrupt_Priority` pragma.

4 The GNAT Configurable Run Time Facility

This chapter describes how to configure the GNAT run-time library, based on specific application requirements.

4.1 Standard Run-Time Mode

In normal mode, the run-time library supplied by GNAT Pro is complete and provides all features specified in the Ada Reference Manual. As far as is practical, only those sections of the run time that are actually needed are linked, so the entire run time library is not always included. Nevertheless, some minimal required set of run-time units is always linked, and therefore the minimal run-time library in this normal mode contains a significant amount of code that may not be required in all operating configurations.

In this standard mode, the run-time library is required to be complete. If the compiler detects that the run-time library lacks interfaces for required language features, then the run-time library is considered to be improperly configured or installed, and a fatal error message is given.

4.2 The Configurable Run Time

This capability allows the creation of run-time configurations that support only a subset of the full Ada language. There are several reasons for providing this kind of subsetting:

- In bare-board situations, it may be desirable to minimize the amount of run-time code by removing features that are not required. As noted above, this happens to some extent with the standard run-time library, because on most operating systems the linker will link in only those units that are referenced by a given program. However, the configurable option allows much finer control, and much smaller amounts of run-time code end up included in an image.
- When using GNAT Pro in High-Integrity mode, you need to restrict the run-time to units that are certifiable. Since the certification process may require significant resources, it is often desirable to reduce this certification effort by minimizing the run-time.
- It may be desirable for stylistic reasons to restrict the language subset that is used (e.g., to eliminate tasking). This may for example be useful in the case where an application is to be certified, since some features make certification much more difficult. This subsetting can be achieved to some extent using the pragma `Restrictions` mechanism defined in Annex H of the *Ada Reference Manual*. The configurable GNAT run-time facility augments this capability by providing much finer grained support.

- When a given set of `Restrictions` is enforced for a program, it may be possible to simplify the corresponding run-time library. This is done in certain cases when pragma `Restrictions` is specified in full run-time mode, but given the large set of restrictions that can be specified, it is not possible to do this tailoring automatically.

Using the configurable run-time capability, you can choose any level of support from the full run-time library to a minimal *Zero Footprint* Profile which generates no run-time code at all. The units included in the library may be either a subset of the standard units provided with GNAT Pro, or they may be specially tailored to the application.

4.3 Run-Time Libraries and Objects

The run-time libraries implement functionality required by features whose support is not provided by code generated directly by the compiler. The complexity of the run-time library depends on features used and kernel capabilities.

When an Ada program is built, the object code that makes up the final executable may come from the following entities (in addition to the user code itself):

- GNAT Pro run-time library
- C library
- Math library
- Internal GCC library
- Startup code

The GNAT and GCC drivers automatically link all these libraries and objects with the final executable, statically or dynamically depending on the target and on some compilation options. The '`-nostdlib`' and `-nodefaultlibs` options may be used to control this automatic behavior.

GNAT Pro attempts to find these libraries and objects in several standard system directories plus any that are specified with the '`-L`' option or the `LIBRARY_PATH` environment variable. The `gcc --print-search-dirs` command prints the name of the configured installation directory and a list of program and library directories where gcc will search.

The following sections define the contents and purpose of the various elements potentially included in an application's executable.

4.3.1 GNAT Pro Run-Time Library

The high abstraction level and expressiveness provided by the full Ada language requires a rather complex run-time library. This library bridges the semantic gap between the high-level Ada constructs and the low-level C functions and

representations available in the target system (in the form of C headers and libraries). Hence, the semantics of Ada constructs are expanded into calls to a collection of lower-level run-time constructions. An example of this is the implementation of Ada tasking.

This GNAT Pro run-time library comprises both C and Ada files. The C run-time files define a common low-level interface that is implemented on top of the available C headers and libraries in the underlying system. Ada packages within the GNAT Pro run-time library implement the required Ada semantics.

In the case of certifiable systems, it is likely that almost no C file is required.

The GNAT Pro run-time library depends of the following set of libraries:

- C Library (‘`libc.a`’) for a number of miscellaneous functions, such as the input/output system, memory management, etc.
- Math Library (‘`libm.a`’) for everything related to the functionality specified in the Ada Numerics Annex.
- Internal GCC Library (‘`libgcc.a`’) for features such as integer and floating point operations, and exception handling.

4.3.2 C Library

This library provides standard ANSI C functionality in the form of:

- Standard Utility Functions (‘`stdlib.h`’)
- Character Type Macros and Functions (‘`ctype.h`’)
- Input and Output (‘`stdio.h`’)
- Strings and Memory (‘`string.h`’)
- Wide Character Strings (‘`wchar.h`’)
- Signal Handling (‘`signal.h`’)
- Time Functions (‘`time.h`’)
- Locale (‘`locale.h`’)

This C subroutine library depends on a few subroutine calls for kernel or operating system services. If the C library is intended to be used on a system that complies with the POSIX.1 standard (also known as IEEE 1003.1), most of these subroutines are supplied with the operating system or kernel.

For bare board configurations these subroutines are not provided with the system. For other systems, only a fraction of these may be provided. In either case, the user must provide, as a minimum, do-nothing stubs or subroutines with the needed functionality, in order to allow the program to link with the subroutines defined in ‘`libc.a`’. Examples of primitives that `libc.a` may be needed:

Basic input/output capabilities

`open`, `close`, `read`, `write`, `stat`, `fstat`, `link`, `unlink`, `lseek`, `isatty`

Accessing the environment
: `environ`

Process management
: `execve`, `fork`, `getpid`, `times`, `wait`, `kill`, `exit`

Heap management
: `sbrk`

In the case of certifiable systems, most of these capabilities are not needed. Hence, the recommended and simpler approach is that the user implements (in Ada or C) just the required functionality, such as:

Simple Input/Output
: `read`, `write`

Basic memory operations
: `memcpy`, `bcopy`, `memmove`, `memcmp`

Dynamic memory (heap) management
: `malloc`, `free`

4.3.3 Math Library

A complete IEEE math library is usually provided by '`libm.a`', which includes functions that take float, double, and long double parameters. Depending on the type used the function has a different extension. These extensions are named after their full precision equivalents; i.e., `sinf()` is the single precision version of the `sin()` function, and `sinl()` is the long double variant. The reduced precision functions run much faster than their IEEE-compliant double precision counterparts, which can make some floating point operations practical on hardware that is too weak for full double precision computations.

4.3.4 Internal GCC Library

This is a library of internal subroutines that GCC uses to overcome shortcomings of particular machines, or to satisfy the special needs of some languages.

The contents of '`libgcc.a`' are documented in the GCC internals manual and may be inspected with standard binary oriented tools such as `nm` or `objdump`. The whole set can be partitioned into the two major groups that follow.

4.3.4.1 Integer and Floating Point Operations

This represents a fairly large set; documentation for most functions is available in the GCC internals manual and in the GCC sources. This section provides a brief introduction.

The names of these functions have the form `__OpcodeModesNvalues`, where:

- *Opcode* specifies what the function does. E.g. `mul` for a multiplication, `div` for a division.
- *Modes* specifies the GCC machine mode of the operands it operates on. For example:

  `si` Single Integer (4bytes)

  `di` Double Integer (8bytes)

  `sf` Single Float (4bytes)

  `df` Double Float (8bytes)
- *Nvalues* specifies the number of values the function deals with, possibly including a result it computes.

Here are some examples:

`__muldi3` Multiply two DI integers and return the DI result

`__negdi2` Return the negation of a DI integer

`__eqdf2` Return zero if neither argument is NaN and the two (DF) arguments are equal

4.3.4.2 Run-Time Support for Exception Handling and Trampolines

The low-level GCC library also includes everything potentially needed to support a compiler configured to use the GCC scheme for exception handling. These are the functions prefixed by `_Unwind` and `__register_frame`.

Note that only some functions in this set are called "implicitly". Most are explicitly called from the regular run-time libraries for exception-aware languages like C++ or Ada, when configured to use the GCC scheme. Moreover, the High-Integrity Profiles are not configured to use the GCC exception handling scheme (see Section 2.1 [Exceptions and the High-Integrity Profiles], page 8, for details).

In addition, "trampolines" (the GCC low-level mechanism to support pointers to nested subprograms), may require several run-time routines to work properly. The compiler Back End will generate the necessary calls on routines such as `__clear_cache` and `__trampoline_setup`

`pragma Restriction (No_Implicit_Dynamic_Code)` can be used to prohibit pointers to nested subprograms, so that support for trampolines is not required in that case.

4.3.5 Startup / Cleanup Code

The startup / cleanup code is usually found in assembly files named '`crt*.S`' (*crt* stands for "C Run Time"). Their objects are linked at the beginning and at the end of the executable. Their purpose is:

- to perform required program initialization (e.g., initialize hardware, reserve space for stack, zeroing the `.bss` section),
- to bootstrap the rest of the application, and
- to arrange the necessary "cleanup" / finalization after program execution completes.

The '`crt0`' file defines a special symbol like `_start` that is both the default base address for the application and the first symbol in the executable binary image.

The '`crt*.S`' files are normally provided by the operating system. In a bare board configuration it is usually the case that only the initial startup code (such as '`crt0.S`') is needed, and must be provided by the user.

4.4 How Object Dependencies are Generated

4.4.1 Explicit `With` Clauses

The use of `with` clauses creates a dependence relationship between Ada units. This relationship is computed at compilation time and recorded in the '`ali`' file produced for each object. The final executable will contain all the objects corresponding to the units in the dependence closure of the main unit.

This is the simplest and most common way of determining the required set of objects in the final application.

4.4.2 Compiler-Generated Calls to GNAT Pro Run-Time Primitives

When an Ada source file is compiled, the GNAT Pro compiler Front End generates an intermediate representation of the original source code. This is an expanded low-level version of the original source code that can be displayed in an Ada-like format, and can be inspected using the `-gnatD` or `-gnatG` compiler switch.

The expanded code contains calls to the run-time primitives that implement different Ada features. The required run-time library packages are linked to the included hierarchy of library units, in the same way as if an explicit `with` had been used. These dependencies on the GNAT Pro run-time units are also determined at compilation time.

4.4.3 Pragma Import

A pragma Import specifies that the designated entity is defined externally. The use of `pragma Import` clauses forces the inclusion of the required external symbol (and transitively, those that it requires) in the resulting executable file. This dependency is resolved at link time, because it is not possible to know in advance which object file contains the required symbol.

The fact that this dependence is resolved late (at link time, after the binder file has been generated) has a potentially dangerous effect: when an Ada subprogram is imported, the binder does not know where the symbol comes from, and the elaboration code that the imported routine may require will not be called.

4.4.4 Back-End Generated Calls to Library Functions

The GCC back-end may generate "implicit" calls to library subprograms for various reasons. Such calls are said to be implicit because they do not directly correspond to explicit subprogram invocations in the application source code.

Implicit calls to library subprograms occur for several reasons:

a. *Integer and floating point operations*. Some source operations require arithmetic support not available on the target hardware.

b. *Run-time support for exception handling and trampolines*. Some high-level constructs require low-level data structure management too complex to emit inline code for.

c. *Basic memory operations*. Some basic memory operations are too expensive to expand inline, e.g. large block copies or comparisons.

For (a), what the compiler knows about the target hardware may depend on compilation options. For instance, `-msoft-float` triggers calls to library functions for floating point operations even if the hardware has the necessary instructions available. Similarly, the `-mcpu` switch allows modifying the compiler's default view of the target hardware.

The functions to support (a) and (b) are located in '`libgcc.a`', the GCC low-level runtime library built together with the compiler itself.

For (c), the called functions are located in the regular system C library, except for the block comparison function on systems where `memcmp` is not available, in which case, the libgcc `__gcc_bcmp` function is used.

4.5 How The Run Time Library Is Configured

There are three major mechanisms for tailoring the run-time library.

- Use of Configuration Pragmas
- Specification of Configuration Parameters
- Restricting the Set of Run-Time Units

These three mechanisms work together to provide a coherent run-time library that provides a well defined subset. The compiler understands these mechanisms, and will properly enforce the corresponding language subset, providing informative and appropriate messages if features not supported by the subset are used.

4.5.1 Use of Configuration Pragmas

A selected set of configuration pragmas can be placed at the start of package `System`, and enforced for all units compiled in the presence of this `System` package:

```
pragma Detect_Blocking;
pragma Discard_Names;
pragma Locking_Policy (name);
pragma Normalize_Scalars;
pragma Polling (On);
pragma Queuing_Policy (name);
pragma Task_Dispatching_Policy (name);
```

The units provided in the corresponding run-time library need not support language features that would be prohibited by these pragmas.

In addition, `Restrictions` pragmas may be used for all simple restrictions which are required to be applied consistently throughout a partition. The current set of such restrictions is given in the following list. GNAT Pro implements all such restrictions defined in the Ada RM, and, in the list below, the RM reference is given for these restrictions. In addition, GNAT Pro also implements a number of implementation-defined restrictions. See the *GNAT Pro Reference Manual* for details of the meaning of these additional restrictions. This list is taken from the run-time source file '`s-rident.ads`', which should be consulted for the definitive current list for your configuration.

```
Boolean_Entry_Barriers,                -- GNAT (Ravenscar)
No_Abort_Statements,                   -- (RM D.7(5), H.4(3))
No_Access_Subprograms,                 -- (RM H.4(17))
No_Allocators,                         -- (RM H.4(7))
No_Asynchronous_Control,               -- (RM D.7(10))
No_Calendar,                           -- GNAT
No_Delay,                              -- (RM H.4(21))
No_Direct_Boolean_Operators,           -- GNAT
No_Dispatch,                           -- (RM H.4(19))
No_Dynamic_Interrupts,                 -- GNAT
No_Dynamic_Priorities,                 -- (RM D.9(9))
No_Enumeration_Maps,                   -- GNAT
No_Entry_Calls_In_Elaboration_Code,    -- GNAT
No_Entry_Queue,                        -- GNAT (Ravenscar)
No_Exception_Handlers,                 -- GNAT
No_Exception_Registration,             -- GNAT
No_Exceptions,                         -- (RM H.4(12))
No_Finalization,                       -- GNAT
No_Fixed_Point,                        -- (RM H.4(15))
No_Floating_Point,                     -- (RM H.4(14))
No_IO,                                 -- (RM H.4(20))
No_Implicit_Conditionals,              -- GNAT
No_Implicit_Dynamic_Code,              -- GNAT
```

```
   No_Implicit_Heap_Allocations,           -- (RM D.8(8), H.4(3))
   No_Implicit_Loops,                      -- GNAT
   No_Initialize_Scalars,                  -- GNAT
   No_Local_Allocators,                    -- (RM H.4(8))
   No_Local_Protected_Objects,             -- GNAT
   No_Nested_Finalization,                 -- (RM D.7(4))
   No_Protected_Type_Allocators,           -- GNAT
   No_Protected_Types,                     -- (RM H.4(5))
   No_Recursion,                           -- (RM H.4(22))
   No_Reentrancy,                          -- (RM H.4(23))
   No_Relative_Delay,                      -- GNAT (Ravenscar)
   No_Requeue,                             -- GNAT
   No_Secondary_Stack,                     -- GNAT
   No_Select_Statements,                   -- GNAT (Ravenscar)
   No_Standard_Storage_Pools,              -- GNAT
   No_Streams,                             -- GNAT
   No_Task_Allocators,                     -- (RM D.7(7))
   No_Task_Attributes,                     -- GNAT
   No_Task_Hierarchy,                      -- (RM D.7(3), H.4(3))
   No_Task_Termination,                    -- GNAT (Ravenscar)
   No_Tasking,                             -- GNAT
   No_Terminate_Alternatives,              -- (RM D.7(6))
   No_Unchecked_Access,                    -- (RM H.4(18))
   No_Unchecked_Conversion,                -- (RM H.4(16))
   No_Unchecked_Deallocation,              -- (RM H.4(9))
   No_Wide_Characters,                     -- GNAT
   Static_Priorities,                      -- GNAT
   Static_Storage_Size,                    -- GNAT

   Max_Asynchronous_Select_Nesting,        -- (RM D.7(18), H.4(3))
   Max_Entry_Queue_Depth,                  -- GNAT
   Max_Protected_Entries,                  -- (RM D.7(14))
   Max_Select_Alternatives,                -- (RM D.7(12))
   Max_Storage_At_Blocking,                -- (RM D.7(17))
   Max_Task_Entries,                       -- (RM D.7(13), H.4(3))
   Max_Tasks,                              -- (RM D.7(19), H.4(3))
```

No other pragmas are allowed in package `System` (other than the pragma `Pure` for `System` itself which is always present).

4.5.2 Specification of Configuration Parameters

The private part of package `System` defines a number of Boolean configuration switches, which control the support of specific language features.

```
   -----------------------
   -- Target Parameters --
   -----------------------

   --  The following parameters correspond to the constants defined in the
```

```
--  private part of System. Note that it is required that all parameters
--  defined here be specified in the target specific version of system.ads
--  There are no default values.

----------------------------
-- Special Target Control --
----------------------------

--  The great majority of GNAT ports are based on GCC. The switches in
--  This section indicate the use of some non-standard target back end.

AAMP : Boolean;
--  Set to True if target is AAMP.

------------------------------
-- Backend Arithmetic Checks --
------------------------------

--  Divide and overflow checks are either done in the front end or
--  back end. The front end will generate checks when required unless
--  the corresponding parameter here is set to indicate that the back
--  end will generate the required checks (or that the checks are
--  automatically performed by the hardware in an appropriate form).

Backend_Divide_Checks : Boolean;
--  Set True if the back end generates divide checks, or if the hardware
--  checks automatically. Set False if the front end must generate the
--  required tests using explicit expanded code.

Backend_Overflow_Checks : Boolean;
--  Set True if the back end generates arithmetic overflow checks, or if
--  the hardware checks automatically. Set False if the front end must
--  generate the required tests using explicit expanded code.

----------------------------------
-- Control of Exception Handling --
----------------------------------

--  GNAT implements three methods of implementing exceptions:

--    Front-End Longjmp/Setjmp Exceptions

--      This approach uses longjmp/setjmp to handle exceptions. It
--      uses less storage, and can often propagate exceptions faster,
--      at the expense of (sometimes considerable) overhead in setting
--      up an exception handler. This approach is available on all
--      targets, and is the default where it is the only approach.

--      The generation of the setjmp and longjmp calls is handled by
```

```
--      the front end of the compiler (this includes gigi in the case
--      of the standard GCC back end). It does not use any back end
--      support (such as the GCC3 exception handling mechanism). When
--      this approach is used, the compiler generates special exception
--      handlers for handling cleanups when an exception is raised.

--    Back-End Zero Cost Exceptions

--      With this approach, the back end handles the generation and
--      handling of exceptions. For example, the GCC3 exception handling
--      mechanisms are used in this mode. The front end simply generates
--      code for explicit exception handlers, and AT END cleanup handlers
--      are simply passed unchanged to the backend for generating cleanups
--      both in the exceptional and non-exceptional cases.

--      As the name implies, this approach generally uses a zero-cost
--      mechanism with tables, but the tables are generated by the back
--      end. However, since the back-end is entirely responsible for the
--      handling of exceptions, another mechanism might be used. In the
--      case of GCC3 for instance, it might be the case that the compiler
--      is configured for setjmp/longjmp handling, then everything will
--      work correctly. However, it is definitely preferred that the
--      back end provide zero cost exception handling.

--    Control of Available Methods and Defaults

--      The following switches specify whether the ZCX method is
--      available in an implementation, and which method is the default
--      method.

ZCX_By_Default : Boolean;
--  Indicates if zero cost exceptions are active by default. If this
--  variable is False, then the only possible exception method is the
--  front-end setjmp/longjmp approach, and this is the default. If
--  this variable is True, then one of the following two flags must
--  be True, and represents the method to be used by default.

--------------------------------
-- Configurable Run-Time Mode --
--------------------------------

--  In configurable run-time mode, the system run-time may not support
--  the full Ada language. The effect of setting this switch is to let
--  the compiler know that it is not surprising (i.e. the system is not
--  misconfigured) if run-time library units or entities within units are
--  not present in the run-time.

Configurable_Run_Time_On_Target : Boolean;
--  Indicates that the system.ads file is for a configurable run-time
```

```
--
--  This has some specific effects as follows
--
--    The binder generates the gnat_argc/argv/envp variables in the
--    binder file instead of being imported from the run-time library.
--    If Command_Line_Args_On_Target is set to False, then the
--    generation of these variables is suppressed completely.
--
--    The binder generates the gnat_exit_status variable in the binder
--    file instead of being imported from the run-time library. If
--    Exit_Status_Supported_On_Target is set to False, then the
--    generation of this variable is suppressed entirely.
--
--    The routine __gnat_break_start is defined within the binder file
--    instead of being imported from the run-time library.
--
--    The variable __gnat_exit_status is generated within the binder file
--    instead of being imported from the run-time library.

Suppress_Standard_Library : Boolean;
--  If this flag is True, then the standard library is not included by
--  default in the executable (see unit System.Standard_Library in file
--  s-stalib.ads for details of what this includes). This is for example
--  set True for the Zero Footprint case, where these files should not
--  be included by default.
--
--  This flag has some other related effects:
--
--    The generation of global variables in the bind file is suppressed,
--    with the exception of the priority of the environment task, which
--    is needed by the Ravenscar run-time.
--
--    The generation of exception tables is suppressed for front end
--    ZCX exception handling (since we assume no exception handling).
--
--    The calls to __gnat_initialize and __gnat_finalize are omitted
--
--    All finalization and initialization (controlled types) is omitted
--
--    The routine __gnat_handler_installed is not imported

---------------------
-- Duration Format --
---------------------

--  By default, type Duration is a 64-bit fixed-point type with a delta
--  and small of 10**(-9) (i.e. it is a count in nanoseconds. This flag
--  allows that standard format to be modified.
```

```
Duration_32_Bits : Boolean;
--  If True, then Duration is represented in 32 bits and the delta and
--  small values are set to 20.0*(10**(-3)) (i.e. it is a count in units
--  of 20 milliseconds.

-----------------------------------
-- Back-End Code Generation Flags --
-----------------------------------

--  These flags indicate possible limitations in what the code generator
--  can handle. They will all be True for a full run-time, but one or more
--  of these may be false for a configurable run-time, and if a feature is
--  used at the source level, and the corresponding flag is false, then an
--  error message will be issued saying the feature is not supported.

Support_Aggregates : Boolean;
--  In the general case, the use of aggregates may generate calls
--  to run-time routines in the C library, including memset, memcpy,
--  memmove, and bcopy. This flag is set to True if these routines
--  are available. If any of these routines is not available, then
--  this flag is False, and the use of aggregates is not permitted.

Support_Composite_Assign : Boolean;
--  The assignment of composite objects other than small records and
--  arrays whose size is 64-bits or less and is set by an explicit
--  size clause may generate calls to memcpy, memmove, and bcopy.
--  If versions of all these routines are available, then this flag
--  is set to True. If any of these routines is not available, then
--  the flag is set False, and composite assignments are not allowed.

Support_Composite_Compare : Boolean;
--  If this flag is True, then the back end supports bit-wise comparison
--  of composite objects for equality, either generating inline code or
--  calling appropriate (and available) run-time routines. If this flag
--  is False, then the back end does not provide this support, and the
--  front end uses component by component comparison for composites.

Support_Long_Shifts : Boolean;
--  If True, the back end supports 64-bit shift operations. If False, then
--  the source program may not contain explicit 64-bit shifts. In addition,
--  the code generated for packed arrays will avoid the use of long shifts.

-------------------------------
-- Control of Stack Checking --
-------------------------------

--  GNAT provides two methods of implementing exceptions:

--    GCC Probing Mechanism
```

```
--      This approach uses the standard GCC mechanism for
--      stack checking. The method assumes that accessing
--      storage immediately beyond the end of the stack
--      will result in a trap that is converted to a storage
--      error by the runtime system. This mechanism has
--      minimal overhead, but requires complex hardware,
--      operating system and run-time support. Probing is
--      the default method where it is available. The stack
--      size for the environment task depends on the operating
--      system and cannot be set in a system-independent way.

--   GNAT Stack-limit Checking

--      This method relies on comparing the stack pointer
--      with per-task stack limits. If the check fails, an
--      exception is explicitly raised. The advantage is
--      that the method requires no extra system dependent
--      runtime support and can be used on systems without
--      memory protection as well, but at the cost of more
--      overhead for doing the check. This method is the
--      default on systems that lack complete support for
--      probing.

Stack_Check_Probes : Boolean;
--  Indicates if stack check probes are used, as opposed to the standard
--  target independent comparison method.

Stack_Check_Default : Boolean;
--  Indicates if stack checking is on by default

----------------------------
-- Command Line Arguments --
----------------------------

--  For most ports of GNAT, command line arguments are supported. The
--  following flag is set to False for targets that do not support
--  command line arguments (VxWorks and AAMP). Note that support of
--  command line arguments is not required on such targets (RM A.15(13)).

Command_Line_Args : Boolean;
--  Set False if no command line arguments on target

--  Similarly, most ports support the use of an exit status, but AAMP
--  is an exception (as allowed by RM A.15(18-20))

Exit_Status_Supported : Boolean;
--  Set False if returning of an exit status is not supported on target
```

```
----------------------
-- Main Program Name --
----------------------

--  When the binder generates the main program to be used to create the
--  executable, the main program name is main by default (to match the
--  usual Unix practice). If this parameter is set to True, then the
--  name is instead by default taken from the actual Ada main program
--  name (just the name of the child if the main program is a child unit).
--  In either case, this value can be overridden using -M name.

Use_Ada_Main_Program_Name : Boolean;
--  Set True to use the Ada main program name as the main name

-------------------------------------------
-- Boolean-Valued Floating-Point Attributes --
-------------------------------------------

--  The constants below give the values for representation oriented
--  floating-point attributes that are the same for all float types
--  on the target. These are all boolean values.

--  A value is only True if the target reliably supports the corresponding
--  feature. Reliably here means that support is guaranteed for all
--  possible settings of the relevant compiler switches (like -mieee),
--  since we cannot control the user setting of those switches.

--  The attributes cannot dependent on the current setting of compiler
--  switches, since the values must be static and consistent throughout
--  the partition. We probably should add such consistency checks in future,
--  but for now we don't do this.

Denorm : Boolean;
--  Set to False on targets that do not reliably support denormals.
--  Reliably here means for all settings of the relevant -m flag, so
--  for example, this is False on the Alpha where denormals are not
--  supported unless -mieee is used.

Machine_Rounds : Boolean;
--  Set to False for targets where S'Machine_Rounds is False

Machine_Overflows : Boolean;
--  Set to True for targets where S'Machine_Overflows is True

Signed_Zeros : Boolean;
--  Set to False on targets that do not reliably support signed zeros.

OpenVMS : Boolean;
--  Set to True if target is OpenVMS.
```

```
----------------------------------------
-- Boolean-Valued Fixed-Point Attributes --
----------------------------------------

Fractional_Fixed_Ops : Boolean;
--  Set to True for targets that support fixed-by-fixed multiplication
--  and division for fixed-point types with a small value equal to
--  2 ** (-(T'Object_Size - 1)) and whose values have an absolute
--  value less than 1.0.

------------------------------------------------------------
-- Handling of Unconstrained Values Returned from Functions --
------------------------------------------------------------

--  Functions that return variable length objects, notably unconstrained
--  arrays are a special case, because there is no simple obvious way of
--  implementing this feature. Furthermore, this capability is not present
--  in C++ or C, so typically the system ABI does not handle this case.

--  GNAT uses two different approaches

--    The Secondary Stack

--      The secondary stack is a special storage pool that is used for
--      this purpose. The called function places the result on the
--      secondary stack, and the caller uses or copies the value from
--      the secondary stack, and pops the secondary stack after the
--      value is consumed. The secondary stack is outside the system
--      ABI, and the important point is that although generally it is
--      handled in a stack like manner corresponding to the subprogram
--      call structure, a return from a function does NOT pop the stack.

--    DSP (Depressed Stack Pointer)

--      Some targets permit the implementation of a function call/return
--      protocol in which the function does not pop the main stack pointer
--      on return, but rather returns with the stack pointer depressed.
--      This is not generally permitted by any ABI, but for at least some
--      targets, the implementation of alloca provides a model for this
--      approach. If return-with-DSP is implemented, then functions that
--      return variable length objects do it by returning with the stack
--      pointer depressed, and the returned object is a pointer to the
--      area within the stack frame of the called procedure that contains
--      the returned value. The caller must then pop the main stack when
--      this value is consumed.

Functions_Return_By_DSP : Boolean;
--  Set to True if target permits functions to return with using the
```

```
--  DSP (depressed stack pointer) approach.

-----------------
-- Data Layout --
-----------------

--  Normally when using the GCC backend, Gigi and GCC perform much of the
--  data layout using the standard layout capabilities of GCC. If the
--  parameter Backend_Layout is set to False, then the front end must
--  perform all data layout. For further details see the package Layout.

Frontend_Layout : Boolean;
--  Set True if front end does layout

------------------------------
-- Control of Stack Creation --
------------------------------

--  In bare board configurations supporting a static task model (such as
--  Ravenscar), the compiler can create statically (at compile time) the
--  stacks to be used by the different tasks.

Preallocated_Stacks : Boolean;
--  Set to True if the compiler creates statically the stacks for the
--  different tasks. Set to False if stacks are created by the underlying
--  operating system at run time.
```

4.5.3 Restricting the Set of Run-Time Units

Many Ada language features generate implicit calls to the run-time library. For example, if we have the Ada procedure:

```
pragma Suppress (All_Checks);
function Calc (X : Integer) return Integer is
begin
   return X ** 4 + X ** 52;
end Calc;
```

Then the compiler will generate the following code (this is '`-gnatG`' output):

```
with system.system__exn_int;

function calc (x : integer) return integer is
begin
   E1b : constant integer := x * x;
   return integer (E1b * E1b +
                   integer(system__exn_int__exn_integer (x, 52)));
end calc;
```

In the generated code, you can see that the compiler generates direct inlined code for `X ** 4` (by computing `(X ** 2) ** 2`). But the computation of `X ** 52`

requires a call to the runtime routine `System.Exn_Int.Exn_Integer` (the double underlines in the '`-gnatG`' output represent dots in the name).

The full GNAT Pro run-time library contains an appropriate package that provides this capability:

```
-- Integer exponentiation (checks off)

package System.Exn_Int is
pragma Pure (Exn_Int);

   function Exn_Integer
     (Left  : Integer;
      Right : Natural)
      return  Integer;

end System.Exn_Int;
```

If the configurable run-time option is chosen (set `Configurable_Run_Time` to `True` in the `System` spec in file '`system.ads`'), then package `System.Exn_Int` may or may not be present in the run-time library. If it is not present, then the subset of Ada does not allow exponentiation by large integer values, and an attempt to compile `Calc` will result in an error message:

```
1. function Calc (X : Integer) return Integer is
2. begin
3.   return X ** 4 + X ** 52;
                     |
   >>> construct not allowed in this configuration
   >>> entity "System.Exn_Int.Exn_Integer" not defined

4. end Calc;
```

The first line of the error message indicates that the construct is not provided in the library. The second line shows the exact entity that is missing. In this case, it is the entity `Exn_Integer` in package `System.Exn_Int`. This package is in file `s-exnint.ads` (you can use the command `gnatkr system.exn_int.ads` to find this file name). If you look at the spec of this package, you will find the specification of this function:

```
function Exn_Integer
 (Left : Integer; Right : Natural) return Integer;
```

If exponentiation is required, then this package must be provided, and must contain an appropriate declaration of the missing entity. There are two ways to accomplish this. Either the standard GNAT body can be copied and used in the configurable run-time, or a new body can be written that satisfies the specification. Rewriting the body may be useful either to simplify the implementation (possibly taking advantage of configuration pragmas provided in '`system.ads`'), or to meet coding requirements of some particular certification protocol.

In either case, you will have to prepare certification materials for the new package, since the existing certification materials for the run-time library will not include this new package.

Alternatively, you could modify the source code to call an exponentiation routine that is defined within your application:

```
with Exp;
function Calc (X : Integer) return Integer is
begin
  return Exp (X, 4) + Exp (X, 52);
end Calc;
```

where `Exp` is an application function that provides the desired exponentiation capability, and is certified along with the rest of the application in the normal manner.

There are several hundred similar units in the library. For each unit, the unit may or may not be present in the configurable run-time, depending on which facilities are required.

4.6 Naming the Run-Time Library

To assist in keeping track of multiple run-time configurations, the GNAT Pro High-Integrity Edition provides a facility for naming the run-time library. To do this, include a line with the following format (starting in column 4) in '`system.ads`':

```
   Run_Time_Name : constant String := "Simple Run Time 1";
```

The name may contain letters, digits, spaces and underlines. If such a name is provided, then error messages pertaining to the subset include the name of the library:

```
1. function Calc (X : Integer) return Integer is
2. begin
3.   return X ** 4 + X ** 52;
                     |
   >>> construct not allowed in this configuration (Simple Run Time 1)

4. end Calc;
```

4.7 Configuring a Special Purpose Library

As described above, the run-time library may be tailored to suit a specific application. This process can be carried out either by starting with one of the supplied run-time configurations and modifying it, or by starting with the full GNAT Pro library.

A small set of standard units is supplied which can be added to the Zero Footprint library to expand the supported subset, including:

Secondary Stack Support
: This allows functions to return unconstrained results, e.g. arbitrary length strings.

Minimal Exception Support
: This allows for a minimal support for propagation of exceptions

It is of course possible to add any units. However, the configuration of a complex run-time library may be quite difficult, and is best carried out in consultation with experts who are familiar with the structure of the GNAT run-time libraries.

Appendix A GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related

matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading

or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified

Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise

combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Heading 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify,

sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

-

'--RTS=' switch 25, 34
'-E' binder switch 10
-f binder switch 23
-fstack-check compiler switch 22
-fverbose-asm compiler switch 20
-gnata compiler switch 22
-gnatD compiler switch 7, 20
-gnatE compiler switch 10
'-gnatG' 17
-gnatG compiler switch 7, 20
-gnatL compiler switch 20
-gnato compiler switch 22
-gnatP compiler switch 22
-gnatR compiler switch 7, 21
-gnatT compiler switch 22
-gnaty*x* compiler switches 7, 21
-gnatz compiler switch 22
-mregnames compiler switch 20
-nodefaultlibs 38
'-nostdlib' 38
'-nostdlib' switch 26
'-O' compiler switch 21
-S compiler switch 20
-save-temps compiler switch 20
-shared binder switch 22
-static binder switch 22
-t binder switch 23
-T binder switch 23
-Wa,-adhl compiler switch 21

_

__gnat_free 10
__gnat_last_chance_handler 28
__gnat_malloc 10

‘

“**” operator (restrictions on usage in Zero Footprint Profile) 27

6

64-bit fixed-point types (excluded from Zero Footprint profile) 27
64-bit integer types (excluded from Zero Footprint profile) 27
64-bit Numeric Types 31

A

AAMP (configuration parameter) 46
abort statements (excluded from Ravenscar Profile) 33
Abort_Task procedure (excluded from Ravenscar Profile) 33
Absolute delay statements (permitted in Ravenscar Profile) 33
accept statements (excluded from Ravenscar Profile) 33
Ada 2005 Reference Manual 2
Ada 95 Reference Manual 2
Ada.Calendar package (excluded from Ravenscar Profile) 33
Ada.Exceptions.Exception_Name 30
Ada.Exceptions.Last_Chance_Handler 30
Ada.Real_Time package (permitted in Ravenscar Profile) 33
Ada.Tags package (permitted in Zero Footprint profile) 27
Ada.Task_Identification package (permitted in Ravenscar Profile, with restrictions) 33
Ada.Unchecked_Conversion generic (permitted in Zero Footprint profile) .. 27
Ada.Unchecked_Deallocation generic 10
Ada.Unchecked_Deallocation generic (permitted in Zero Footprint profile) .. 27
Allocator 10
and/or usage control 16
Annex E (excluded from Zero Footprint profile) 27
Annex H restrictions 22
Annex H support 7
Array and record assignments and the High-Integrity Profiles 11

Asynchronous Transfer of Control (excluded from Ravenscar Profile) 33
`Atomic` pragma (permitted in Ravenscar Profile) 33
`Attach_Handler` pragma 34
Avoiding elaboration code 17

B

Back-End zero cost exceptions 47
`Backend_Divide_Checks` (configuration parameter) 46
`Backend_Overflow_Checks` (configuration parameter) 46
Bare board configuration 37
`bcopy` 11
Binder switches 22
`Body_Version` attribute (excluded from Zero Footprint profile) 27
Boolean operations on packed arrays (excluded from Zero Footprint Profile) 27

C

`Ceiling_Locking` locking policy (permitted in Ravenscar Profile) 33
`cert` (value for '`--RTS=`' switch) 25
Cert mode 22
Cert profile 25
Cert Profile 1, 29
Cert Profile (included in Ravenscar Cert Profile) 34
Child units (permitted in High-Integrity Profiles) 5
Choosing a Predefined Profile 25
`Command_Line_Args` (configuration parameter) 50
Comparison operations on arrays (excluded from Zero Footprint Profile) 27
Compiler switches 22
conditional operators 16
Configurable run time 37
`Configurable_Run_Time` (configuration parameter) 47
`Configurable_Run_Time` (in package `System`) 54
Configuration pragmas (for tailoring the run time) 44
`Constraint_Error` 9
Controlled types (excluded from Zero Footprint profile) 27
Conventions 3
`Count` attribute (permitted in Ravenscar Profile, with restrictions) 33

D

`delay` statements, absolute (permitted in Ravenscar Profile) 33
`delay` statements, relative (excluded from Ravenscar Profile) 33
`Denorm` (configuration parameter) 51
Discriminants, for protected types (permitted in Ravenscar Profile) 33
Discriminants, for task types (permitted in Ravenscar Profile) 33
Dispatching policy, `FIFO_Within_Priority` (permitted in Ravenscar Profile) 33
Dispatching policy, implementation-defined (permitted in Ravenscar Profile) 33
Distributed Systems Annex (excluded from Zero Footprint profile) 27
DO-178B 5
`Duration_32_Bits` (configuration parameter) 49
Dynamic Allocation 31
Dynamic allocation of protected objects (excluded from Ravenscar profile) 33
Dynamic allocation of tasks (excluded from Ravenscar profile) 33
Dynamic priorities (excluded from Ravenscar Profile) 33

E

Elaboration code 17
Elaboration routine (generated by binder) . . 26
Entries, for protected objects and types (permitted in Ravenscar Profile, with restrictions) 33
Entries, for tasks (excluded from Ravenscar Profile) 33
Entry barriers (permitted in Ravenscar Profile, with restrictions) 33
Entry queue size limit (in Ravenscar Profile) 33

Example - `memcpy` function in Ada 12
Exception declaration (permitted under No_Exception_Handlers) 10
Exception handlers (permitted under No_Exception_Handlers) 9
Exception propagation (excluded from Zero Footprint Profile) 26
Exceptions and the High-Integrity Profiles . . 8
Exceptions and the Last Chance Handler - Cert ProfileCert and Ravenscar Cert Profiles 29
Exceptions and the Last Chance Handler - ZFP and Ravenscar SFP 28
`Exit_Status_Supported` (configuration parameter) 50
Exponentiation 31
Exponentiation (and configurable run time) 54
Exponentiation operator (restrictions on usage in Zero Footprint Profile) 27
`External_Tag` attribute (excluded from Zero Footprint profile) 27

F

`FIFO_Within_Priority` dispatching policy (permitted in Ravenscar Profile) 33
`Fractional_Fixed_Ops` (configuration parameter) 52
Free Documentation License, GNU 57
Front-End longjmp/setjmp exceptions 46
`Frontend_Layout` (configuration parameter) 53
Full-Runtime profile 25
Full-Runtime Profile 1
Functions returning unconstrained objects 14
`Functions_Return_By_DSP` (configuration parameter) 52

G

Generic templates (permitted in High-Integrity Profiles) 5
GNAT Pro High-Integrity Edition for VxWorks DO-178B 11, 34
GNAT Pro Reference Manual 2, 7
GNAT Pro Run-Time Library 38
GNAT Pro User's Guide 2, 22
GNAT Pro User's Guide Supplement for Cross Platforms 2
`gnat.adc` file 11
`GNAT.IO` package (permitted in Zero Footprint profile) 28
`GNAT.Source_Info` package (permitted in Zero Footprint profile) 28
`Gnat_Free` 10
`Gnat_Malloc` 10
`gnatbind` switches 22
GNU Free Documentation License 57

H

High-Integrity Profile program 1
High-Integrity Profiles 1, 5, 22

I

`Image` attribute (excluded from Zero Footprint profile) 27
Implicit conditionals 14
Implicit loops 14
`intConnect` routine 34
`Interfaces` package (permitted in Zero Footprint profile) 27
`Interfaces.C` package (permitted in Zero Footprint profile) 27
`Interfaces.Integer_64` type (excluded from Zero Footprint profile) 27
`Interfaces.Unsigned_64` type (excluded from Zero Footprint profile) 27
Interrupt handling (in the Ravenscar Profiles) 34
`Interrupt_Priority` pragma 34, 35

L

Last Chance Handler 30
`Last_Chance_Handler` 29
Library-level protected objects and types (permitted in Ravenscar Profile) 33
Library-level task objects and types (permitted in Ravenscar Profile) 33
License, GNU Free Documentation 57
Locally-declared protected objects and types (excluded from Ravenscar Profile) 33

Locally-declared task objects and types (excluded from Ravenscar Profile) 32
Locking policy, `Ceiling_Locking` (permitted in Ravenscar Profile) 33
`Long_Long_Integer` (excluded from Zero Footprint profile) 27

M

`Machine_Overflows` (configuration parameter) 51
`Machine_Rounds` (configuration parameter) 51
`Mantissa` attribute (excluded from Zero Footprint profile) 27
`memcpy` 11
`memLib` 11
`memmove` 11
`memNoMoreAllocations` 11
Minimal Exception Support 56
Modes (of GNAT Pro Configuration) 25

N

Naming the run-time library 55
`No_Allocators` restriction 11
`No_Direct_Boolean_Operators` restriction 17
`No_Exception_Handlers` restriction 9
`No_Exceptions` restriction 10
`No_Implicit_Conditionals` restriction .. 7, 14
`No_Implicit_Conditionals`, features excluded by 15
`No_Implicit_Dynamic_Code` restrictions identifier 41
`No_Implicit_Heap_Allocators` restriction 11
`No_Implicit_Loops` restriction 7, 14
`No_Implicit_Loops`, features excluded by .. 16
`No_Local_Allocators` restriction 11
`No_Secondary_Stack` restriction 14
`No_Unchecked_Deallocation` restriction 11
Non-tasking Ravenscar Profile (permitted predefined packages) 34

O

Object-Oriented Programming (and the High-Integrity Profiles) 5
Object-Oriented Programming and the High-Integrity Profiles 13
`OpenVMS` (configuration parameter) 51
Optimization issues 21
Other useful features 21

P

Packed arrays (restrictions in Zero Footprint Profile) 26
`pragma Atomic` (permitted in Ravenscar Profile) 33
`pragma Detect_Blocking` 44
`pragma Discard_Names` 44
`pragma Discard_Names` (automatic in both Ravenscar Profiles) 34
`pragma Discard_Names` (automatic in Zero Footprint Profile) 28
`pragma Locking_Policy` 44
`pragma Normalize_Scalars` 44
`pragma Polling` 44
`pragma Queuing_Policy` 44
`pragma Restrictions` 5, 7, 14, 22, 44
`pragma Restrictions(No_Exception_Handlers)` (automatic in Ravenscar SFP Profile) 34
`pragma Restrictions(No_Exception_Handlers)` (automatic in Zero Footprint Profile) .. 28
`pragma Suppress_Exception_Locations` 29
`pragma Task_Dispatching_Policy` 44
`pragma Volatile` (permitted in Ravenscar Profile) 33
`Preallocated_Stacks` (configuration parameter) 53
Predefined environment, restrictions on 27
Priorities, dynamic (excluded from Ravenscar Profile) 33
Profile 1
`Program_Error` 9
Protected entries (permitted in Ravenscar Profile, with restrictions) 33
Protected objects and types, library-level (permitted in Ravenscar Profile) 33

Protected objects and types, locally-declared (excluded from Ravenscar Profile) 33
Protected procedures as interrupt handlers (permitted in Ravenscar Profile) 33
Protected type discriminants (permitted in Ravenscar Profile) 33

R

`raise` statement (permitted under No_Exception_Handlers) 10
Ravenscar Cert Profile (superset of Cert Profile) 34
Ravenscar mode 22
Ravenscar Profile 5
Ravenscar profiles 25
Ravenscar Profiles 1, 32, 34
Ravenscar Profiles (excluded features) 32
Ravenscar Profiles (permitted features) 33
Ravenscar SFP Profile (superset of Zero Footprint Profile) 34
`ravenscar-cert` (value for '`--RTS=`' switch) 25, 34
`ravenscar-cert-rtp` (value for '`--RTS=`' switch) 25, 34
`ravenscar-sfp` (value for '`--RTS=`' switch) 25, 34
Record and array assignments and the High-Integrity Profiles 11
Relative `delay` statements (excluded from Ravenscar Profile) 33
Removal of Deactivated Code 18
`requeue` statements (excluded from Ravenscar Profile) 33
Reviewable object code 7
`Run_Time_Name` (for configurable run time) 55

S

'`s-secsta.adb`' (package body `System.Secondary_Stack`) 28
'`s-secsta.ads`' (package spec `System.Secondary_Stack`) 14
Secondary Stack 30
Secondary stack (for unconstrained objects returned by functions) 14
Secondary Stack Support 56
`select` statements (excluded from Ravenscar Profile) 33
Sequential features in Ravenscar Profiles ... 34
Shared data (permitted accesses in Ravenscar Profile) 33
`Signed_Zeros` (configuration parameter) ... 51
Slice assignment and implicit loops and conditionals 15
`Stack_Check_Default` (configuration parameter) 50
`Stack_Check_Probes` (configuration parameter) 50
`Storage_Error` 9
`Support_Aggregates` (configuration parameter) 49
`Support_Composite_Assign` (configuration parameter) 49
`Support_Composite_Compare` (configuration parameter) 49
`Support_Long_Shifts` (configuration parameter) 49
`Suppress_Standard_Library` (configuration parameter) 48
`System` package (permitted in Zero Footprint profile) 27
`System.Address_To_Access_Conversions` generic (permitted in Zero Footprint profile) 28
`System.Exn_Int` package 54
`System.Machine_Code` package (permitted in Zero Footprint profile) 28
`System.Secondary_Stack` 14
`System.Storage_Elements` package (permitted in Zero Footprint profile) 28

T

Tagged types at library level (permitted in High-Integrity Profiles) 5
Tagged types at nested levels (excluded from High-Integity Profile) 27
Task attribute functions (excluded from Ravenscar Profile) 33
Task attributes, user-defined (excluded from Ravenscar Profile) 33
Task entries (excluded from Ravenscar Profile) 33

Task objects and types, library-level (permitted in Ravenscar Profile) 33
Task objects and types, locally-declared (excluded from Ravenscar Profile) 32
Task termination (excluded from Ravenscar Profile) 33
Task type discriminants (permitted in Ravenscar Profile) 33
Tasking (excluded from Zero Footprint Profile) 26
Thread Registration 31
Thread Registration Issues 28
Traceability from Source Code to Object Code 20
Trampolines 41
Typographical conventions 3

U

`Unchecked_Deallocation` generic 10
`Unchecked_Deallocation` generic (permitted in Zero Footprint profile) 27
Unconstrained objects, returned by functions 14
`Use_Ada_Main_Program_Name` (configuration parameter) 51
Using GNU GCC 22
Utility Routines 31

V

`Value` attribute (excluded from Zero Footprint profile) 27
`Version` attribute (excluded from Zero Footprint profile) 27
`Volatile` pragma (permitted in Ravenscar Profile) 33

W

What you should know before reading this guide 2
`Width` attribute (excluded from Zero Footprint profile) 27

Z

`ZCX_By_Default` (configuration parameter) 47
Zero Footprint mode 22, 27
Zero Footprint profile 25
Zero Footprint Profile 1, 5, 26, 38
Zero Footprint Profile (excluded features) . . 26
Zero Footprint Profile (included in Ravenscar SFP Profile) 34
Zero Footprint Profile (permitted predefined packages) 27
`zfp` (value for '`--RTS=`' switch) 25

Table of Contents

About This Guide 1
- What This Guide Contains 2
- What You Should Know Before Reading This Guide 2
- Related Information 2
- Conventions 3

1 The High Integrity Philosophy 5

2 Using GNAT Pro Features Relevant to High-Integrity 7
- 2.1 Exceptions and the High-Integrity Profiles 8
- 2.2 Allocators and the High-Integrity Profiles 10
- 2.3 Array and Record Assignments and the High-Integrity Profiles 11
- 2.4 Object-Oriented Programming and the High-Integrity Profiles 12
- 2.5 Functions Returning Unconstrained Objects 14
- 2.6 Controlling Implicit Conditionals and Loops 14
- 2.7 Controlling Use of Conditional Operators 16
- 2.8 Avoiding Elaboration Code 17
- 2.9 Removal of Deactivated Code 18
- 2.10 Traceability from Source Code to Object Code 20
- 2.11 Optimization issues 21
- 2.12 Other useful features 21
- 2.13 Compilation options for the GNAT Pro High-Integrity Tool Chain 22
  - 2.13.1 Compiler Switches 22
  - 2.13.2 `gnatbind` Switches 22

3 The Predefined Profiles 25
- 3.1 Choosing a Predefined Profile 25
- 3.2 The Zero Footprint Profile 26
  - 3.2.1 The '`-nostdlib`' Switch 26
  - 3.2.2 Ada Restrictions in the Zero Footprint Profile 26
  - 3.2.3 Predefined Packages in the Zero Footprint Profile 27
  - 3.2.4 Thread Registration Issues 28
  - 3.2.5 Pragmas Automatically Enabled in the Zero Footprint Profile 28

3.2.6 Exceptions and the Last Chance Handler - ZFP and Ravenscar SFP ... 28
3.3 The Cert Profile ... 29
3.3.1 Exceptions and the Last Chance Handler - Cert and Ravenscar Cert Profiles ... 29
3.3.2 Secondary Stack ... 30
3.3.3 Thread Registration ... 30
3.3.4 Dynamic Allocation ... 31
3.3.5 Exponentiation ... 31
3.3.6 64-bit Numeric Types ... 31
3.3.7 Utility Routines ... 31
3.4 The Ravenscar Profiles ... 32
3.4.1 Ada Restrictions in the Ravenscar Profiles ... 32
3.4.2 Ada Features Permitted in the Ravenscar Profiles ... 33
3.4.3 Pragmas Automatically Enabled in the Ravenscar Profiles ... 34
3.4.4 Non-tasking Predefined Packages in the Ravenscar Profiles ... 34
3.4.5 Interrupt Handling in the Ravenscar Profiles ... 34

4 The GNAT Configurable Run Time Facility ... 37

4.1 Standard Run-Time Mode ... 37
4.2 The Configurable Run Time ... 37
4.3 Run-Time Libraries and Objects ... 38
4.3.1 GNAT Pro Run-Time Library ... 38
4.3.2 C Library ... 39
4.3.3 Math Library ... 40
4.3.4 Internal GCC Library ... 40
4.3.4.1 Integer and Floating Point Operations ... 40
4.3.4.2 Run-Time Support for Exception Handling and Trampolines ... 41
4.3.5 Startup / Cleanup Code ... 41
4.4 How Object Dependencies are Generated ... 42
4.4.1 Explicit `With` Clauses ... 42
4.4.2 Compiler-Generated Calls to GNAT Pro Run-Time Primitives ... 42
4.4.3 Pragma Import ... 42
4.4.4 Back-End Generated Calls to Library Functions ... 43
4.5 How The Run Time Library Is Configured ... 43
4.5.1 Use of Configuration Pragmas ... 44
4.5.2 Specification of Configuration Parameters ... 45
4.5.3 Restricting the Set of Run-Time Units ... 53
4.6 Naming the Run-Time Library ... 55
4.7 Configuring a Special Purpose Library ... 55

Appendix A GNU Free Documentation License.... 57

Index..... 65