Ada Issue 10266 Task Termination Procedure

!standard C.7.3 (01) 05-03-21 AI95-100266-02/10
!standard C.7 (00)
!standard C.7 (01)
!class amendment 01-06-01
!status Amendment 200Y 04-06-25
!status WG9 approved 04-11-18
!status ARG Approved 6-1-2 04-06-13
!status work item 01-06-01
!status received 01-06-01
!priority High
!difficulty Medium
!subject Task termination procedure

!summary

A mechanism is proposed for associating a protected procedure with a task. This procedure is invoked when the task is about to terminate (either normally, as a result of an unhandled exception or due to abort). If a task terminates due to an unhandled exception, the exception occurrence is passed as a parameter to the procedure.

!problem

In Ada 95, a task propagating an exception will be silently terminated.
This can be a significant hazard in high integrity systems.

!proposal

!wording

Change clause C.7 to

Task Information

This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.

Add new section C.7.3

The Package Task_Termination

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
```

```
  pragma Preelaborate (Task_Termination);

  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);

  type Termination_Handler is access protected procedure
    (Cause : in Cause_Of_Termination;
     T     : in Ada.Task_Identification.Task_Id;
     X     : in Ada.Exceptions.Exception_Occurrence);

  procedure Set_Dependents_Fallback_Handler
    (Handler: in Termination_Handler);
  function Current_Task_Fallback_Handler return Termination_Handler;

  procedure Set_Specific_Handler
    (T       : in Ada.Task_Identification.Task_Id;
     Handler : in Termination_Handler);

  function Specific_Handler (T : Ada.Task_Identification.Task_Id)
      return Termination_Handler;

end Ada.Task_Termination;
```

Dynamic Semantics

The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a *handler*. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal and X is set to Null_Occurrence. If the task terminates because of an exception raised by the execution of its task_body, then Cause is set to Unhandled_Exception and X is set to the associated exception occurrence.

Each task has two termination handlers, a *fall-back handler* and a *specific handler*. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be *set* if it is associated with a non-null value of type Termination_Handler, and *cleared* otherwise. When a task is created, its specific handler and fall-back handler are cleared.

The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task; if Handler is null, that fall-back handler is cleared, otherwise it is set to be Handler.all. If a fall-back handler had previously been set it is replaced.

The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise it returns null.

The procedure Set_Specific_Handler changes the specific handler for the task identified by T; if Handler is null, that specific handler is cleared, otherwise it is set to be Handler.all. If a specific handler had previously been set it is replaced.

The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise it returns null.

As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_Id.

An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

Erroneous Execution

For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous.

!discussion

Many safety critical and high integrity systems prohibit (or discourage) exception handling, and so the use of a "when others" handler at the outermost level of the task body is then not available. Furthermore, there may be many tasks in a system, and a systematic, centralized way of handling unhandled exceptions is preferred to having to repeat code in every task body. The proposed solution is applicable at the general language level but it is also appropriate for a Ravenscar environment.

The primary goal of this proposal is to address only the issue of silent termination. Consequently, the proposal does not address issues of termination as a result of failures after task creation but before task activation. It is not clear how useful detection of failure after creation but before activation is, as the declarative region in which the task is being created could fail before the creation point. Hence, the order of elaboration of the declarative region containing the task would dictate whether or not the event would be detected. Note also, failure of task before execution in a Ravenscar environment would cause the program to fail and the problem would need to be handled at a different level.

This proposal allows fall-back handlers (for the active partition) to be assigned for all tasks; or per-task handlers to be set. It does not attempt to define task groups. But a fall-back handler is used for all dependent tasks. Hence setting the fall-back handler for the environment task will cover all tasks in the partition.

With a task hierarchy two programming styles are possible. The first involves replacing the fall-back handler for a part of the tree - this is directly supported by the proposal. The second involves adding to the existing fall-back handler - this can be programmed by first getting the original fall-back handler and then creating a new handler that incorporates the new functionality plus a call to the original handler.

If a null fall-back handler is set then this is equivalent to there being no handler.

The requirement is that the user-supplied protected subprogram is usually called after the task has been finalized but on the stack of the terminating task. Consequently, if Task_Identification.Current_Task is called from the handler, the terminating task identifier is returned.

Note: This proposal provides an alternative to that of AI95-00266-01 which was rejected by IRTAW11; it responds to earlier comments by the ARG. The proposal here introduces fall-back handlers for task hierarchies, but does not go as far as task groups.

!example

The following example illustrates how the mechanisms can be used.

The example is a library package that logs the termination of all tasks and, separately, the terminations due to unhandled exceptions. It also releases a guardian task if any task fails due to being aborted. At the end of the program (when the environment task wishes to terminate) the logged information is output. This example assumes no tasks are declared within library units (or that elaboration control is used to make sure the code of the package body is executed before any library tasks start executing).

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Task_Termination; use Ada.Task_Termination;
with Ada.Exceptions; use Ada.Exceptions;
package Termination_Logging is -- library package
  protected Logger is
    entry Guardian_Control;
    procedure Log_Termination(C : Cause_Of_Termination;
      Id: Task_Id; X: Exception_Occurrence := Null_Occurrence);
    procedure Output(C : Cause_Of_Termination;
      Id: Task_Id; X: Exception_Occurrence := Null_Occurrence);
  private
    Abort_Occurrence : Boolean := False;
    Finished : Natural := 0;
    Exception_Finished : Natural := 0;
  end Logger;
  task Guardian; -- calls Logger.Guardian_Control
end Termination_Logging;

package body Termination_Logging is
  protected body Logger is
    entry Guardian_Control when Abort_Occurrence is
    begin
      Abort_Occurrence := False;
    end Guardian_Control;
    procedure Log_Termination (C : Cause_Of_Termination;
      Id: Task_Id; X: Exception_Occurrence := Null_Occurrence) is
    begin
      Finished := Finished + 1;
      if C = Unhandled_Exception then
        Exception_Finished := Exception_Finished + 1;
      elsif C = Abnormal then
        Abort_Occurrence := True;
      end if;
    end Log_Termination;
    procedure Output(C : Cause_Of_Termination;
      Id: Task_Id; X: Exception_Occurrence := Null_Occurrence) is
    begin
      -- print out the values of Finished and Exception_Finished
    end Output;
    task body Guardian is ...
  end Logger;
begin
  Set_Dependents_Fallback_Handler(Logger.Log_Termination'Access);
  Set_Specific_Handler(Current_Task, Logger.Output'Access);
end Termination_Logging;
```

!corrigendum C.7(00)

@drepl
Task Identification and Attributes
@dby
Task Information

!corrigendum C.7(01)

@drepl
This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined.
@dby
This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.

!corrigendum C.7.3(01)

@dinsc

@i<@s8<Static Semantics>>

The following language-defined library package exists:

```
@xcode<@b<with> Ada.Task_Identification;
@b<with> Ada.Exceptions;
@b<package> Ada.Task_Termination @b<is>
   @b<pragma> Preelaborate (Task_Termination);

   @b<type> Cause_Of_Termination @b<is> (Normal, Abnormal,
Unhandled_Exception);
```

```
   @b<type> Termination_Handler @b<is access protected procedure>
     (Cause : @b<in> Cause_Of_Termination;
      T     : @b<in> Ada.Task_Identification.Task_Id;
      X     : @b<in> Ada.Exceptions.Exception_Occurrence);

   @b<procedure> Set_Dependents_Fallback_Handler
     (Handler: @b<in> Termination_Handler);
   @b<function> Current_Task_Fallback_Handler return Termination_Handler;

   @b<procedure> Set_Specific_Handler
     (T       : @b<in> Ada.Task_Identification.Task_Id;
      Handler : @b<in> Termination_Handler);
   @b<function> Specific_Handler (T : Ada.Task_Identification.Task_Id)
      @b<return> Termination_Handler;

@b<end> Ada.Task_Termination;>
```

@i<@s8<Dynamic Semantics>>

The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a @i<handler>. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal and X is set to Null_Occurrence. If the task terminates because of an exception raised by the execution of its @fa<task_body>, then Cause is set to Unhandled_Exception and X is set to the associated exception occurrence.

Each task has two termination handlers, a @i<fall-back handler> and a @i<specific handler>. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be @i<set> if it is associated with a non-null value of type Termination_Handler, and @i<cleared> otherwise. When a task is created, its specific handler and fall-back handler are cleared.

The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task; if Handler is @b<null>, that fall-back handler is cleared, otherwise it is set to be Handler.@b<all>. If a fall-back handler had previously been set it is replaced.

The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise it returns @b<null>.

The procedure Set_Specific_Handler changes the specific handler for the task identified by T; if Handler is @b<null>, that specific handler is cleared; otherwise it is set to be Handler.@b<all>. If a specific handler had previously been set it is replaced.

The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise it returns @b<null>.

As part of the finalization of a @fa<task_body>, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_ID.

An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

@i<@s8<Erroneous Execution>>

For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous.

!ACATS test

ACATS test(s) need to be constructed for this feature.

!appendix