## D.2.1 The Task Dispatching Model

The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

***Dynamic Semantics***

A task can become a running task only if it is ready (see 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

Task dispatching is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called **task dispatching points**. A task reaches a task dispatching point whenever it becomes blocked, and when it terminates.

Other task dispatching points are defined throughout this Annex for specific policies.

Task dispatching policies are specified in terms of conceptual ready queues and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the **head** of the queue, and the last position is called the ***tail*** of the queue. A task is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

Each processor also has one running task, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. ***The task selected is the one at the head of the highest priority nonempty ready queue***; this task is then removed from all ready queues to which it belongs.

***Implementation Permissions***

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching.

An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

For optimization purposes, an implementation may alter the points at which task

dispatching occurs, in an implementation-defined manner. However, ***a delay_statement always corresponds to at least one task dispatching point***.

***NOTES***

7 Section 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. ***When a task is not ready, it is said to be blocked***.

8 An example of a possible implementation-defined execution resource is a page of physical memory, which needs to be loaded with a particular page of virtual memory before a task can continue execution.

9 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.

10 ***While a task is running, it is not on any ready queue***. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor.

11 *In a multiprocessor system, a task can be on the ready queues of more than one processor*. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

12 The priority of a task is determined by rules specified in this subclause, and under D.1, "Task Priorities", D.3, "Priority Ceiling Locking", and D.5, "Dynamic Priorities".

13 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

## D.2.2 Pragma Task_Dispatching_Policy

***Syntax***

The form of a pragma Task_Dispatching_Policy is as follows:

```
pragma Task_Dispatching_Policy(policy_identifier);
```

***Legality Rules***

The policy_identifier shall either be one defined in this Annex or an implementation-defined identifier.

***Post-Compilation Rules***

A Task_Dispatching_Policy pragma is a configuration pragma.

***Dynamic Semantics***

A task dispatching policy specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues, and whether a task is inserted at the head or the tail of the queue for its active priority. The task dispatching policy is specified by a Task_Dispatching_Policy configuration pragma. If no such pragma appears in any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

***Implementation Permissions***

Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

## D.2.3 The Standard Task Dispatching Policy

This clause defines the policy_identifier, **FIFO_Within_Priorities**.

***Post-Compilation Rules***

If the FIFO_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

***Dynamic Semantics***

When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

* When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.
* When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.
* When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.
* When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

**Each of the events specified above is a task dispatching point** (see D.2.1).

A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority

than the priority of the running task. The currently running task is said to be preempted and it is added at the head of the ready queue for its active priority.

***Documentation Requirements***

Priority inversion is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

* The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and
* whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

***NOTES***

14 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15 Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

## D.2.4 Non-Preemptive Dispatching

A non-preemptive dispatching policy is defined via policy_identifier **Non_Preemptive_FIFO_Within_Priorities**.

***Legality Rules***

Non_Preemptive_FIFO_Within_Priorities can be specified as the policy_identifier of pragma Task_Dispatching_Policy (see D.2.2).

***Post-Compilation Rules***

If the Non_Preemptive_FIFO_Within_Priorities is specified for a partition then Ceiling_Locking (see D.3) shall also be specified for that partition.

***Dynamic Semantics***

When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

* When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.
* When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is moved from the ready queue for its old active priority and is added at the tail of the ready queue for its new

active priority.

* When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

**Each of the events specified above is a task dispatching point** (see D.2.1).

***Implementation Permission***

Since implementations are allowed to round all ceiling priorities in subrange System_Priority to System_Priority'last (see D.3), an implementation may allow a task to execute within a protected object without raising its active priority provided the protected object does not contain pragma Interrupt_Priority, Interrupt_Handler or Attach_Handler.