**European Space Agency Contract Report**

The work described in this report was performed under ESA contract. Responsibility for the contents resides in the author or organisation that prepared it.

# Open Ravenscar Real-Time Kernel

ESTEC/Contract No.13863/99/NL/MV

# Design Definition File

# Software Design Document

Version 1.9 — 20 November, 2001

FOR OPENRAVENSCAR 2.2B

UNIVERSIDAD POLITÉCNICA DE MADRID
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

UNIVERSITY OF YORK
DEPARTMENT OF COMPUTER SCIENCE

CONSTRUCCIONES AERONÁUTICAS, S.A.
DIVISIÓN ESPACIO

**Status:** Final

**Authors:** José F. Ruiz
Juan A. de la Puente
Juan Zamorano
Jesús González-Barahona
Ramón Fernández-Marina

**Revised by:** Ángel Álvarez
Alejandro Alonso

**History**

| Version | Date | Comments |
|---|---|---|
| 1.1 | 1999-11-24 | First version issued for revision |
| 1.2 | 1999-12-03 | Revised after comments from reviewers |
| 1.3 | 2000-01-20 | Changes made according to PDR action items. |
| 1.4 | 2000-02-25 | Issued for internal revision before CDR. |
| 1.5 | 2000-03-07 | Revised after comments from reviewers.<br>Submitted to CDR. |
| 1.6 | 2000-05-12 | Changes made according to CDR action items.<br>Glossary made a separate document. |
| 1.7 | 2000-07-29 | Changes made according to QR&AR action items. |
| 1.8 | 2001-02-20 | Changes made for ORK v2.2. |
| 1.9 | 2001-11-20 | Minor changes made for ORK v2.2b. |

**UPM Development team** Juan Antonio de la Puente
Juan Zamorano
José F. Ruiz
Jesús González-Barahona
Vicente Matellán
Ramón Fernández
Rodrigo García
Andrés Arias
Juan Manuel Dodero
Alejandro Alonso
Ángel Álvarez
José Centeno
Pedro de las Heras

**Project consortium:** Universidad Politécnica de Madrid
Real-Time Systems Group
Department of Telematic Systems Engineering (DIT/UPM).

University of York
Real-Time Systems Group
Department of Computer Science.

Construcciones Aeronáuticas, S.A. (CASA)
Space Division.

# Contents

# Chapter 1

# Introduction

## 1.1 Purpose

The purpose of this document is to describe the design of the *Open Ravenscar Real-Time Kernel* software.

The Open Ravenscar Real-Time Kernel (ORK) is an open-source real-time kernel of reduced size and complexity, for which users can seek certification for mission-critical space applications. The kernel supports Ada 95 applications on an ERC32 based computer. A C interface is also provided.

## 1.2 Scope

This document applies to ork-erc32, a software package based on ORK, a compact real-time kernel for the ERC32 processor with programming interfaces for GNAT, the GNU Ada Compiler, and gcc, the GNU C compiler. Debugging of real-time programs using the kernel is based on gdb, the GNU debugger, and a graphical front-end to interact with it.

The ork-erc32 package includes:

- ORK, the Open Ravenscar Real-Time Kernel itself.
- An adapted cross-development version of GNAT 3.13 targeted to ORK on the ERC32 (SPARC V7) architecture. This version is built from the following components:
  - GNAT 3.13 sources with ORK-ERC32 patches, and special versions of some GNARL (GNU Ada Runtime Library) and all of the GNULL (GNU Lower Level) packages.
  - binutils-2.9.1 sources with ORK-ERC32 patches.
  - newlib-1.8.2 sources with ORK-ERC32 patches.
  - gcc-2.8.1 sources with ORK-ERC32 patches.
- GDB-ORK, an adapted version of GDB 4.17 with ORK-ERC32 patches.
- DDD-ORK, an adapted version of DDD 3.2 with ORK-ERC32 patches.
- MKPROM-ORK, an adapted version of MKPROM for ERC32 with ORK patches.
- RMON-ORK, an adapted version of Remote Debugger Monitor for ERC32 with ORK patches.

- ORK-CIL, the ORK C Interface Library.

## 1.3 Glossary

Acronyms and definitions related to ORK can be found in a separate document, *Open Ravenscar Real-Time Kernel Glossary and documentation guide*, to which the reader is referred.

## 1.4 References

### 1.4.1 Applicable documents

1. ECCS-E40A. Space Engineering — Software [1].
2. Ada 95 Reference Manual [2].
3. Ada 95 — Guidance for High Integrity Systems [3].
4. Alan Burns. The Ravenscar profile [4].
5. C Programming Language [5].
6. POSIX Real-Time Standards [6].

### 1.4.2 Reference documents

1. ERC-32 Manuals [7, 8, 9, 10].
2. ERC-32 GCC Manual [11].
3. Ada 95 — Quality and Style [12].
4. HOOD Reference Manual 3.1 [13].
5. GNAT Manuals [14, 15].
6. Debugging with GDB [16].

Additional references can be found in the bibliography at the end of this volume.

## 1.5 Document overview

This document is organised as follows: chapter 2 makes a general description of the kernel architecture. Chapter 3 describes the top-level design of the system interfaces. Chapter 4 contains the standards, conventions and procedures followed in the design of this product. Chapter 5 provides the software top-level architectural design of the product. Chapter 6 contains a detailed description of each software package. Finally, chapter 7 shows where the code listings are available.

# Chapter 2

# Software overview

## 2.1 The Open Ravenscar Real-Time Kernel

The Open Ravenscar Real-Time Kernel (ORK) is a tasking kernel for the Ada language [2] which provides full conformance with the Ravenscar profile [3, 4] on ERC32-based computers. The kernel has been designed for efficient support of Ada tasking constructs, but it can also be used with C programs. A C interface layer (ORK-CIL) is available for this purpose.

ORK supports the restricted version of Ada tasking defined by the profile, which includes static tasks (with no entries) and protected objects (with at most one entry), a real-time clock and delay until statements, and protected interrupt handler procedures, as well as other tasking features.

The kernel is fully integrated with the GNAT compilation system. The restrictions of the Ravenscar profile are enforced on Ada application programs by means of appropriate restriction pragmas. In this way, conformance with the profile can be secured almost entirely at compile time. The only exceptions are task termination and protected entry call by more than one task, which can only be detected at run time [4].

Debugging support for the ORK kernel, including tasking, is based on an enhanced version of the GDB debugger. A graphical front-end for the debugger is also provided.

The ork-erc32 software has the following components (figure 2.1):

- A specialized version of GNARL, the GNU Ada Runtime Library, from GNAT 3.13 .
- A specialized version of GNULL, the GNU Low-Level Layer, from GNAT 3.13 .
- A C interface layer, based on a subset of pthreads (part of ORK-ERC32 1.0).
- The ORK kernel itself (the main part of ORK-ERC32 1.0).
- An adapted version of GDB 4.17 and DDD 3.2.

## 2.2 The GNU Ada Run-Time Library (GNARL)

The GNU Ada Runtime Library (GNARL) [17] provides tasking support to Ada programs, and is part of the GNAT compilation system. Most of it is independent of the underlying OS and hardware, so that it can be easily ported to new platforms.

GNARL offers a procedural interface (GNARLI) to Ada programs. This interface should not be changed, or the compiler itself would have to be modified.

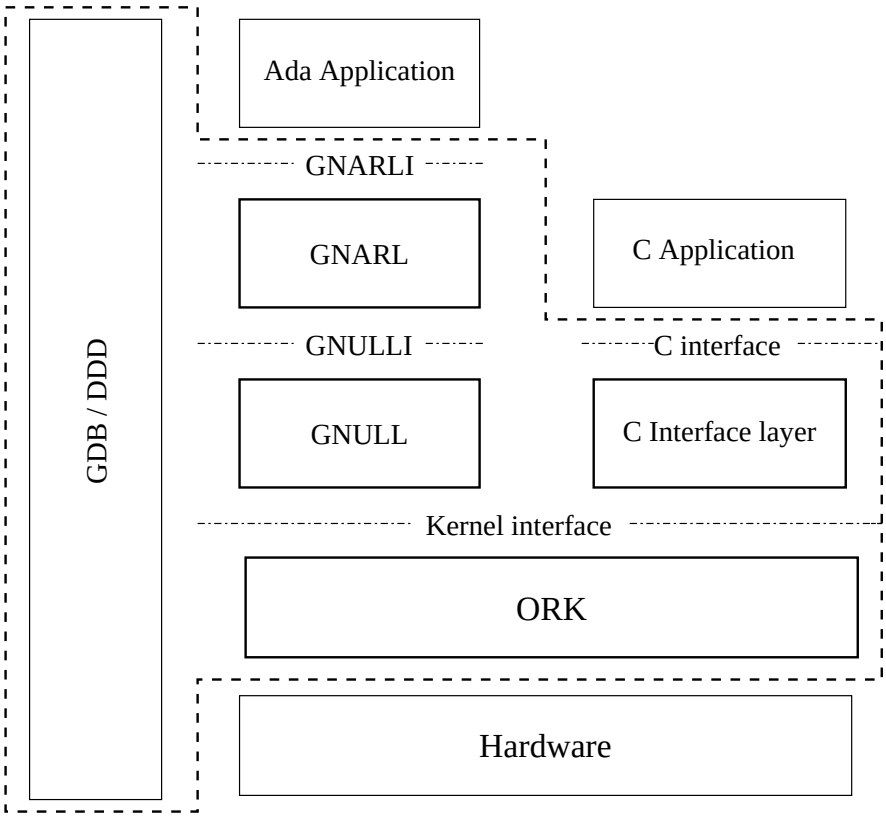


Figure 2.1: Architecture of ORK and main interfaces. The components inside the dotted line are part of the ork-erc32 distribution.

The GNARL packages implement the full Ada tasking model. However, enforcing the Ravenscar profile on a program makes some of GNARL packages unnecessary, and allows simplified versions of others to be used. From GNAT 3.12 on, a simpler implementation of tasks and protected objects for Ravenscar compliant programs is automatically selected when the pragma Ravenscar is in effect.

The specialized version of GNARL for ORK consists of three kinds of packages:

- Standard GNARL packages. These packages are taken unchanged. Most of the GNARL packages are in this category, including all the specifications that make up GNARLI.

- Specific GNARL packages. These packages have been modified in order to adapt them to the Ravenscar profile and ORK specific characteristics.

- New packages that have been added to GNARL in order to support ORK-specific features.

In addition to this, there are some GNARL packages that are not used under the Ravenscar profile restrictions.

## 2.3 The GNU Lower-Level (GNULL) Library

The purpose of GNULL (GNU Low-Level library) is to provide the implementation of low-level services that GNARL needs to request from the underlying operating system.

GNULL provides an interface to the GNARL upper layers called GNULLI (GNULL Interface) which is intended to be OS and hardware independent. Modifying this interface would require changing the upper layer GNARL packages.

The specific version of GNULL for ORK consists of:

- The specifications of some packages which define the GNULL interface (GNULLI). All the interface elements in these specifications have been left unchanged with respect to the current GNAT distribution so that most of the GNARL can be used "as is" (see 2.2 above.).

- The bodies of the GNULL packages, which have been rewritten in order to adapt them to ORK.

The GNULL interface provides much more than is actually needed to implement the restricted Ravenscar tasking functionality. However, in order not to modify the GNARL upper level components and avoid compilation errors, the GNULLI for ORK still contains the full set of operation specifications. The bodies of the superfluous operations raise an exception in order to properly signal violations of the profile at execution time. Notice that this is mainly useful for debugging purposes, as the Ravenscar restrictions should be checked at compilation time by means of appropriate pragmas.

## 2.4 The C interface layer

The purpose of the C interface layer is to provide an application program interface (API) to kernel that can be used from C programs. The interface replicates the functionality of the kernel by means of a set of C type definitions and procedures.

The C interface layer consists of a number of C header (.h) and program (.c) files.

## 2.5 The kernel layer

The kernel layer provides all the required functionality to support real-time programming on top of the ERC32 hardware architecture. The kernel functions can be grouped as follows:

1. Task management, including task creation, synchronization, and scheduling.

2. Time services, including absolute delays and real-time clock.

3. Memory management. The only kinds of dynamic storage allocation supported by the kernel are those required to allocate task control blocks (TCBs) and stack space for tasks at system startup. Deallocation is not supported.

4. Interrupt handling.

The kernel interface to these functions consists of the specifications of some Ada packages, which together make up the kernel interface.

## 2.6 The GNU debugger

The Open Ravenscar Kernel is provided with debugging facilities, based on GDB (GNU debugger.) GDB is widely known as a very portable and powerful debugger, available on many hosts, and capable of debugging many targets. It currently supports source level debugging in various languages, including C, C++ and Ada.

For the purposes of debugging ORK-based programs, some facilities for debugging Ada tasks implemented using ORK have been included in GNARL. Task level debugging is very platform dependent, and therefore specific support for a given task implementation has to be built into GDB. This support is implemented with a set of GDB scripts, providing new task debugging functions. The scripts require support from the kernel, either directly or by means of some GNARL packages which use the kernel information. The debugging interface consists of these GNARL packages, plus some operations defined in the specifications of the GNULLI and kernel interface packages.

A graphical front-end is provided on top of GDB, based on DDD (Data Display Debugger.) DDD is a program designed to act as a simple to use, yet complete, debugging graphic interface, which can interact with several debuggers (including GDB). Some new functionality has been added to it in order to make it a suitable graphical debugger for ORK. This functionality is mainly implemented as a set of patches which enable DDD to support task-level debugging.

# Chapter 3

# System interfaces context

The Open Ravenscar Real-Time Kernel (ORK) provides support for the restricted version of Ada tasking defined by the Ravenscar profile. There are two blocks which are supposed to use these services: the GNULL Layer and the C Interface Layer (see figure 2.1). Both layers use the kernel services through the kernel interface offered by ORK.

The purpose of GNULL is to isolate GNARL from the underlying kernel or operating system. GNULL provides an interface called GNULLI which is assumed to be OS independent. When porting GNARL on top of ORK the GNULL layer translates GNARL calls into ORK primitives.

A C Interface is provided to make ORK callable from C programs. The C Interface Layer provides the appropriate conversion mechanisms to routine calling and parameter passing conventions, to allow C applications to use the ORK primitives easily.

This document is focused on ORK itself, and the kernel internal interface. A detailed description of all the external interfaces can be found in the *Interface Control Document* of this project.

# Chapter 4

# Design standards, conventions, and procedures

## 4.1 Architectural design method

The graphical HOOD notation [13] is used for architectural design and interface description.

According to the small size of the project, the details of the interfaces are written directly in Ada instead of the HOOD textual notation.

## 4.2 Detailed design method

The graphical HOOD notation is also used to represent the system dependencies and its hierarchy.

Names and comments are written in English.

## 4.3 Code documentation standards

The code formatting of Ada source code follows the guidelines described in Ada 95 Quality and Style Guide [12]. This format is also used with assembly code as appropriate.

## 4.4 Naming conventions

Following the guidelines defined in Ada 95 Quality and Style Guide [12], the selection of names is made to clarify the object's or entity's intended use.

## 4.5 Programming standards

ORK is implemented mainly in Ada 95. Assembly language is used for the lowest-level operations.

The guidelines defined in Ada 95 Quality and Style Guide [12] are followed for the code written in Ada 95. The guidelines are also applied to assembly code as far as possible.

The safe subset of Ada used for the implementation of the kernel is defined from the recommendations made by the Ada High Integrity Systems Standard [3]. Notice that the following restrictions apply only to the kernel, not to GNARL or GNULL packages.

Ada features are split into fourteen groups. These groups are categorized into three types:

1. Fully used. The Ada features that were used without any restriction are:

   - Packages (child and library).

2. Partially used. Now it will be listed the groups that are partially used, with a brief description of the concrete features that are forbidden or not used:

   - Types with static attributes.
     - Discriminated records are not allowed, because they can be used to create unconstrained objects, to make some components inaccessible in some variants, and to define indefinite generic formal parameters and private types.
     - Tagged types, and therefore class wide operations, are also forbidden, to avoid the difficulty involved with dispatching operations.
   - Declarations.
     - Complex definitions of aliased objects or components are not used. These are definitions which could render properties of the object inconsistent with non-aliased objects of the same type. Examples of this occur when the original type is indefinite, unconstrained, or modified by representation clauses.
     - Declarative parts in block statements are not used. This feature presents some drawbacks to Flow Analysis and Symbolic Analysis as well as to structural coverage.
   - Names, including scope and visibility.
     - Complex forms of renaming (i.e., those which require run-time evaluation of bounds or object components, or those which extend component lifetime) are forbidden because they hinder Symbolic Analysis, Flow Analysis and Range Checking, and complicate Object Code Analysis as they embed run-time code that has no associated visible source code.
     - Overloading of subprogram is not used to facilitate Flow Analysis, Symbolic Analysis, and Object Code Analysis.
     - Package nesting is not used, because it makes difficult coverage-based testing, and Range Checking becomes problematic.
   - Expressions.
     - Slices of arrays are not used to ease the understanding of the code.
     - Type conversions are only allowed for numeric types. More complex conversions can either generate additional code, or require a temporary object, or require dynamic checks.
   - Statements.
     - goto statements are forbidden, because their use is contrary to all principles of structured programming.
   - Subprograms.

  - Indefinite formal parameters are not used because they may need dynamic storage.
  - Complex return types (indefinite types, unconstrained types, and tagged types) are forbidden because they require dynamic storage techniques.
  - Return statements in procedures are not allowed because they can obscure and cause difficulties for Flow Analysis, Object Code Analysis, etc.

- Arithmetic types
  - Modular integer types are not used, because their predefined operations are not those of classical mathematics, and care is needed to ensure that the operations perform the intended function.

- Low level and interfacing.
  - Unchecked access is forbidden, because it can lead to dangling references or corruption of data.
  - Streams are not used. They require class wide types and access parameters, and are therefore difficult to analyse.

- Access types and types with dynamic attributes.
  - Unconstrained array types are not used.
  - Full access types are forbidden. They need to allocate memory from the heap and other storage areas, making memory use unpredictable, timing analysis problematic, and heap exhaustion and fragmentation a significant risk. It can also create unbounded aliasing problems.
  - Restricted storage pools are not allowed. They are not needed for ORK, and require careful implementation and use to ensure the algorithms are predictable.
  - Controlled types are not used because they introduce hidden control flows due to user-defined initialisation, assignment and finalisation.
  - Indefinite objects are forbidden. They consume time and storage in ways which are difficult, if not impossible, to predict.
  - Non-static array objects are not allowed because time and memory used depends on dynamic bounds.

3. Not used. Finally, it will be shown the features that are not used in the implementation of the kernel. Some of them were not needed at all, and some others were forbidden because they were not considered safe:

   - Generics. Generics are not used, because they are not needed.
   - Exceptions. Exceptions are not used within the kernel.
   - Tasking. Not used.
   - Distribution. Not used within the kernel.

# Chapter 5

# Software top-level architectural design

## 5.1 Overall architecture

The functionality provided by ORK can be divided into the following sets of services:

- Thread management
- Synchronization
- Scheduling
- Storage allocation
- Time-keeping and delays
- Interrupt handling
- Serial output

These sets of functions are defined in different Ada packages. There are also some more packages in the kernel which are used to isolate the hardware dependent aspects. These packages are shown in figure 5.1.

Only five of these packages are designed to be visible to the upper layers. They are: Kernel.Interrupts, Kernel.Time, Kernel.Memory, Kernel.Threads, and Kernel.Serial_Output. The other three packages (Kernel.CPU_Primitives, Kernel.Peripherals, and Kernel.Parameters) are used to implement internal services not available to the external world, isolating machine dependent issues and implementation defined restrictions. The only exception is that package System (which contains the definition of system dependent types and constants) imports some values from Kernel.Parameters.

Kernel primitives in ORK are always non-threaded (interrupts are disabled while accessing the kernel), so that kernel operations are only executed on behalf of a specific user-level thread to which the relevant overhead can thus be charged. There are no implicit threads hidden within the kernel (e.g. to support I/O operations). Actually, there is a thread (called Dummy_Thread) which is automatically created by the kernel to be executed when there is not any other ready thread to execute (see section 5.2.1). However, this thread does not interfere with any other thread in the system, because as soon as there is any ready thread to execute, the Dummy_Thread is immediately preempted.

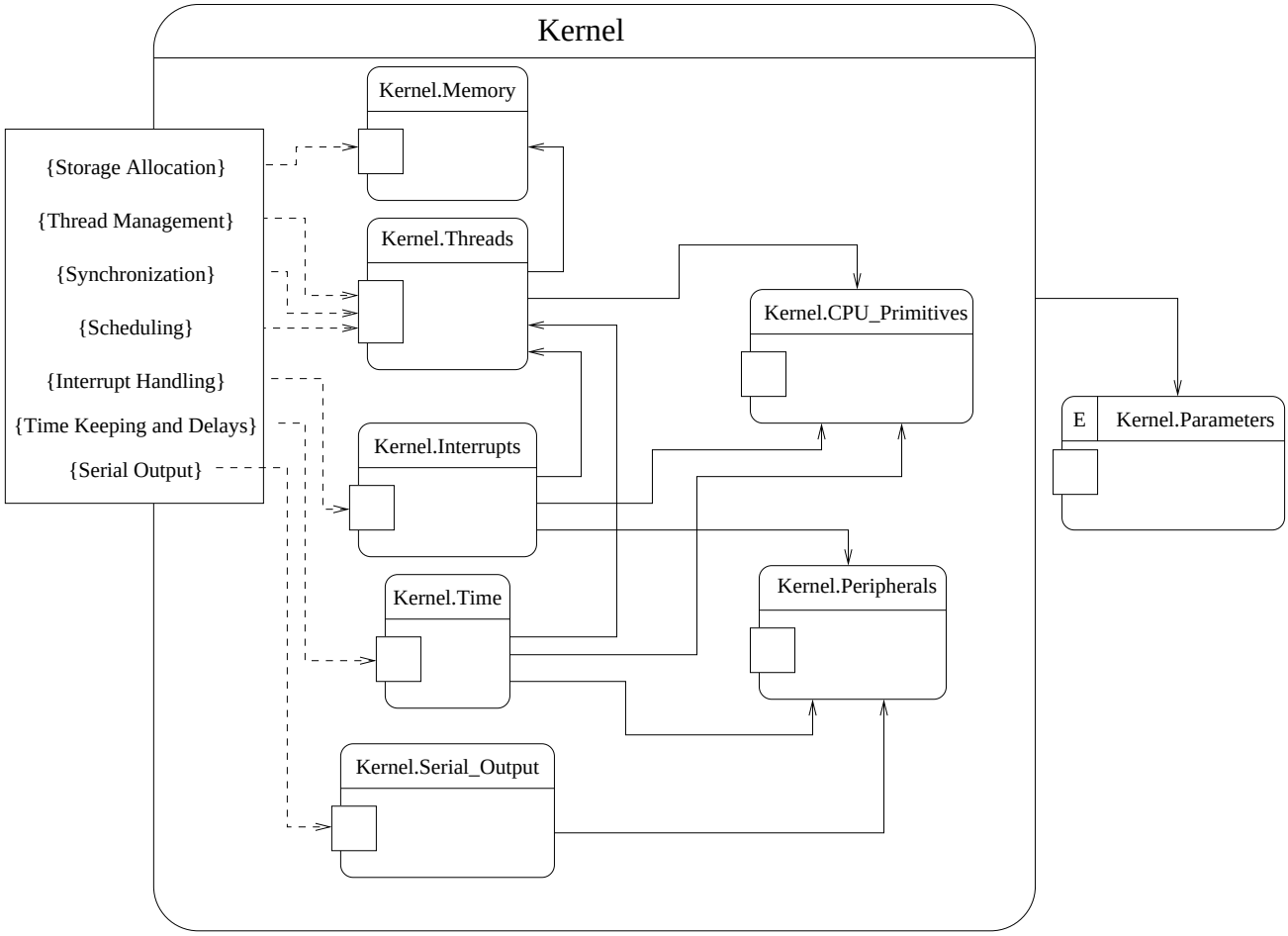


Figure 5.1: Open Ravenscar Real-Time Kernel

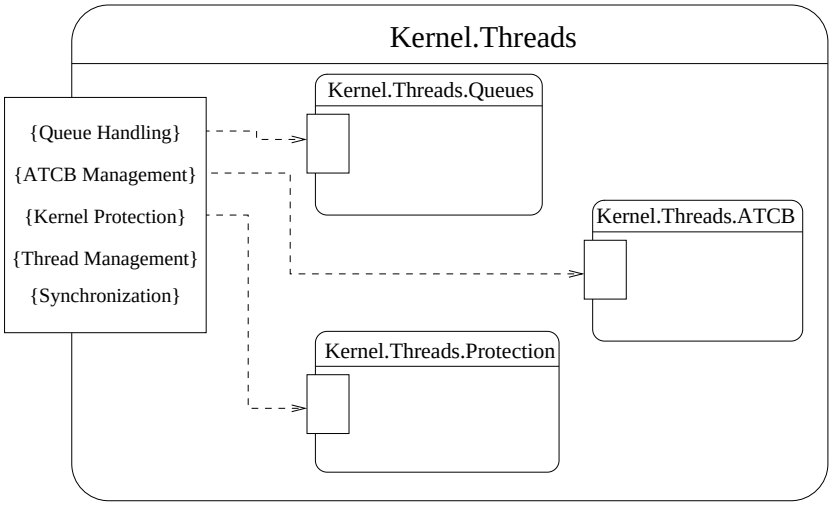


Figure 5.2: Kernel.Threads hierarchy

## 5.2 Software item components

The sets of functions identified in the ORK architecture are implemented by different Ada packages. This section describes the different kernel modules depicted in figure 5.1.

### 5.2.1 The package Kernel.Threads

This package implements the primitives related to thread management, synchronization, and scheduling; it also contains the data definitions related to these services. This package does not depend on the target machine. The data and primitives defined here are visible to both C applications and GNULL.

Kernel.Threads uses three children packages (see figure 5.2) to implement the functionality provided.

This package defines the thread identifiers used both by GNULL and C applications. These identifiers are required by some low-level tasking functions, such as those related to synchronization.

The specification of this package also contains the synchronization elements required by GNULL, not only for the runtime internal data protection, but also for the implementation of protected objects. The types of synchronization elements needed by GNULL are:

- Mutexes. Mutexes are objects which provide access with mutual exclusion to shared data. They implement the Immediate Priority Ceiling Protocol.
- Condition Variables. These objects provide the functionality required by a thread to voluntarily suspend itself to wait for some condition to be satisfied.

The functions implemented by this package are:

- Creation of a concurrent thread of execution to execute the code of an Ada or C task.
- Identification of the currently executing thread.
- Operations to insert, remove, and change the position of a thread within the list of ready threads. These are internal services that cannot be used by any other package or layer.

- Synchronization. The kernel provides primitives to allow thread synchronization using both mutexes and condition variables. Other synchronization methods can be easily implemented using these two objects.

The synchronization primitives provided by ORK are briefly explained in the following paragraphs.

ORK provides operations to acquire and release mutexes following the Immediate Priority Ceiling Protocol. GNULL defines two different procedures to acquire a mutex (Read_Lock and Write_Lock), depending on the kind of access required; several threads can acquire a mutex for reading at a time, but just one thread is allowed to lock a mutex for writing. In a monoprocessor system, such as ORK/ERC32, having different implementations for reading and writing is an unnecessary overhead [18]. Therefore, ORK provides only one primitive to acquire a mutex with Write_Lock semantics. Read_Lock operations are mapped to the same primitive, so that the effect of both operations is exactly the same.

ORK takes great advantage of being targeted primarily to a monoprocessor system, and its implementation of mutexes is very simple and efficient. In case of migration to a multiprocessor, the synchronization primitives should be reimplemented to allow efficient concurrent reading accesses to mutexes.

The kernel protects its internal data avoiding kernel operations to be disturbed by any external interrupt. This way, kernel operations are atomic. GNARL also needs to protect its internal data, but this library relies on kernel primitives (mutex operations) to guarantee the atomic access. As runtime operations are performed at the highest priority, the priority ceiling checking would be unnecessary and this overhead is avoided by using a simpler mutex (called RTS_Lock) with the highest priority, which does not check for ceiling priority violations. However, ORK implements only one type of mutex which always checks ceiling priority violations. Avoiding just one check is not a strong enough reason to implement two different types of mutexes.

The semantics of condition variables have also been dramatically simplified with respect to POSIX [19]. The simplifications are motivated by:

- The maximum number of waiting threads is one.
- There are no timed-wait operations.
- The Ravenscar profile does not allow any other form of awakening threads than signaling a condition variable. Select statements and abortions in the full Ada language make it possible to cancel a waiting operation before signaling the condition.

Therefore, the mechanisms implemented by condition variables to suspend and resume a thread are very simple, without even requiring any queue for storing waiting tasks.

### 5.2.2 The package Kernel.Interrupts

This package is visible both to C applications and GNULL. The implementation of this package is very simple because all the hardware related issues are managed inside the package Kernel.CPU_Primitives.

The interface offered by this package contains the functions to:

- Install interrupt handlers.
- Detach interrupt handlers.

- Obtain the current handler for any interrupt.

An interrupt represents a class of events that are detected by the hardware or the system software. When an interrupt occurs an *Interrupt Service Routine* (ISR) (implemented by package Kernel.CPU_Primitives, see section 5.2.7) is invoked to make the interrupt available to the kernel. The following paragraphs describe the mechanism used by the kernel to handle interrupts.

Protected procedures have appropriate semantics for fast interrupt handlers; they can be directly invoked by the hardware and share data with tasks and other interrupt handlers. The Ravenscar profile does not allow any other form of interrupt handlers.

The type System.Any_Priority represents all the possible priorities in the system. The highest priorities in this range are used to represent the interrupt priorities (type System.Interrupt_Priority). Therefore, hardware priorities are mapped to software priority, providing a unified priority model [20]. This model also implies that tasks with priorities in the range of System.Interrupt_Priority block interrupts with lower priorities.

The SPARC architecture has 15 different interrupt levels which are mapped to the 15 elements of the type System.Interrupt_Priority. Therefore, when a thread executes with a priority within this interrupt range, the interrupts corresponding to the levels below the current interrupt level are disabled by ORK. When a thread changes its currently active priority (due, for example, to the execution of a mutex primitive) the level to which interrupts are enabled also change.

When attaching a protected procedure to an interrupt, once the interrupt handler begins to execute its priority is raised to the ceiling of the protected object. Thus, the handler can only be preempted by other interrupt with a priority higher than the ceiling of the protected object; on the other hand while any shared data (within the protected object which provides the protected procedure handler) is being accessed by other threads of control, all interrupts attached to this protected object are disabled [21], and obvioulsly all the interrupts with a lower priority than the ceiling of that protected object.

This way, ORK schedules interrupt handlers like any other thread in the system; interrupts have the peculiarity that the Immediate Priority Ceiling Protocol guarantees that whenever an interrupt is acknoledged (that is, this is not currently masked) it begins to execute its attached protected procedure being sure that the protected object is always free, as it has been explained in the previous paragraphs.

One important thing that must be taken into account is that when updating internal kernel data, interrupts are disabled (see section 6.1.1). This way the kernel protects all its critical sections, except for non-maskable interrupts, which are used to signal fatal system failures and must be handled immediately.

Notice that the blocking time for interrupts can be easily analysed, accounting the blocking effects due to higher priority tasks and interrupts. The blocking time caused by the execution of kernel operations (mentioned in the previous paragraph) can be modeled the same as accesses to a protected object with the highest priority.

According to the current GNARL implementation, interrupt handlers are executed within the context of especially dedicated “server” tasks, one of them associated to each interrupt. In this way, GNARL implements a unified priority model in which interrupts have their own priorities (in fact the priorities assigned to the respective interrupt handlers). Hence, all interrupt handlers having priorities lower than the active priority of the currently executing task (or interrupt handler) are effectively inhibited. Inhibition will remain while the current active priority is maintained higher or equal to the priority of the interrupt handler.

Ada 95 allows an implementation to handle an interrupt efficiently by arranging for the interrupt handler to be invoked directly by the hardware [2]. Since interrupts may

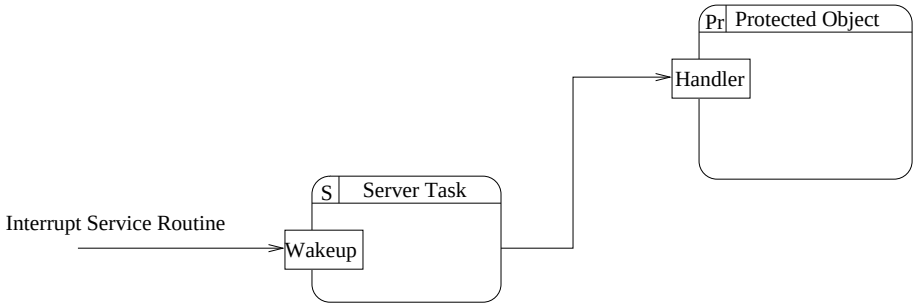


Figure 5.3: Interrupt handling in GNARL

occur very frequently and require fast response, the unnecessary overhead of using server tasks, as mentioned in the previous paragraph, may be intolerable. Attaching protected procedures directly to ISR's would seem at first to be the best solution, however it is not possible to call a protected procedure from an interrupt handler that is not executing within a server task context. Therefore, even if it may appear to be wasteful to interpose a separate task for each interrupt handler, this approach solves the mutual exclusion problem of preventing concurrent execution of the handler procedure with other operations of the same protected object [22], but with a very expensive mechanism. It will be shown later that the ORK kernel can solve the mutual exclusion problem with a much simpler model.

Using server tasks, priorities and mutual exclusion are handled in the standard way for tasks and protected objects. Server tasks also give a clean execution model compared to other approaches in which the handler is executed in the context of the interrupted task. Figure 5.3 shows the mechanism used to call interrupt handlers following this scheme.

Server tasks move the problem of how to ensure mutual exclusion from interrupt handlers to kernel synchronization primitives. They could also increase the level of concurrency allowed inside the kernel.

We could also think about dedicating one server task for all the possible interrupts or providing a server task for each interrupt. Although the former approach saves runtime space, it would block other interrupts during the protected procedure call. For this reason GNARL provides a separate server task for each interrupt which would eliminate the problem of delaying or losing interrupts [22].

This is a good approach for a generic run-time system which must support the full Ada language. But the implementation of this scheme in package System.Interrupts (a member of GNARL) contains tasks with entries which violate the Ravenscar profile. Moreover, in the case of a Ravenscar compliant kernel there are several restrictions that make interrupt handling much simpler:

1. Only protected procedures can be used as interrupt handlers.

2. The only locking policy accepted within protected objects is Ceiling locking.

These simplifications, together with the fact that within our kernel all the interrupts with a lower priority than the currently active priority are masked, make impossible that an interrupt handler is blocked waiting for a protected object to be free. Therefore, there is not need for any server task context to allow the interrupt to wait. The protected procedure can be installed as a low-level asynchronous handler procedure, callable directly from the hardware (see figure 5.4). The effect is that interrupt handlers are executed as if they were directly invoked by the interrupted task, but using the interrupt stack that was mentioned in the begining of this section.

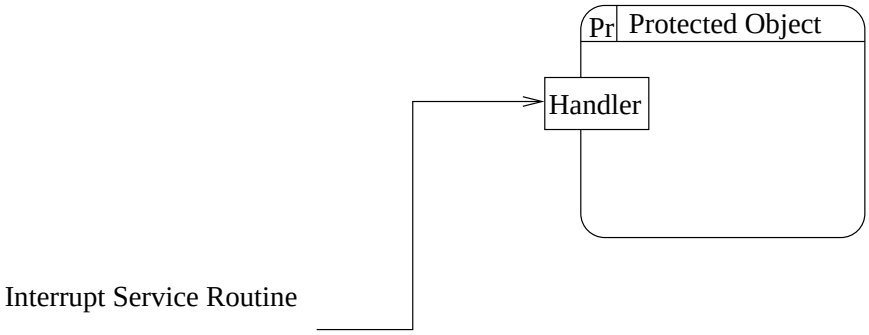


Figure 5.4: Interrupt handling in ORK

This design simplifies not only the conceptual mechanism but also the performance of the system. Obviously, the portability of this solution is reduced as it can not be used for POSIX compliant operating systems. However, as it was mentioned before our target is a bare single processor system and we believe that this solution is fast enough for embedded systems to account for the lack of portability. The current GNARL implementations rely on POSIX signals to handle interrupts although signal semantics are too expensive [23, 24]. Our approach is to directly attach the user handler to the interrupt.

### 5.2.3 The package Kernel.Time

Kernel.Time provides primitives for getting the time from the underlying hardware clock, and the mechanisms for delaying threads until some specified time. These services can be directly used by both the GNULL layer and C applications.

This package is independent from the machines to which the kernel is ported. The implementation of the hardware dependent issues is left to the package Kernel.CPU_Primitives and Kernel.Peripherals.

Delaying mechanisms are quite complex in the full GNARL (that is, the runtime library for the full Ada language), but this has been largely simplified in the restricted kernel.

The current GNARL implementation uses condition variable operations to execute all kinds of delays. This scheme allows timed calls to be canceled before the expiration of the timer. The use of condition variables to implement delay operations in ORK would be unnecessarily expensive, as the profile does not allow for any means of canceling a delay. Therefore, ORK furnishes a simpler way to read the hardware clock and to share the timers among threads. Threads will wait inside the timer queue until their respective expiration time, and there will not be any other event to awake threads.

Delay statements are transformed by GNULL into direct calls to the ORK timer module.

ORK represents the type Time as a signed 64-bit value which represents a number of nanoseconds. The range of time values can uniquely represents the range of real times from program start-up to almost 300 years later, which is consistent with the Real-Time Annex of the Ada Reference Manual (ALRM D.8) [2].

### 5.2.4 The package Kernel.Memory

This package is in charge of the dynamic memory management, and its functionality is visible both to GNULL and C applications. At the initialization of the system, the size and number of some objects (such as stacks or TCB's) are fixed; space for these objects

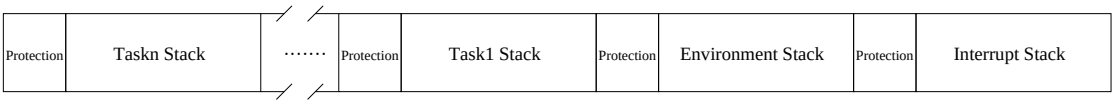


Figure 5.5: Stack layout

is requested at the initialization, but the allocation mechanism is very simple. ORK does not allow for freeing memory, and so memory space is assigned in a contiguous manner without any need to find the right hole for allocating objects.

The Ravenscar profile does not explicitly disallow the use of dynamic memory as this profile only covers tasking related issues, but it seems natural that an application designed following the Ravenscar restrictions should also follow the sequential restrictions defined by the Ada HIS standard [3]. Therefore, these memory functions should only be used at start-up time. However, there is no compiler check for this, and also no runtime check, so it is up to the user not to use dynamic memory after initialization.

The different stacks associated to each task are protected to avoid stack overflow. When a task tries to request more stack than allowed, Storage_Error is raised. The exception is raised when the task performs a write operation within the area named as "protected" in figure 5.5. Read operations within the protected areas do not raise any exception, because unfortunately the ERC32 hardware only implements write access protection.

The MEC in ERC32 allows two different segments to be write protected. One of them is moved when the running thread changes, and the other is fixed to protect always the interrupt stack.

Tasks are therefore allowed to read/write inside the stack space of any other task. At first, ORK was designed to allow task to move its stack pointer only within the bounds of its own stack. But the mechanism used with protected objects did not work with this restriction, because one task may execute a protected entry body on behalf of another task, and the former may modify data that is stored in the latter's stack. This is the model implemented by GNARL for servicing entry queues (allowed by ALRM 9.5.3 par. 22) to minimize unnecessary context switches.

### 5.2.5 The package Kernel.Serial_Output

This package allows applications to display the application output on the user screen. The application sends characters (and strings) through UART channel A, which is connected to the user screen when using the simulator (SIS or TSIM).

Under real targets, using the remote target monitor, an alphanumeric terminal or a communication software (like kermit or tip) can also be attached to UART channel A to show the application output. Remote target monitor uses UART channel B as host-target link.

### 5.2.6 The package Kernel.Parameters

This package contains some types and constants exclusively used by the kernel (and the package System). This package is not visible to GNULL or C programs; GNULL layer uses the package System to extract target dependent information. Here we can find:

- Maximum number of threads allowed.
- The default stack size.

- Maximum space available for the dynamic data to be defined at initialization.
- The priority range, including the band used for interrupt priorities.
- The clock frequency.

This package has no body, and it is used by Kernel.Threads, Kernel.Memory and Kernel.CPU_Primitives.

These parameters are user configurable to allow the kernel to be taylored to a concrete board or application.

### 5.2.7 The package Kernel.CPU_Primitives

The implementation of this package is strongly processor dependent, while it offers the same interface to the rest of kernel packages, providing a machine independent interface. This scheme simplifies porting the kernel to other targets. This package encapsulates functions to:

- Save and restore the machine state for context switches.
- Install trap and interrupt handlers. This function is in charge of inserting the low level Interrupt Service Routine (ISR) within the trap table. The functionality provided by package Kernel.Interrupts uses this target dependent function to isolate dependencies on the target.
- Enable and disable interrupts, as well as changing the level to which interrupts are allowed.

These functions are implemented in assembler. This package is not visible either to GNULL or to C applications.

The main duties of the ISR are changing to the interrupt stack and handling the nesting of interrupts. The ISR implemented by this package is common to all interrupts.

When executing interrupt handlers ORK provides an interrupt stack. The other option is to leave the interrupt to use the stack of the interrupted thread; but this would artificially inflate the stack requirements for each thread, since every thread would have to include enough space to account for the worst case interrupt stack requirements in addition to its own worst case usage. When processing a non-nested interrupt the kernel should switch to the interrupt stack before invoking the handler.

This package also isolates the definition of some target dependent constants:

- Size of the buffer to store the context of the threads.
- Register window size.

### 5.2.8 The package Kernel.Peripherals

This package provides the interface to the peripherals available in the system. It makes easier the porting of the kernel to another target board with different peripherals.

It can be found here the interrupt names related to the different peripherals in the board used.

The peripherals currently handled by ORK are:

- The *General Purpose Timer*.

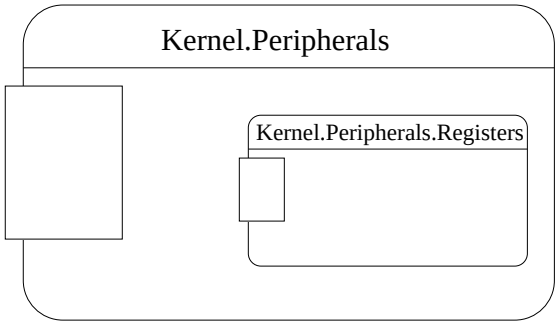


Figure 5.6: Kernel.Peripherals hierarchy

- The *Real Time Clock*.
- The memory controller.
- UART

A child package (Kernel.Peripherals.Registers) is used to isolate the kernel mappings to the different peripheral registers (see figure 5.6).

## 5.3 Internal Interfaces Design

The external interface of the kernel is defined by the specifications of the following Ada packages:

- Kernel.Threads
- Kernel.Time
- Kernel.Interrupts
- Kernel.Memory
- Kernel.Serial_Output

This external interface is largely explained in the *Interface Control Document* of this project.

The kernel also contains some more packages (see figure 5.1) which provide the internal primitives required to implement the kernel functionality. These packages are:

- Kernel.Threads.Queues
- Kernel.Threads.ATCB
- Kernel.Threads.Protection
- Kernel.Parameters
- Kernel.Peripherals
- Kernel.Peripherals.Registers
- Kernel.CPU_Primitives

The specifications of these packages are described in the next sections.

### 5.3.1 The package Kernel.Threads.Queues

The kernel needs two different queues to keep the threads ordered. One of them is the ready queue, where the kernel keeps the ready threads ordered by its priority. The other queue is the one used for timer handling, where all threads that have requested a delay, and are still waiting for it, are ordered by its expiration time. This package provides the functionality required to handle the ready queue and the alarm queue.

```
with Kernel.Time;
-- Used for Time

package Kernel.Threads.Queues is

   -- thread descriptors

   function Dummy_Thread_Id return Thread_Id;

   function Environment_Thread_Id return Thread_Id;

   function Get_New_Thread_Descriptor return Thread_Id;

   -- Ready list

   procedure Insert_At_Head (Thread : Thread_Id);

   procedure Insert_At_Tail (Thread : Thread_Id);

   procedure Extract_From_Ready (Thread : Thread_Id);

   function Next_Running return Thread_Id;

   Running_Thread : Thread_Id;

   -- Alarm list

   procedure Insert_Alarm (T : Kernel.Time.Time;
                           Thread : Thread_Id;
                           Is_First : out Boolean);

   function Extract_First_Alarm return Thread_Id;

   function Get_Next_Alarm_Time return Kernel.Time.Time;

end Kernel.Threads.Queues;
```

The variable Running_Thread defined in this package contains the identifier of the thread that is currently executing. Its value is updated each time a new thread acquire the processor.

This variable is declared in the specification of this package to make it visible to the debugger. Therefore, even if maintaining a shared variable is not the most "elegant" way of providing this information (it would be preferable to provide a function which returns the Running_Thread value) the scripting language used by the debugger places this limitation.

The set of operations related to the ready queue provided by this package are the following:

**Dummy_Thread_Id.** Get the thread identifier associated to the Dummy_Thread. This thread is executed when there is no other ready thread.

**Environment_Thread_Id.** Get the identifier for the environment thread. This thread executes the code of the main procedure of the program.

**Insert_At_Head.** Insert the thread in the ready queue, at the head of its active priority.

**Insert_At_Tail.** Insert the thread in the ready queue, at the tail of its active priority.

**Extract_From_Ready.** Remove the thread from the ready list.

**Next_Running.** Get the identifier of the thread that is placed at the head of the highest active priority in the ready queue.

The alarm queue is handled using the following primitives:

**Insert_Alarm.** Insert the thread in the alarm queue. The queue is ordered by its absolute expiration time. The first place is occupied by the first alarm to be raised.

**Extract_First_Alarm.** Get the identifier of the thread placed at the head of the alarm queue. The thread is also extracted from the alarm queue.

**Get_Next_Alarm_Time.** Return the absolute delay of the first alarm in the queue.

### 5.3.2 The package Kernel.Threads.ATCB

This package is used by GNULL layer to store and get the ATCB associated to each thread. This interface has been moved outside the package Kernel.Threads because these procedures should not be used by C applications.

```
with System;
-- Used for Address

package Kernel.Threads.ATCB is

   procedure Set_ATCB (ATCB : System.Address;
                       Thread_Id : Kernel.Threads.Thread_Id :=
                         Kernel.Threads.Queues.Running_Thread);

   function Thread_Self_ATCB return System.Address;

end Kernel.Threads.ATCB;
```

The operations provided are:

**Set_ATCB.** Store the pointer to the ATCB which owns the thread inside the thread descriptor. It is used to extract Ada task information from a thread identifier.

**Thread_Self_ATCB.** Get the ATCB associated to the currently running thread. This function returns Null_Thread_Id when the thread does not belong to an Ada task.

### 5.3.3 The package Kernel.Threads.Protection

This package is in charge of providing the procedures to allow access with mutual exclusion to the kernel internal data.

---

---

The procedures exported by this package are:

**Enter_Kernel.** Get exclusive read/write access to the kernel data. All interrupts are disabled.

**Leave_Kernel.** Finish the exclusive access to the kernel. This procedure performs a context switch if necessary, and restores the level to which interrupts are disabled (it depends on the active priority of the currently executing thread).

**Dispatch.** Notify to the kernel that the highest priority ready thread could have changed, therefore the dispatcher must be called when leaving the kernel.

### 5.3.4 The package Kernel.Parameters

This package contains types and constants which can be modified by the user to tailor the kernel to a concrete board or application.

---

---

Some of these definitions are dependent on the application. They are:

- Maximum number of tasks.
- Space reserved for stacks. This value has been calculated assuming that it will be used the maximum number of allowed tasks (with the default stack space), and adding the space required for interrupts.
- Default stack size for threads.
- Stack size reserved for the interrupts.
- Range of priorities supported by the kernel.
- The period of the interrupts generated by the Real Time Clock.

There is another value dependent on the board:

- Clock frequency.

The constants dependent on the processor are:

- Number of interrupt levels supported by the processor.
- Maximum value of the interrupt priority range. This value depends on the previously defined number of interrupt levels.
- The range of interrupts supported by the target architecture. There is a one-to-one correspondence between these hardware interrupt levels and software priorities in the range of System.Interrupt_Priority.
- Subtype Range_of_Vector defines the union of external (asynchronous) interrupts and software generated (synchronous) interrupts. This type is used to index the handler table.

### 5.3.5 The package Kernel.Peripherals

This package provides the interface to the peripherals available in the target board. The required types and primitives are defined here.

---

```
with System;
-- used for System.Address

with Kernel.Parameters;
-- used for Clock_Interrupt_Period                                    5
--          Interrupt_Level

package Kernel.Peripherals is

   package KPa renames Kernel.Parameters;                             10

   ---------------
   -- Initialization  -
   ---------------

                                                                      15
   procedure Init_Board;

   ---------------
   -- Clock and timer -
   ---------------                                                    20

   type Timer_Interval is
                range 0 .. Kernel.Parameters.Clock_Interrupt_Period - 1;

   Clock_Freq_Hz : constant Integer := Integer (KPa.Clock_Frequency * 10**6);   25

   procedure Set_Alarm (nanoseconds : Timer_Interval);

   procedure Cancel_And_Set_Alarm (nanoseconds : Timer_Interval);
                                                                      30
   function Read_Clock return Timer_Interval;

   procedure Clear_Alarm_Interrupt;

   procedure Clear_Clock_Interrupt;                                   35

   ----------------
   -- Interrupts -
   ----------------

                                                                      40
   function To_Vector (Level : KPa.Interrupt_Level) return KPa.Range_Of_Vector;

   function Priority_Of_Interrupt (Level : KPa.Interrupt_Level) return
     System.Any_Priority;

                                                                      45
   Watch_Dog_Time_Out      : constant Interrupt_Level := 15;
   External_Interrupt_4    : constant Interrupt_Level := 14;
   Real_Time_Clock         : constant Interrupt_Level := 13;
   General_Purpose_Timer   : constant Interrupt_Level := 12;
   External_Interrupt_3    : constant Interrupt_Level := 11;          50
   External_Interrupt_2    : constant Interrupt_Level := 10;
   DMA_Time_Out            : constant Interrupt_Level := 9;
   DMA_Access_Error        : constant Interrupt_Level := 8;
   UART_Error              : constant Interrupt_Level := 7;
   Correctable_Error_In_Memory : constant Interrupt_Level := 6;       55
```

```
   UART_B_Ready             : constant Interrupt_Level := 5;
   UART_A_Ready             : constant Interrupt_Level := 4;
   External_Interrupt_1     : constant Interrupt_Level := 3;
   External_Interrupt_0     : constant Interrupt_Level := 2;
   Masked_Hardware_Errors   : constant Interrupt_Level := 1;

   --------------------
   -- Memory protection -
   --------------------

   type Protection_Segment_Id is (Segment_1, Segment_2);

   procedure Protect_Segment (base : System.Address;
                              ending : System.Address;
                              id : Protection_Segment_Id);

   ---------------------
   -- Serial output -
   ---------------------

   type UART_Baudrate is
        range Clock_Freq_Hz / (32 * 255 * 2) - 1 .. Clock_Freq_Hz / 32 - 1;

   type UART_Parity is (None, Even, Odd);

   type UART_Stop_Bits is (One, Two);

   type UART_Channel is (A, B);

   procedure Init_UART (baudrate : UART_Baudrate;
                        parity : UART_Parity;
                        stop_bits : UART_Stop_Bits);

   procedure UART_Send (char : Character;
                        channel : UART_Channel);

end Kernel.Peripherals;
```

---

There is one procedure to initialize the board:

**Init_Board.** Initialize the hardware available in the board. This procedure must be invoked when booting the system.

There are also definitions used for time keeping and delays. The type Timer_Interval defines the range of nanoseconds used to represent the time. The following paragraphs describe the primitives used to interact with the clock and timer:

**Set_Alarm.** Arm the timer to raise an interrupt after the number of nanoseconds specified by the argument of this procedure.

**Cancel_And_Set_Alarm.** Cancel a previous alarm and set a new one. This procedure is identical to the previous one (in this implementation), because with the ERC32 setting a new alarm cancel the previous one.

**Read_Clock.** Return the number of nanoseconds since the last clock interrupt.

**Clear_Alarm_Interrupt.** Clear the timer interrupt from the set of pending interrupts. Nothing has to be done in the ERC32, because interrupts are cleared automatically when they are acknowledged.

**Clear_Clock_Interrupt.** The same as the previous procedure but with the clock interrupt. This procedure does nothing in the ERC32 porting of ORK.

The interrupt level for each interrupt is defined here. Interrupt levels are not shared among interrupts, so the level defines uniquely each hardware interrupt. Two functions are also defined for the hardware interrupts associated to these peripherals:

**To_Vector.** This function is used to obtain the right place within the vector table for each interrupt.

**Priority_Of_Interrupt.** This function returns the software priority for each interrupt.

The MEC in ERC32 allows the definition of two different segments that can be write protected. ORK uses this mechanism to protect stack bounds. The type Protection_Segment_Id is defined to differentiate the two segments that can be protected. The function to write-protect segments is:

**Protect_Segment.** This procedure needs the bounds of the segment to be protected, and the identifier used to distinguish the two segments that can be protected. After executing this procedure, if there is any attempt to write within the bounds of any protected segment, a memory exception occurs.

ORK also provides support for sending characters through a serial line. This capability requires the use of the UART. Some types are defined to reflect the configuration of the UART. Type UART_Baudrate defines the range of valid rates. The type UART_Parity contains the values for the parity of the UART. Type UART_Stop_Bits defines whether there are one or two stop bits. There are two channels in the UART, defined by the type UART_Channel.

Two procedures are used to manage the UART:

**Init_UART.** This procedure initializes the UART. The baud rate, parity, and number of stop bits is set.

**UART_Send.** This procedure sends the character received as argument through the specified channel of the UART.

### 5.3.6 The package Kernel.Peripherals.Registers

This package contains the addresses of the memory mapped registers used to configure the peripherals, as well as the range of bits which represents each field inside the registers.

### 5.3.7 The package Kernel.CPU_Primitives

This package isolates the processor dependent primitives. It facilitates the porting to another target.

---

**with** System;
**with** Kernel.Parameters;

```
package Kernel.CPU_Primitives is

   package KPa renames Kernel.Parameters;

   type Context_Buffer is private;

   procedure Context_Switch (Current : access Context_Buffer;
                             Next    : access Context_Buffer;
                             Running_Thread_Id : System.Address);

   procedure Initialize_Context (Buffer : access Context_Buffer;
                                 Program_Counter : System.Address;
                                 Priority : System.Any_Priority;
                                 Stack_Pointer : System.Address;
                                 Stack_Size : Integer);

   procedure Install_Trap_Handler (Service_Routine : System.Address;
                                   Vector : KPa.Range_Of_Vector);

   procedure Install_Interrupt_Handler (Service_Routine : System.Address;
                                        Vector : KPa.Range_Of_Vector);

   procedure Disable_Interrupts;

   procedure Enable_Interrupts (Level : in KPa.Interrupt_Level);

private
   subtype Range_Of_Context is Natural range 1 .. 54;

   type Context_Buffer is array (Range_Of_Context) of System.Address;

end Kernel.CPU_Primitives;
```

---

This package defines a private type (Context_Buffer) which stores the contents of the hardware registers. Two more values (the beginning and end of the stack owned by the thread) are also stored here, to allow the bounded stack protection. In the case of the ERC32 the number of 32-bit registers that make up the state for each thread are:

- 2 Program Counter registers (PC, nPC)
- 8 out registers
- 7 global registers (g0 does not need to be stored, because it is a special register, always returning a zero when read and discarding whatever is written to it)
- The Processor State register (PSR)
- The Multiply/Divide register (Y)
- 32 Floating Point registers
- The FPU Control/Status register (FSR)
- The beginning of the task stack
- The end of the task stack

Therefore, the total amount of 32-bit registers to save per task is 54.
The primitives provided by this package are:

**Context_Switch.** This procedure saves the hardware context of the thread which is leaving the processor, restoring the context of the thread which acquires its ownership. The pointers to the places where storing the context for both tasks are passed as the argument of the procedure. The argument Running_Thread_Id is used to pass the thread identifier of the new running thread, so that when this thread actually acquires the processor, the variable Running_Thread is updated.

**Initialize_Context.** This procedure stores the initial value for the registers.

The first time the thread acquires the processor, the Program Counter, the Stack Pointer, the Frame Pointer, and the Processor State Register have the contents required to execute the code associated to the thread, using its own stack, and executing at the interrupt level required by the active priority of the thread. The values of the beginning and end of the stack are also initialized to allow the stack bounds protection for this thread.

**Install_Trap_Handler.** This procedure install the Service Routine passed as argument as the handler for the synchronous trap specified when calling this procedure.

**Install_Interrupt_Handler.** This procedure is the same as the previous one, installing the Interrupt Service Routine for the specified interrupt.

**Disable_Interrupts.** This procedure disables all the external interrupts, except the non-maskable interrupt.

**Enable_Interrupts.** The processor interrupt level is set to the level specified in this procedure call.

# Chapter 6

# Software components detailed design

This chapter describes the detailed implementation of the kernel packages.

## 6.1 Kernel.Threads

This package is the central component of the ORK architecture. The operations related to the basic tasking functionality are defined here. The primitives for exclusive use by other kernel packages are defined in three children packages.

There is a procedure which initializes the thread environment, called Initialize. Its purpose is to initialize the ready queue, inserting the Environment_Thread and the Dummy_Thread within that queue. The Environment_Thread is the thread which executes the environment code, that is, the main procedure. The Dummy_Thread is an internally used thread which is only selected to execute when there is not any ready threads in the system. As this thread only executes when no other thread is ready to execute, and it is immediately preempted when any other thread becomes ready, its execution does not interfere with the rest of threads.

---

```
type Thread_Descriptor;
-- This type contains the internal information about each thread.

type Thread_Id is access all Thread_Descriptor;
-- This type is used as identifier.

Null_Thread_Id : constant Thread_Id := null;

type Thread_Body is access
  function (arg : System.Address) return System.Address;
-- Pointer to the function that should be executed by the thread.

type Thread_Descriptor is record
   Code : Thread_Body := null;
   Args : System.Address := System.Null_Address;
   ATCB : System.Address := System.Null_Address;
   Context : aliased Kernel.CPU_Primitives.Context_Buffer;
   Base_Priority : System.Any_Priority := System.Any_Priority'First;
   Active_Priority : System.Any_Priority := System.Any_Priority'First;
   Lock_Nesting_Level : Natural := 0;
   Previous : Thread_Id := Null_Thread_Id;
   Next : Thread_Id := Null_Thread_Id;
   Alarm_Time : Kernel.Time.Time := Kernel.Time.Time'Last;
   Next_Alarm : Thread_Id := Null_Thread_Id;
end record;
```

The types used for identifying a thread (Thread_Id) and storing the information about a thread (Thread_Descriptor) are defined in this package. The latter is internally implemented as a pointer to the former. The Thread_Descriptor is a private record which contains the following fields:

- Code. The pointer to the procedure to be executed by the thread.
- Args. The arguments required by the procedure defined above.
- ATCB. The address of the Ada Task Control Block associated to the relevant thread. This field is meaningless when the kernel is used by C applications. The ATCB structure will be described in detail later in this section.
- Context. The space to save the hardware context (stack pointer, program counter, etc.) of the thread when was last preempted. This array also contains the beginning and end of the stack space reserved for each thread, so that each time the running thread changes, the right stack space is write protected (see section 6.7).
- Base_Priority. The base priority of the thread. This priority corresponds to the priority of the thread when it was created, and does not change along the lifetime of the thread because the Ravenscar profile does not allow dynamic priorities.
- Active_Priority. The active priority of the thread. Active priority differs from the base priority due to dynamic priority changes caused by the ceiling locking policy.
- Lock_Nesting_Level. The number of mutexes held by the thread. It is used to know when the base priority must be restored after an Mutex_Unlock operation.
- Previous. Pointers to the revious thread in the ready queue. If the thread is at the head of the queue, this pointer is null.
- Next. Pointers to the next thread in the ready queue. If the thread is at the tail of the queue, this pointer is null. The ready queue is implemented as a doubly linked list, hence the need for two pointers in the thread descriptor.
- Alarm_Time. The time when the alarm expires. If the thread has not a pending alarm the value of this field is set to the maximum time value.
- Next_Alarm. Pointer to the next thread within the alarm queue. The queue is ordered by its absolute expiration time. The first place is occupied by the nearest alarm to expire.

The ATCB definition can be found in package System.Tasking. This type contains the information about Ada tasks.

```
type Common_ATCB is record
   State : Task_States := Unactivated;
   Parent : Task_ID;
   Base_Priority : System.Any_Priority;
   Current_Priority : System.Any_Priority := 0;
   Task_Image : System.Task_Info.Task_Image_Type;
   Call : Entry_Call_Link;
   LL : aliased Task_Primitives.Private_Data;
   Task_Arg : System.Address;
   Stack_Size : System.Parameters.Size_Type;
```

```
   Task_Entry_Point : Task_Procedure_Access;
   Compiler_Data : System.Soft_Links.TSD;
   All_Tasks_Link : Task_ID;
   Activation_Link : Task_ID;
   Activator : Task_ID;                                                    15
   Wait_Count : Integer := 0;
   Elaborated : Access_Boolean;
   Activation_Failed : Boolean := False;
end record;
                                                                           20
type Restricted_Ada_Task_Control_Block (Entry_Num : Task_Entry_Index) is
record
   Common : Common_ATCB;
   Entry_Call : Entry_Call_Record;
end record;                                                                25
```

The type Common_ATCB is used to hold information common to both the restricted GNARL (used for implementing the Ravenscar profile) and the regular version of it.

- State. Encodes the information about the current state of the task. The possible states for a restricted task are Unactivated, Runnable, Activator_Sleep, and Entry_Caller_Sleep.
- Parent. The task on which this task depends. In a Ravenscar compliant program, the only parent allowed is the Environment_Task, because there is no hierarchy of tasks.
- Base_Priority. Base priority of the task. The Ravenscar profile does not allow this value to change.
- Current_Priority. This field is equal to the active priority of the task, except that the effects of protected objects priority ceilings are not reflected.
- Task_Image. Holds an access to string that provides a readable identifier for task, built from the variable of which it is a value or component.
- Call. The entry call that has been accepted by this task. This field should not be placed here (in the common part), because the Ravenscar profile forbids task entries. However, the debugger needs to access to this field easily. Moving this to a different location would require a non trivial amount of work in the debugger.
- LL. Control block used by the underlying low-level tasking service (GNULL).
- Task_Arg. The argument to task procedure. This field is currently unused, but it could provide a handle for discriminant information.
- Stack_Size. Requested stack size.
- Task_Entry_Point. Information needed to call the procedure containing the code for the body of this task.
- Compiler_Data. Task-specific data needed by the compiler to store per-task structures.
- All_Tasks_Link. Used to link this task to the list of all tasks in the system.
- Activation_Link. Used to link this task to a list of tasks to be activated.

- Activator. The task that created this task. This value is set to null if and only if the task has completed activation.
- Wait_Count. This count is used by a task that is waiting for other tasks. At all other times, the value should be zero. It is used differently in several different states, but since a task cannot be in more than one of these states at the same time, a single counter suffices.
- Elaborated. Pointer to a flag indicating that this task body has been elaborated. The flag is created and managed by the compiler generated code.
- Activation_Failed. Set to True if activation of a chain of tasks fails, so that the activator should raise Tasking_Error.

Type Restricted_Ada_Task_Control_Block needs significantly less memory than regular Ada Task Control Block. The Entry_Num discriminant has not been deleted (even when task entries are not allowed in the restricted run time) to keep the same interface as the regular ATCB. This way, minor changes have to be made to the compiler.

The components of the Restricted_Ada_Task_Control_Block are:

- Common. The common part described above.
- Entry_Call. This field is used on entry call "queues" associated with protected objects.

The operations that can be performed on a thread are:

- Creation. This procedure returns the identifier of the new thread. The data that must be passed to the procedure are the code and arguments of the function to be executed by the thread (passed as System.Address to facilitate the use of the kernel by C applications), the priority of the thread and the stack size for this thread.
- Identification. There is a function to query the identifier of the currently running thread.
- Setting the priority. Even when the Ravenscar profile does not allow any form of dynamic priority changes other than caused by the ceiling locking policy, the initialization of a thread needs to modify the priority of the thread to allow the correct initialization of the system.
- Getting the priority. There is a function to query the base priority of a thread.
- Yield. A thread can voluntarily transfer the ownership of the processor to the next ready thread within its active priority queue.

This package also contains the synchronization elements provided by the kernel (mutexes and condition variables), as well as the primitives related to them.

ORK only needs and implements one type of mutex to support the Immediate Priority Ceiling Protocol. Therefore, just two fields are needed:

- The ceiling priority of the mutex.
- The active priority of the thread just prior to acquiring the mutex. This is the priority that must be returned to when releasing the mutex.

As we are using a strictly preemptive scheduling policy for a single processor scheme which does not allow priority ceiling violations, [25, 26] show that the scheduling policy guarantees that there is no way a task can attempt to seize a lock that is held by another suspended or preempted task. Hence, no explicit locking mechanism is required. Operations of high-priority tasks automatically appear atomic to low-priority tasks. No provision has to be made for queue management inside these locks while the procedures to seize and release the locks can also be lightened. We do not need to check if the lock is free or not, because if we attempt to seize a lock it will always be free.

From these observations we can derive that the operations that can be performed on mutexes are described as:

- Initialization. This procedure sets the value of the ceiling priority of the mutex.
- Locking. This procedure is used to acquire the mutex. It suffices to simply update the active priority of the current task to the ceiling priority of the lock used.
- Unlocking. This primitive is used to release the mutex. The active priority of the task needs to be restored, and preemption could occur if there is any other ready task with a higher priority.

The Ravenscar profile does not allow finalization of objects, so there is no kernel primitive for the finalization of mutexes.

LIFO order of unlocking is required (GNARL always follows this policy). It allows a more efficient implementation of mutexes, through the use of a stack structure to save and restore active priorities, and to prevent long-duration blocking through "chaining" of overlapping critical sections.

In the case of condition variables, space would be needed to store the thread that is waiting for the condition to be signaled. As the Ravenscar profile does not allow more than one thread to be waiting on the same condition, no such queue is needed anyway.

If the kernel detects that a thread tries to queue on a condition that is already used by another one, the thread is suspended forever. However, if ORK is used together with GNARL (not by a C application), any attempt to queue on an already used condition raises Program_Error, because this situation is checked by GNARL.

The operations provided for the condition variables are:

- Initialization. The only thing to do for the initialization is to set that there is not any thread waiting.
- Condition_Wait. This procedure suspends the calling thread until another thread signals the condition. Waiting on a condition is always associated to a mutex. The thread must hold that mutex when calling Condition_Wait. The effect of this call is to atomically release the lock and to suspend the thread. The identifier of the calling thread is stored inside the condition variable so as to know which thread to wake up when the condition is signaled. When the thread is awakened the mutex that the thread was holding when the call to Condition_Wait was made is acquired again atomically.
- Condition_Signal. This procedure becomes ready the thread that was waiting for the condition to be signaled, if any. If there is no thread waiting, this call has no effect; this is the semantic implemented by POSIX and therefore, the behaviour expected by GNARL.

Even when the visibility for condition variables should be eliminated and replaced by Sleep and Wakeup operations (for performance and avoidance of error-prone operations) at the GNULL layer [19], support for a semantically reduced condition variables implementation should be provided at least for use by C applications.

This package has an internally defined procedure (Thread_Caller) that acts as a wrapper for the function to be executed by the thread. This procedure is also responsible for extracting the thread from the ready queue if it tries to finalize. Task finalization is a bounded error in the Ravenscar profile and the default action is to suspend the thread forever. GNULL layer can change the actions to take upon task finalization using the procedure Set_Exit_Task_Procedure from package System.Task_Primitives_Operations. This procedure requires an argument which is the parameterless procedure to be executed upon any task finalization.

### 6.1.1 Kernel.Threads.Protection

The variables inside the kernel must be updated in mutual exclusion. There are two procedures to signal that these data are being modified (Enter_Kernel and Leave_Kernel). The first procedure just disables interrupts, so that the following execution cannot be preempted at least until Leave_Kernel is called. The procedure Leave_Kernel enables interrupts to the level corresponding to the currently active priority. Leave_Kernel is also in charge of finding out if as a result of the changes made to the kernel data, the highest priority thread is no longer the same as before. If so, the thread is dispatched.

A dispatching call can be requested by four reasons:

- The thread executing within the processor calls an operation which changes its state to blocked.
- The currently running thread voluntarily transfer the ownership of the processor to the next ready thread within its active priority queue.
- The running thread lowers its priority (when releasing a mutex) and there is a ready thread with a priority higher than the new active priority of the running thread.
- A thread with an active priority higher than the currently active priority becomes ready.

The first three cases are easy to handle, because the thread which triggers the context switch (by calling Leave_Kernel) is the thread that is executing within the processor. When a thread with a higher priority than the currently active priority becomes ready there are some difficulties, because there are two different ways of awaking a thread:

- From the currently running thread.
- From an interrupt handler.

Again, the first case is easy, because the running thread synchronously calls the context switch routine. However, when a thread is awaken from an interrupt handler it must be noticed that the hardware context of the thread that was executing was modified by the Interrupt Service Routine (ISR). Therefore, even if context switches may result from the execution of nested interrupts, their effect is deferred until completion of the interrupt processing (to preserve the context to be saved), and the highest priority thread will acquire the processor on exit from the chain of all nested interrupt handlers.

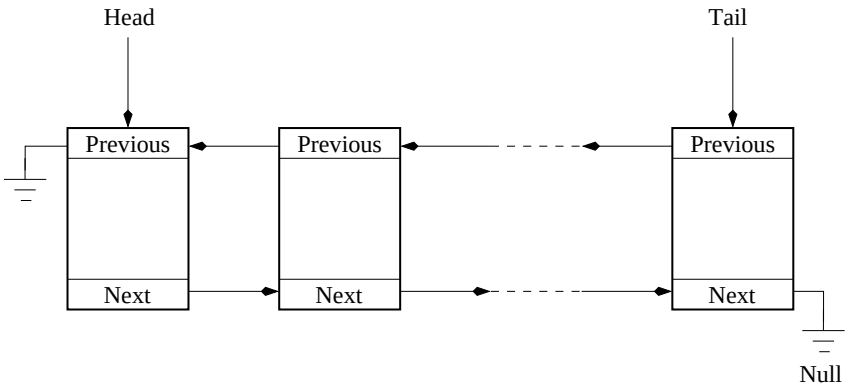


Figure 6.1: Structure of the ready queue

### 6.1.2 Kernel.Threads.Queues

This package is in charge of handling the two different queues available for threads: the ready queue and the timer queue.

The ready queue is modelled as a priority queue, and internally implemented as a doubly linked list. Each element of the list is a thread descriptor pointing to the previous and the next element of the queue. The queue is null-terminated, so the previous element of the first element (or head) of the queue and the next element of the last element (or tail) of the queue are the `Null` identifier. The design of this queue can be seen in figure 6.1.

Space for the maximum number of threads that can exist in the system (256 by default configuration) is statically reserved at initialization.

The primitives that can be executed on the queue are:

- Create a thread descriptor. The first preallocated free thread descriptor is assigned to the caller.
- Insert a thread in the ready queue, either at the head or at the tail of its active priority.
- Remove the thread from the ready queue. The Ravenscar profile restrictions only allow the currently running thread to be removed.
- Get the identifier of the first thread with the highest priority.

This package also stores the identifier of the currently running thread. This variable changes its value whenever a thread is dispatched.

With respect to the alarm queue, it is implemented as a single queue ordered by its expiration time. The first place in the queue is occupied by the alarm which expires first. An internally defined variable is used to store the pointer to this first thread.

The operations implemented for this queue are:

- Insert a new alarm in the queue. This procedure needs the identifier of the thread that is going to wait and the absolute time when the thread must be awaken. This procedure also has an output argument which signal if the thread has been inserted as first within this queue. This value is used to know if the programmed alarm must be changed.
- Extract the first element from the queue. When the timer expires, the element must be deleted. Moreover, the identifier of the thread that was waiting is returned to allow the thread to be reinserted in the ready queue.
- Query the time when the first pending alarm expires.

### 6.1.3 Kernel.Threads.ATCB

The GNULL layer needs to store within each thread descriptor the pointer to the Ada Task Control Block associated to the respective thread. This pointer is used for an efficient implementation of the Self function required by GNULL.

There are two procedures to read and to write the ATCB stored within the thread descriptor. These primitives have been placed in this child package because the ATCB is only needed by GNULL. Thus, when the kernel is being used by a C application it should not be disturbed by GNULL specific issues.

When using GNARL on top of POSIX threads, the functions related to Ada task identification are commonly very inefficient. This is due to the fact that POSIX threads are not specifically designed to execute Ada tasks; the relationship to Ada task is usually implemented using functions to set/query thread-specific data, which impose a big overhead to the widely used Self operation.

## 6.2 Kernel.Interrupts

Our solution to interrupt handling is based on the fact that we only support the Ravenscar profile, and that we do not run on top of a POSIX operating system but on bare hardware. In addition, on the fact that ORK is targeted to a single processor system.

The problem to solve derives from the way GNARL implements Ada interrupt support. It uses tasks with entries, which violate the Ravenscar profile, and the implementation is conditioned by the fact that the caller can get blocked when invoking a protected procedure, so the caller needs to be an Ada task in order to block safely.

Fortunately, thanks to the use of Locking_Policy (Ceiling_Locking), the Ravenscar profile prevents the caller from getting blocked when invoking a protected procedure. The priority of a protected object which has a procedure attached to an interrupt must be at least the hardware Interrupt_Priority of that interrupt (otherwise either the program is erroneous or Program_Error is raised if the priority given falls outside the range of Interrupt_Priority), as it is stated in the Systems Programming Annex of the Ada Reference Manual (ALRM C.3.1 par. 14) [2].

As a result, for as long as the active priority of the running task is equal to or greater than the one of an interrupt, that interrupt will not be recognized by the processor. On the contrary, the interrupt will remain pending until the active priority of the running task becomes lower than the priority of the interrupt, and only then will the interrupt be recognized. It follows that if an interrupt is recognized, then the caller of the protected procedure attached to that interrupt will not be blocked, as the protected object cannot be in use. Otherwise the active priority of the running task would be at least equal to the priority ceiling of the protected object, which cannot be because the interrupt was recognized.

To sum up, the kernel uses protected procedures (together with some kernel prologue and epilogue) as low level interrupt handlers.

Another important implication from this interrupt model is that users should always use distinct priorities for tasks and protected objects with protected handlers; otherwise, tasks could unnecessarily delay the interrupt handling.

The user of package Kernel.Interrupts (whether direct, as for C applications, or indirect, via Ada.Interrupts, as for Ada applications) must provide the address of a parameterless procedure as handler.

This package provides operations to:

- Attach a handler to an interrupt. Each time the interrupt is delivered the handler is executed. If the currently active priority is lower than the interrupt priority the interrupt is immediately delivered to the processor.

- Detach a handler. The previously attached handler is detached, and a default interrupt handler is installed. This default handler is an internal procedure which does nothing.

- Return the current handler for an interrupt.

## 6.3 Kernel.Time

This package is in charge of handling the time keeping and delay primitives.

Time is represented at this level as a 64-bit integer number of nanoseconds. The interval of time values that can be represented in this way is approximately -292..+292 years.

The alarm queue used by this package, as well as the primitives required for its handling, are defined in package Kernel.Threads.Queues (see section 6.1.2).

The implementation of this package was made to provide a high resolution clock with low overhead in timer handling; the combination of a timestamp counter and a high resolution timer contributes to improve the performance and granularity of the time management.

A timestamp counter, built into most modern CPUs, provides the standard time to be used. The representation of time for using in accounting and scheduling is based on the values from this timestamp counter. Linux, as well as most other operating systems maintain a sense of time using a periodic interrupt from a timer chip, which is known as the "heartbeat" of the system. The heartbeat of the Linux kernel is usually 10 ms. Such a coarse grained timing mechanism is insufficient for many real-time applications.

It is very common to implement timers based on a periodic interrupt. For example, when using RTEMS [27] on top of the ERC32, timers are also based on a periodic interrupt (with a user configurable period). In order to provide a more precise timer support, a high resolution timer can usually be implemented by using the single-shot mode of the hardware timers. Therefore, the interrupts are generated on demand, and not periodically.

One of the ways to increase the temporal granularity of a periodic based timer would be to program the timer chip to interrupt the kernel at higher frequencies. This is not an acceptable solution as the overhead increase due to this is tremendous. In fact, we need to program the timer chip to generate interrupts only when there is some scheduled work that needs to be accomplished. The key observation is that even when we want a microsecond resolution, we do not expect to have timing events every microsecond. We therefore need a mechanism by which timer interrupts are allowed to occur at any microsecond, not necessarily every microsecond. This is the RT-Linux [28], KURT [29] and Linux/RK [30] way of handling high resolution timers.

The ERC32 hardware provides two timers (apart from the special *Watchdog* timer) which can be programmed to be either of single-shot type or of periodical type [31]. We use one of them (the *Real Time Clock*) as a timestamp counter and the other (called *General Purpose Timer*) as a high-resolution timer. The former timer provides the basis for a high resolution clock, while the latter offers the required support for precise alarm handling. Both timers are clocked by the internal system clock, and they use a two-stage counter which is shown in figure 6.2. If we call GPTC the *General Purpose Timer*

*Counter* and GPTS the *Scaler*, and SYSCLK the system clock frequency, the timeout for the *General Purpose Timer* before the interrupt occurs is calculated as:

$$Timeout = \frac{(GPTC+1)(GPTS+1)}{SYSCLK}$$

The previous formula has a factor which is (GPTS+1). The +1 term is there because the test for zero occurs before the decrement on SYSCLK. As the minimum value for GPTC is 1 and for GPTS is 0, the minimum Timeout delay is the duration of one clock period.

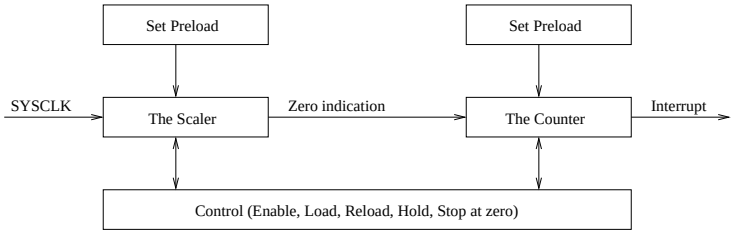


Figure 6.2: Timer design

The *Real Time Clock* is programmed by ORK to interrupt periodically, updating the most significant part of the clock. The less significant part of the clock is held in the hardware clock register. This periodic interrupt is necessary, because of the maximum time space that can be represented using the hardware counter and scaler. This maximum value can be obtained using the highest values for the *Real Time Clock Scaler* (RTCS) and *Real Time Clock Counter* (RTCC), which are 255 (8-bits register) and 4_294_967_295 (32-bits register) respectively. Using a 10 MHz ERC32, the maximum time value that could be represented without using any software register is:

$$Time = \frac{(RTCC+1)(RTCS+1)}{SYSCLK} = \frac{2^{32} \times 2^8}{10^7} = 109\_951 seconds$$

This amount of time is obviously too short, and requires the use of a software register to store the most significant part of the clock.

In order to obtain the highest possible resolution, ORK sets the RTCS preload value to zero. As a result, the resolution of Kernel.Time.Clock is the same as the SYSCLK period, that is 100 ns. The periodic interrupt period (which is given by the RTCC preload value) can be up to 429 s ($= 2^{32}/10^7$). These values are valid for the usual ERC32 system clock frequency of 10 MHz.

The *Real Time Clock* period can be modified by changing the value of Kernel.Parameters.Clock_Interrupt_Period, which represents the integer number of nanoseconds of the desired clock period. Depending on the selected period for the clock interrupt, the overhead imposed to the system changes.

The *General Purpose Timer Counter* is reprogrammed on demand every time an alarm is set, to signal the time when the alarm expires. It does not produce periodic interrupts, but when needed. ORK sets also the GPTC Scaler preload value to zero. As a result, the resolution of Kernel.Time.Delay_Until is the same as the SYSCLK period, that is 100 ns for the usual ERC32 system clock frequency of 10 MHz.

## 6.4 Kernel.Memory

This package is in charge of reserving space for the objects that are known at the initialization of the system. At this point, the size and number of some objects (such as stacks) are

fixed; space for these objects is dynamically requested, but the allocation mechanism is very simple because ORK allocates memory statically and uses a straightforward sequential and contiguous allocation strategy. Memory deallocation is not supported by ORK, so if a program continuously consumes heap it could exaust the dynamic memory.

A contiguous array is defined to store all the stacks in the system, as well as another array to store the rest of the dynamic data. The default size for all the stacks is set to 1_325_056 bytes. This size is calculated assuming that we are using the default maximum number of threads (256 plus the environment and the dummy thread) with the default stack size (5_120 bytes). That value also includes the default interrupt stack size (2_048 bytes) and the protection regions for each stack (256 bytes per each). Those default values are defined in Kernel.Parameters.

It can be specified a different stack size for each task by modifing the Storage_Size attribute of the tasks. The pragma Storage_Size sets the value of Storage_Size to be at least the value specified in the pragma [2, ch. 13.3]. The minimum value for the stack size is defined in Kernel.Parameters.Default_Stack_Size. This value overrides the value specified by the pragma if this were lower.

As it was explained in section 5.2.4 dynamic memory should only be used at start-up, and without allowing deallocation.

## 6.5 Kernel.Serial_Output

This package is in charge of sending characters to the remote host machine. The application output is sent through the UART A, from which the host machine can extract the application output by using a terminal emulator software.

## 6.6 Kernel.Parameters

This package only contains constants to be used internally by the kernel. The kernel can be adapted to the user needs modifying the values defined here.

## 6.7 Kernel.CPU_Primitives

This package contains the primitives which are dependent of the underlying processor. There is another package Kernel.Peripherals which isolates the kernel from the peripherals installed in the target machine.

The functionality provided by this package is:

- Save and restore the machine state for context switches.
- Install the low level Interrupt Service Routine for trap and interrupt.
- Enable and disable interrupts, as well as changing the level to which interrupts are allowed.

Those functions are implemented in assembler, and imported to the Ada code.

Stack checking mechanism is provided. When a task tries to request more stack than allowed an exception (Storage_Error) is raised. The mechanism is implemented using the memory access protection functionality provided by the MEC in ERC32. Two different segments can be write protected; one of them is placed at the lower bound of the currently

active stack to detect any request of stack outside its limit, and the other protects the interrupt stack.

The kernel inserts a small prologue and epilogue to the user interrupt handler, to allow the correct execution of the interrupted thread. As nesting of interrupts is allowed (an interrupt can be recognized while processing a lower priority interrupt), the prologue changes the current stack to the interrupt stack, if the interrupt is not nested, and stores the floating point context. The epilogue is in charge of performing a context switch to the highest priority thread (if this is not the currently running thread) when the most external interrupt has finished its execution.

### 6.7.1 Fast context switch

The SPARC V7 has a total of 167 user-allocable registers and 128 of these are used for the overlapping register windows. The 128 window registers are grouped into eight sets of 24 registers called windows (see figure 6.3).

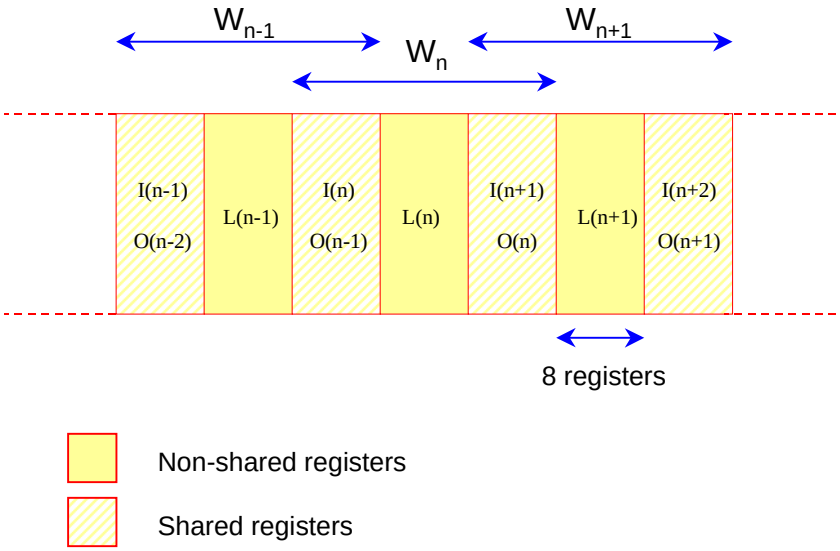


Figure 6.3: Overlapping windows (24 registers per window)

The first eight registers in a window are called *in* registers, and the last eight are the *out* registers, and the eight registers that are between the in and out registers are called *local* registers. In figure 6.3, I(n), L(n), and O(n) represent the *in*, *local*, and *out* registers of window n respectively. At any time, only one window is visible. The other registers are comprised of 7 global registers and 32 floating-point registers.

When calling a subroutine the visible window changes from $W_n$ to $W_{n+1}$ (using the save instruction) to provide new registers for the new subroutine. On subroutine return, the previous register mapping is restored (with the restore instruction). As shown in figure 6.3 adjacent windows have common registers, so that the in registers overlap with the previous window, and the out registers overlap with the following window.

It can be seen that the first 16 registers in a window are non-scratch since their values will be retained across function calls; we can be sure that we can use them safely in our scope, regardless of the registers used by the functions called. The last 8 are scratch since their values cannot be guaranteed upon return from the called function [32]; if we call a function which modifies its in registers, the out registers of the caller are therefore modified.

This overlap of window registers is used as an efficient means of passing parameters during procedure calls and returns. There is a 5-bit field in the *Processor State Register* (PSR), called *Current Window Pointer* (CWP), that points to the currently active window (the window visible to the programmer).

During a context switch, the register windows of the current thread must be flushed onto the thread stack before one window will be loaded with the top frame of the new thread.

There are two different approaches to follow for the flushing policy. You could flush all register windows or just the windows currently in use [20]. The implementation of the context switch under *SunOS 4.x* simply flushes all register windows of the processor. For a scheme with frequent context switches it is less likely that a thread uses all of the windows, and so it would be useful to implement the context switch such that only the windows currently in use are flushed to memory. It is a matter of fact that the average calling depth during the execution of a program is not very large, and therefore the set of registers that imperatively must be flushed to memory is small.

Taking advantage of the execution points at which it is not necessary to save (and also not necessary to restore) the entire state of the machine [32], ORK adopts the latter approach so as to reduce the excessive overhead of saving and restoring unused window registers.

Not only efficiency, but also the predictability of execution is a crucial concern to ORK. The worst case execution time (WCET) of the two alternative approaches is approximately the same. The adopted implementation however exhibits a better average execution time. This is of no use for timing and scheduling analysis however, which must by definition use only WCET values.

Another issue to take into account is that not all the tasks will use the floating point unit. Thus, the floating point context should not be stored until necessary. It should remain in the floating point registers and not disturbed until another floating point task is switched to. The current implementation saves the floating point context only when necessary.

The same applies for interrupt handlers, the floating point context is saved and restored only if the interrupt handler uses the floating point context.

## 6.8 Kernel.Peripherals

In ORK the set of peripherals which are internally managed are:

- The *General Purpose Timer*.
- The *Real Time Clock*.
- The memory controller.
- The UART

The package Kernel.Peripherals.Registers contains the mappings of the different registers related to peripherals which make them accessible to the kernel.

# Chapter 7

# Software code listings

The source code of ORK is distributed with the GNAT/ORK cross-development system. The latest available version of the compiler can be found in the software repository of the Open Ravenscar project at `http://www.openravenscar.org`.

# Bibliography


[1] ECCS. *ECCS-E-40A Space Engineering — Software*, 1999.

[2] *Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995*, 1995. Available from Springer-Verlag, LNCS no. 1246.

[3] ISO/IEC/JTC1/SC22/WG9. *Guidance for the use of the Ada Programming Language in High Integrity Systems*, 2000. ISO/IEC TR 15942:2000.

[4] Alan Burns. The Ravenscar profile. *Ada Letters*, XIX(4):49–52, 1999.

[5] *ISO/IEC-9899:1990 — Programming Languages — C*, 1990.

[6] IEEE. *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (Incorporating IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995)*, 1990. ISO/IEC 9945-1:1996.

[7] TEMIC. *SPARC V7 Instruction Set Manual*, 1996.

[8] TEMIC. *TSC691E Integer Unit User s Manual for Embedded Real Time 32 bit Computer (ERC32)*, 1996.

[9] TEMIC. *TSC692E Floating Point Unit User s Manual for Embedded Real Time 32 bit Computer (ERC32)*, 1996.

[10] TEMIC. *TSC693E Memory Controller User s Manual for Embedded Real Time 32 bit Computer (ERC32)*, 1996.

[11] Jiri Gaisler. The ERC32 GNU cross-compiler system. Technical report, ESA/ESTEC, 1999. Version 2.0.6.

[12] Christine Ausnit-Hood, Kent A. Johnson, Robert G. Petit IV, and Steven B. Opdahl, editors. *Ada 95 Quality and Style*. Number 1344 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[13] HOOD user Group. *HOOD Reference Manual*, 1993. Version 3.1.

[14] Ada Core Technologies. *GNAT User's Guide. Version 3.13w*, November 1999.

[15] Ada Core Technologies. *GNAT Reference Manual. Version 3.13w*, November 1999.

[16] Richard M. Stallman and Roland H. Pessch. *Debugging with GDB*. Free Software Foundation, 5th edition, 1998. For GDB version 4.17.

[17] E.W. Giering and T.P. Baker. The GNU Ada Runtime Library (GNARL): Design and implementation. In *Proceedings of the Washington Ada Symposium*, 1994.

[18] Dong-Ik Oh and T.P. Baker. The Gnu Ada'95 Tasking Implementation: Real-Time Features and Optimization. *SIGPLAN'97*, June 1997. Workshop on Compiler and Language Support for Real-Time Systems.

[19] T.P. Baker, Dong-Ik Oh, and Seung-Jin Moon. Low-Level Ada tasking support for GNAT - performance and portability problems. In *Proceedings of the Washington Ada Symposium*, July 1996.

[20] T.P. Baker and Offer Pazy. A unified priority-based kernel for Ada. Technical report, ACM SIGAda, Ada Run-Time Environment Working Group, March 1995.

[21] Intermetrics. *Ada 95 Rationale: Language and Standard Libraries.*, 1995. Available from Springer-Verlag, LNCS no. 1247.

[22] Dong-Ik Oh, T.P. Baker, and Seung-Jin Moon. The GNARL implementation of POSIX/Ada signal services. In *Proceedings of the Ada-Europe'96*, 1996.

[23] José F. Ruiz and Jesús M. González-Barahona. Implementing a new low-level tasking support for the GNAT runtime system. In Michael González-Harbour and Juan A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe'99*, number 1622 in LNCS, pages 298–307. Springer-Verlag, 1999.

[24] Juan A. de la Puente, José F. Ruiz, and Jesús M. González-Barahona. Real-time programming with GNAT: Specialised kernels versus POSIX threads. *Ada Letters*, XIX(2):73–77, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[25] Intermetrics, Inc. *Ada 9X Mapping*, August 1991.

[26] H. Shen and T.P. Baker. A Linux kernel module implementation of restricted Ada tasking. *Ada Letters*, XIX(2):96–103, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[27] OAR. *RTEMS SPARC Applications Supplement*, 1997.

[28] Michael Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, June 1997. Available at `http://www.rtlinux.org/\~baraban/thesis`.

[29] Robert Hill, Balaji Srinivasan, Shyam Pather, and Douglas Niehaus. Temporal resolution and real-time extensions to linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998.

[30] Shuichi Oikawa and Ragunathan Rajkumar. Linux/RK: A portable resource kernel in linux. IEEE Real-Time Systems Symposium Work-in-progress Session, December 1998.

[31] Temic/Matra Marconi Space. *SPARC RT Memory Controller (MEC) User's Manual*, April 1997.

[32] J.S. Snyder, D.B. Whalley, and T.P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, February 1995.