The Design and Implementation of the Open Ravenscar Kernel*

Juan A. de la Puente Juan Zamorano José Ruiz Ramón Fernández Rodrigo García

Department of Telematics Engineering
Technical University of Madrid, Spain
E-mail: jpuente@dit.upm.es

Abstract

This paper describes the design and implementation of the Open Ravenscar Kernel (ORK), 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 tasking on an ERC32 (SPARC v7) architecture in an efficient and compact way. It is closely integrated with the GNAT runtime library and other tools.

1. Introduction

The Open Ravenscar Real-Time Kernel (ORK) [10, 11] is a tasking kernel for the Ada language [2] which provides full conformance with the Ravenscar profile [6, 4, 7] on ERC32-based computers. ERC32 is a radiation-hardened implementation of the SPARC V7 architecture, which has been adopted by the European Space Agency (ESA) as the current standard processor for spacecraft on-board computer systems [12].

ORK supports the restricted version of Ada tasking defined by the Ravenscar 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. Debugging support for the ORK kernel, including tasking, is based on an enhanced version of the GDB debugger and the DDD graphic front-end. The distribution includes an adapted version of GNAT hosted on GNU/Linux workstations and targeted to ERC32 bare computers, the kernel itself, adapted version of GDB and DDD, and some additional libraries and tools. It is freely available as an open source product, with a GPL license[1].

*This work has been funded by ESA/ESTEC contract no. No.13863/99/NL/MV.

[1]ORK and its associated software can be downloaded from http://www.openravenscar.com

This paper describes the design and implementation of ORK. The rest of the paper is organised as follows: Section 2 describes how the Ravenscar profile can be implemented in GNAT. Section 3 describes the ORK design and section 4 deals with some implementation issues. Finally, some conclusions and plans for the near future are included in section 5.

2. Support for the Ravenscar profile in GNAT

2.1. Compile-time checking

Most of the Ada subset defined by the Ravenscar profile can be checked at compile time by using an appropriate set of restriction identifiers with the pragma Restrictions (ALRM, D.7, H.4). However, not all the Ravenscar restrictions can be enforced by standard identifiers, and thus a number of additional restriction identifiers have been proposed at the last IRTAW meetings in order to support the profile [7].

The most recent versions of GNAT (from 3.12 on) have included most of the non-standard Ravenscar restrictions as implementation-specific pragmas. However, there are a couple of restrictions that are not implemented in GNAT or are implemented in a slightly different way than specified by the profile:

- The Ravenscar restriction Simple_Barrier_Variables is replaced in GNAT by Boolean_Entry_Barriers. The semantics of this restriction is the same as the original one.
- The Ravenscar restriction Max_Entry_Queue_Depth => N (with $N = 1$ for Ravenscar compliant programs) is replaced in GNAT by No_Entry_Queue. In this case, the semantics is the same, but the restriction name is somewhat misleading, as there may still be one task waiting on an entry barrier to be opened (i.e. a queue with just one task).

Program 1 Sample configuration file for GNAT

```
-- file gnat.adc
pragma Ravenscar;

pragma Restrictions (Max_Tasks => ...);
-- N  must be equal to the number
         -- of application tasks

-- additional restrictions, if any

pragma Task_Dispatching_Policy
  (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
```

The purpose of defining the Ravenscar set of restrictions was to enable a simpler, more efficient runtime to be used. The approach taken in GNAT for this purpose is not to use the Restrictions pragma instead to define a new pragma Ravenscar that enforces the full set of profile restrictions, and selects a reduced version of the GNAT run-time library (GNARL)[2]. However, the Ravenscar restriction Max_Tasks => N cannot be enforced by this pragma and has still to be included in a pragma Restrictions. In addition to this, the pragma Ravenscar enforces two redundant restrictions (No_Terminate_Alternatives and Max_Select_Alternatives => 0, and one restriction which is not required by the Ravenscar profile, No_Dynamic_ Interrupts.

The conclusion is that, although there are some differences between the restrictions imposed by the pragma Ravenscar and the IRTAW definition of the Ravenscar profile, the pragma can be used in its present form, together with other configuration pragmas (see below), to check Ravenscar compliance of Ada programs compiled with GNAT. Program 1 shows a sample configuration file for GNAT that can be used to check that a program is Ravenscar-compliant at compile time (with the exception of the restrictions that no task should terminate and no call to an entry with an already queued entry should be made, which can only be checked at run time).

2.2. The GNU Runtime Library

The GNU Ada Runtime Library (GNARL) [13] implements Ada tasking in a portable way. The GNARL components which are dependent on a particular machine and operating system are known as GNULL (GNU Low-level Library), and their interface to the platform-independent part of the GNARL is called GNULLI (GNULL Interface). Most implementations of GNULL are built on top of an existing set of POSIX thread functions [14], which in turn may be implemented on top of an operating system.

Ada programs compiled with the pragma Ravenscar use a restricted GNARL that takes advantage of the simplified tasking model of the Ravenscar profile to reduce the size and execution time overhead of the runtime [1]. The restricted GNARL, however, is not fully compliant with the Ravenscar profile in that it does not currently support protected interrupt handlers, which are explicitly allowed by the profile.

The ORK distribution includes a patch for the GNAT 3.13 front end, as well as some replacement GNARL packages that have been implemented in order to properly support interrupt handlers in Ravenscar-restricted programs.

3. The design of the Open Ravenscar Kernel

3.1. Overall architecture

The general approach that has been followed in the design of the ORK architecture is to implement tasking with a small, dedicated kernel, which does not have the unnecessary burden of a full pthreads implementation [9]. ORK provides an almost direct implementation of GNULLI, with GNULL packages acting as a thin glue interface layer (figure 1), so that most of the GNULL operations are implemented as simple inlined calls to kernel subprograms.

The kernel itself consists of a set of Ada packages, all of them children of an empty root package called Kernel (figure 2). This structure is similar to that of the JTK [9, 19] and Top-Layer [16] kernels. Some of the packages have additional children that extend their interfaces so that some of their internal functionality is made visible to other kernel packages.

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 hidden threads within the kernel.

3.2. Thread management

Thread management operations are implemented by the package Kernel.Threads. Its main functions include thread creation, thread identification, thread scheduling, and thread synchronization with mutexes and condition variables.

Scheduling is performed according to the FIFO within priorities policy (ALRM D.2.2). Mutexes are locked and unlocked according to the ceiling locking policy (ALRM D.3).

Mutex operations have been simplified with respect to the GNULL definitions. In particular, GNULL defines two different procedures to acquire a mutex (Read_Lock and

[2] Notice that the reduced runtime is not selected if the restrictions are specified individually instead of including pragma Ravenscar.

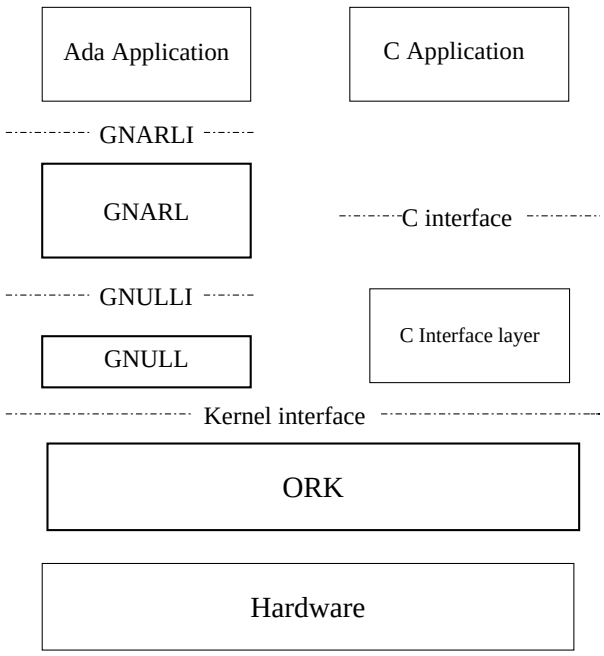

Figure 1. Architecture of GNAT-ORK and main interfaces

Write_Lock), depending on the kind of access required, so that multiple readers are allowed to acquire a mutex concurrently. However, as ORK is designed for a monoprocessor architecture, having different operations for reading and writing locks is an unnecessary overhead. Therefore, ORK provides only one Lock operation, with Write_Lock semantics. The GNULL Read_Lock and RTS_Lock (which is intended to protect GNARL internal data) operations are mapped to the single kernel Lock operation.

The kernel condition variables are used by GNARL only to implement protected object entry call. Since the Ravenscar profile restricts the use of protected entries, a simplification with respect to the POSIX definition of condition variables is possible, so that the maximum number of waiting threads is one, there are no timed-wait operations, and wait operations cannot be cancelled.

3.3. Storage management

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 a Ravenscar-compliant program should not use dynamically allocated memory, according to the recommendations of the Ada HIS standard [15]. However, at least a limited form of dynamic memory allocation is required by GNARL to allocate space for task control blocks and stacks at system initialization. In order to provide this functionality and still keep the advantages of static memory allocation, ORK provides restricted memory allocation with the following characteristics:

- Memory can be allocated only at system initialization time.
- Memory cannot be freed.

In this way, a simple contiguous allocation scheme can be implemented, which can be easily checked for certification purposes.

The kernel provides stack protection for individual task stacks, so that Storage_Error is raised whenever a task tries to move its stack pointer outside the bounds of its stack area.

3.4. Time management

The current GNARL implementation uses condition variable operations to execute all kinds of delays. This scheme allows timed calls to be signalled before the timer expires. 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 cancelling a delay. Therefore, ORK provides a simpler way to read the hardware clock and to share the timers among threads. Threads will wait inside the queue until its 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.

3.5. Interrupt handling

In the current GNARL implementation, interrupt handlers are executed within the context of specially dedicated server tasks, each one associated to an interrupt source. This approach simplifies the scheduling of interrupt handlers, and provides a simple way to achieve mutual exclusion between handlers. However, this implementation uses task rendez-vous, which is forbidden in the Ravenscar profile and cannot thus be used with ORK.

The Ada interrupt handling model [2, 3, 5] implies that a protected handler can only be preempted by a higher priority interrupt. ORK masks all interrupts with a lower priority than the currently active priority, by making the hardware priority equal to the active priority. Moreover, the Ravenscar profile requires the ceiling locking protocol, which means that protected interrupt handlers cannot preempt other operations on the same protected objects. As a result, an interrupt handler can never be blocked waiting for a protected object to be free, and protected handlers can be directly invoked from the ISR. The package Kernel.Interrupts provides operations to install and detach interrupt handlers.

3.6. Other design issues

The kernel interface is a purely procedural one, as there is no need for separate user and supervisor execution modes. All the program runs in supervisor mode, as it is common in embedded systems. Mutual exclusion in the kernel is achieved by means of a monolithic monitor [17] protected by disabling interrupts, so that interrupt delivery is postponed when a kernel function is executed [19, 18].

Mutex lock operations are implemented by simply raising the locking thread active priority to the ceiling priority of the mutex. This is consistent with the implementation proposed by the Ada Rationale [3] for protected objects.

4. Implementation issues

4.1. Language

The kernel is written in Ada 95, except for a small part which is written in SPARC assembly language. A sequential subset of Ada has been defined based on the recommendations of the Ada HIS standard [15]. The Ada features which are not used in the subset are detailed elsewhere [22].

4.2. Thread scheduling and synchronization

The ready thread queue is implemented as a priority-ordered double-linked list. Space for the maximum number of threads that can exist in the system is reserved at initialization, thus avoiding the need for dynamic storage management.

Delayed threads are put on a single queue, ordered by delay expiration time. The queue is implemented as a linear linked list. An "alarm clock" approach is used to signal delay expiration and the subsequent thread activation.

Mutexes are implemented in an efficient way, by raising the priority of the locking thread to the ceiling priority of the mutex.

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

There are two different approaches to follow for the flushing policy. The kernel can flush all register windows, or just the windows currently in use [5]. The latter approach gives better average context switch time [20], and is the one used in ORK. However, the worst case value is approximately the same in both approaches.

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. For the sake of simplicity, the current ORK implementation always saves the floating point context.

The measured context switching time for a 10MHz ERC32 ranges from 83 to 85μs. The interrupt latency is between 285 and 295μs. These figures have been measured on an ERC32 simulator, but we expect them to be close to the real target.

4.4. Time management

The ERC32 hardware provides two timers (apart from the special *Watchdog* timer) which can be programmed to operate on either single-shot or periodical mode [21]. ORK uses one of them (the *Real Time Clock*) as a timestamp counter, and the other (called *General Purpose Timer*) as a high-resolution timer. The first one provides the basis for a high-resolution clock, while the second offers the required support for precise alarm handling. Both timers are clocked by the internal system clock, and they use a two-stage counter (figure 3).

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. For a 10MHz ERC32, the clock granularity is 100ns, and the maximum time interval is about 136 years for a 1s interrupt period.

The *General Purpose Timer Counter* is reprogrammed on demand every time an alarm is set, to signal the time when the alarm expires. This mechanism is used to implement high-resolution (100ns) delays.

5. Conclusions and future work

The Open Ravenscar Kernel supports the full Ravenscar profile with the GNAT compilation system. It has been thoroughfully tested with a Ravenscar profile compatible subset of the ACVC suite, plus a set of additional tests that have been specifically designed to check compliance with the profile. Some problems have been found, partly due to the GNAT implementation, but all of them can be solved with compiler modifications. The validation process and its results are described in the *Software Validation and Verification Report*, available at the Open Ravenscar web site [8].

The kernel has a reduced size (8 KB plus 4 KB for the vector table). However, the minimum size program is about

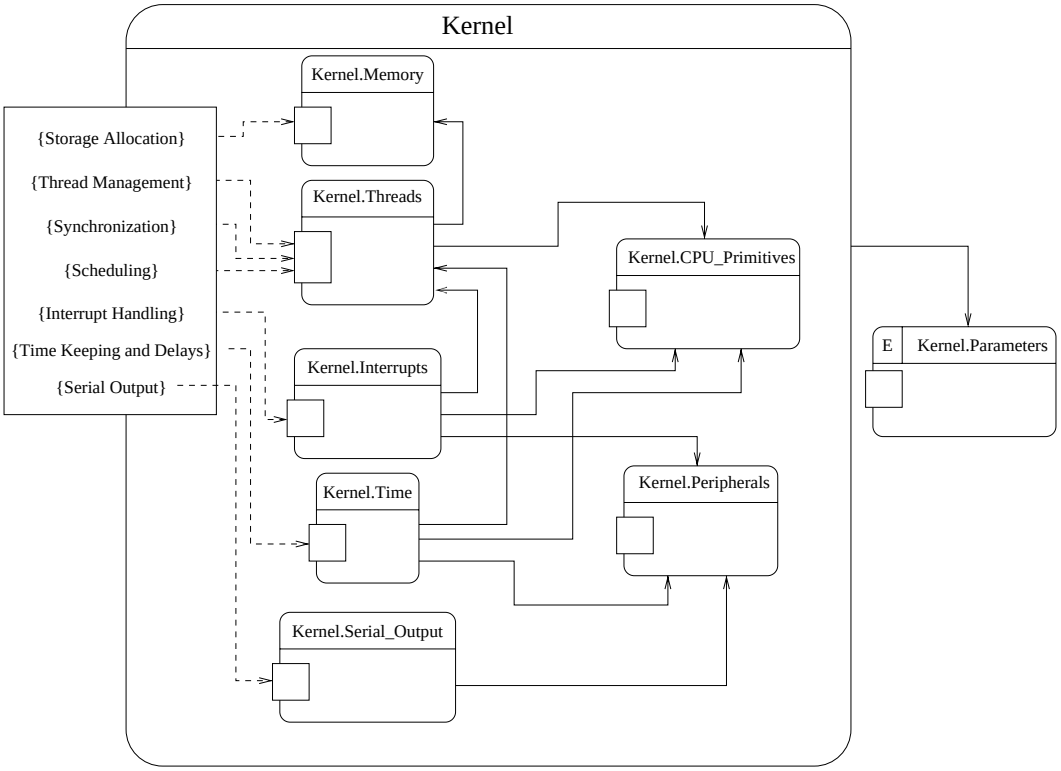

Figure 2. ORK packages.

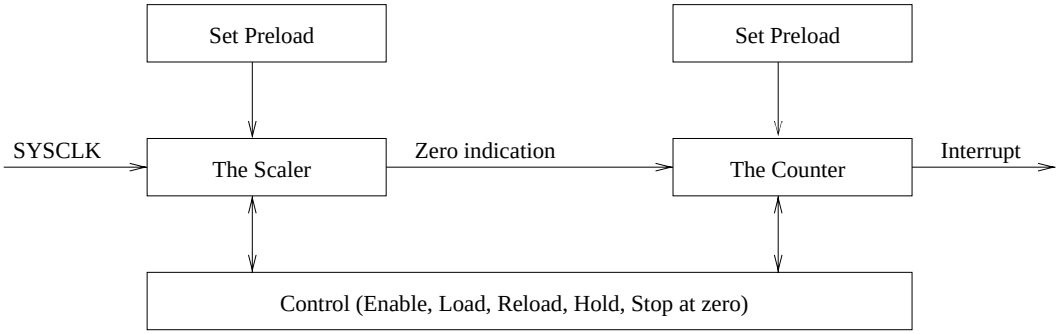

Figure 3. Timer design

95 KB, mainly due to the high amount of code linked into the executable file as a result of references made by GNARL and the GNAT compiler itself. Many of these references are not used by Ravenscar-compliant programs, and thus the code size should be reduced accordingly.

A conclusion is that more effort has to be devoted to adapting GNAT and the upper layers of the run time library to the Ravenscar profile. In spite of the presence of pragma Ravenscar, there are still a number of issues that have to be solved so that GNAT supports the profile in an efficient way. Interrupt handling is one example of such issues, which in this case has been solved by the ORK team.

The kernel has been tested on an ERC32 simulator only. Plans for the near future include validating ORK on real targets, as well as porting it to other platforms. We also plan to perform accurate measurements in order to provide the metrics required by ALRM annex D.

References

[1] Ada Core Technologies. *GNAT Reference Manual. Version 3.13a*, March 2000.

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

[4] L. Asplund, B. Johnson, and K. Lundqvist. Session summary: The Ravenscar profile and implementation issues. *Ada Letters*, XIX(25):12–14, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

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

[6] T. Baker and T. Vardanega. Session summary: Tasking profiles. *Ada Letters*, XVII(5):5–7, 1997. Proceedings of the 8th International Ada Real-Time Workshop.

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

[8] CASA Space Division. *Open Ranvenscar Real-Time Kernel — Software Validation and Verification Report*, July 2000. Available at `http://www.openravenscar.org`.

[9] J. A. de la Puente, J. F. Ruiz, and J. 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.

[10] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An open Ravenscar real-time kernel for GNAT. In H. B. Keller and E. Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[11] J. A. de la Puente, J. F. Ruiz, J. Zamorano, R. García, and R. Fernández-Marina. ORK: An open source real-time kernel for on-board software systems. In *DASIA 2000 - Data Systems in Aerospace*, Montreal, Canada, May 2000.

[12] ESA. *32 Bit Microprocessor and Computer System Development*, 1992. Report 9848/92/NL/FM.

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

[14] 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.

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

[16] M. Kamrad and B. Spinney. An Ada runtime system implementation of the Ravenscar profile for a high speed application layer data switch. In M. González-Harbour and J. A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe'99*, number 1622 in LNCS, pages 26–38. Springer-Verlag, 1999.

[17] A. Mok. The design of real-time programming systems based on process models. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1984.

[18] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

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

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

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

[22] UPM. *Open Ravenscar Kernel — Software Design Document*, May 2000. Revision 1.6.