Open Ravenscar Real-Time Kernel

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

Operation Manual

Version 2.2b — 19 November, 2001

FOR OPENRAVENSCAR 2.2B, WITH METRICS ANNEX

UNIVERSIDAD POLITÉCNICA DE MADRID
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

UNIVERSITY OF YORK
DEPARTMENT OF COMPUTER SCIENCE

CONSTRUCCIONES AERONÁUTICAS, S.A.
DIVISIÓN ESPACIO

Published by

Departamento de Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid
ETSI Telecomunicación, Ciudad Universitaria
E-28040 Madrid
SPAIN

Send questions, comments, suggestions, etc. to `help@openravenscar.org`.

Send bug reports to `bug-report@openravenscar.org`.

Status:	Final
Authors:	Juan A. de la Puente José F. Ruiz Juan Zamorano Jesús González-Barahona Ramón Fernández-Marina Miguel Ánguel Ajo
Revised by:	Ángel Álvarez Alejandro Alonso

History

Version	**Date**	**Comments**
1.1 – 1.5		Internal revisions.
1.6	2000-06-16	First public release.
2.0	2000-09-28	Second public release.
2.2	2001-03-21	Support for remote targets and Solaris host
2.2b	2001-11-19	Appendix F “Metrics” included included.

Contents

Chapter 1

Introduction

1.1 Intended readership

This manual contains operation instructions for the *Open Ravenscar Real-time Kernel* (ORK) for the ERC32 computer and other associate software. It should be read by application programmers and system administrators of software projects using ORK.

Previous knowledge of the Ada 95 [6] and C [7] programming languages is assumed. The reader should also be familiar with the GNAT and GCC programming environments [2], the GDB debugger [34], and the fundamentals of real-time programming and the Ravenscar profile [10, 8, 29].

1.2 Purpose

The purpose of this document is to describe how to use, install, and configure the *Open Ravenscar Real-Time Kernel* (ORK) for ERC32-based computers.

The *Open Ravenscar Real-Time Kernel* 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 both Ada 95 and C applications on an ERC32-based computer.

ORK is fully integrated with GNAT, the GNU-NYU Ada Translator, and with GDB, the GNU debugger. It also works with DDD, a graphic front-end for GDB. The ORK software package includes specialized modules for using these tools with ORK.

1.3 Applicability statement

This manual applies to the following versions of software (openravenscar 2.2b):

- ORK-ERC32 2.2b
- GNAT 3.13
- GCC 2.8.1
- GDB 4.17
- DDD 3.2

1.4 Free software

ORK is *free software*. This means that everyone is free to use it and free to redistribute it on a free basis. This applies to the source code and documentation as well.

ORK is not in the public domain. It is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of ORK that they might get from you. The precise conditions are found in the GNU General Public Licence that comes with ORK. See appendix E for details.

The easiest way to get a copy of ORK is from someone else who has it. The GPL gives you the freedom to copy or adapt a licensed program, but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get to source code), and the freedom to distribute further copies.

You can also get the latest version from the ORK web site,

```
http://www.openravenscar.org/
```

1.5 Related documents

The following documents contain additional information about software tools that can be used to develop ORK-based real-time software:

1. Ada 95 Reference Manual [6].
2. GNU Emacs Manual [24]
3. GNAT User's Guide [5].
4. GNAT Reference Manual [4].
5. Debugging with GDB [34].
6. Debugging with DDD [44].
7. Version Management with CVS [23].

1.6 Conventions

The following typographical conventions are used in this manual:

- Functions, utility program names, standard names, and code listings, are typeset in a `fixed width font`.
- Variables and parameters are typeset in a *`slanted font`*.
- Options are enclosed in square brackets: `[]`
- Commands that are entered by the user are preceded by `$`.

1.7 Problem reporting instructions

If you obtained ORK from a support organization, we recommend you contact that organization first. You can find contact information for support organizations on the ORK web site,

`http://www.openravenscar.org`.

In any event, we also recommend that you send bug reports for ORK to this address:

`bug-report@openravenscar.org`

We welcome bug reports, as they are a vital part of the process of the continuing improvement of ORK. You will help us (and make it more likely that we will look at your report in a timely manner) if you follow these guidelines:

- Please report each bug in a separate message, and add a short but specific subject.
- Please include full sources. We can't duplicate errors without the full sources. Include all sources in the single email message with appropriate indications in the multiple file cases, see below. Also say exactly what you saw, do not assume that we can guess what you saw, or duplicate the behaviour you encountered.

  All sources must be sent in plain ASCII or ISO-8859-1 format.
- Please include a complete identification of the version of the system you are running (i.e. development and target environments, as well as versions of ORK and all other software you are using).
- Please try to reduce your example to a simple one.

If you think that you have found a bug in GNAT, GCC, GDB, or DDD, rather than ORK, please send the bug report to the appropriate address:

- GNAT: `report@gnat.com`
- GCC: `bug-gcc@gnu.org`
- GDB: `bug-gdb@gnu.org`
- DDD: `bug-ddd@gnu.org`

1.8 Glossary

1.8.1 Definitions

Development platform. The computer system (hardware and software) which is used to write, compile, and debug embedded software.

Execution platform. The computer hardware, and possibly basic ROM resident software (such as a bootstrap loader) where the embedded software is executed.

Target platform. The same as execution platform.

RP program. A program that complies with the Ravenscar profile restrictions.

1.8.2 Acronyms

ALRM	Ada Language Reference Manual.
API	Application Program Interface.
ATCB	Ada Task Control Block.
CIL	C Interface Library.
DDD	Data Display Debugger.
ESA	European Space Agency.
ESTEC	European Space Research and Technology Center.
FSF	Free Software Foundation.
GDB	GNU Debugger.
GNAT	GNU New York University Ada Translator.
GNARL	GNU Ada Run-Time Library.
GNARLI	GNU Ada Run-Time Library Interface.
GNORT	GNAT No Run Time.
GNU	GNU is Not Unix.
GNULL	GNU Low-Level Library.
GNULLI	GNU Low-Level Library Interface.
GPL	GNU Public License.
GUI	Graphic User Interface.
HIS	High-Integrity System.
IRTAW	International Real-Time Ada Workshop.
ISO	International Standards Organization.
LGPL	Lesser GNU Public License (formerly Library GPL).
MEC	Memory Controller (a component of the ERC32 chipset).
ORK	Open Real-Time Ravenscar Kernel.
OS	Operating System.
PC	IBM Personal Computer architecture.
PR	Pragma Ravenscar (GNAT specifi c).
PROM	Programmable Read-Only Memory.
RAVENSCAR	Reliable Ada Verifi able Executive Needed for Scheduling Critical Applications in Real-Time.

RP	Ravenscar Profi le.
RP program	An Ada program that complies with the Ravenscar profi le.
SIS	SPARC Instruction set Simulator.
URL	Uniform Resource Locator.

1.9 References

1.9.1 Applicable documents

1. ECCS-E40A. Space Engineering — Software [21].
2. Ada 95 Reference Manual [6].
3. Guidance for the Use of the Ada Programming Language in High Integrity Systems [29].
4. Alan Burns. The Ravenscar profi le [12].
5. C Programming Language [7].
6. POSIX Real-Time Standards [28].

1.9.2 Reference documents

1. ERC-32 Manuals [35, 36, 37, 38].
2. ERC-32 GCC Manual [25].
3. Ada 95 — Quality and Style [9].
4. HOOD 3.1 Reference Manual [27]
5. GNAT Manuals [3, 1].
6. Debugging with GDB [34].

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

1.10 Contributors to ORK

ORK was developed by a team of the Department of Telematics Engineering, Universidad Politécnica de Madrid[1] (DIT/UPM), lead by Juan Antonio de la Puente. The other members of the team were Juan Zamorano, José F. Ruiz, Ramón Fernández, and Rodrigo García. Alejandro Alonso and Ángel Álvarez acted as document and code reviewers, and contributed to the technical discussions with many fruitful comments and suggestions. The same team developed the adapted packages that enable GNAT to work with ORK.

GDB was adapted to ORK by Jesús González-Barahona, Vicente Matellán, Andrés Arias, and Juan Manuel Dodero. José Centeno and Pedro de las Heras acted as reviewers

[1]Technical University of Madrid.

for this part of the work. All of them work at the Department of Engineering, Universidad Rey Juan Carlos, Madrid.[2]

The ORK software was validated by Jesús Borruel and Juan Carlos Morcuende, from Construcciones Aeronáuticas (CASA), Space Division. We also relied very much on Andy Wellings and Alan Burns, of York University, for reviewing and discussions about the Ravenscar profi le and its implementation.

ORK was developed under contract with ESA, the European Space Agency. Jorge Amador, Tullio Vardanega and Jean-Loup Terraillon provided many positive criticism and contributed the user's view during the development. The project was carried out from September, 1999 to June, 2000.

1.11 Document overview

The rest of this document is organised as follows:

- Chapter 2 describes the general structure of the ORK software and the Ravenscar profi le restrictions.
- Chapter 3 contains instructions for writing, compiling, linking, executing, and debugging programs with the ORK software.
- Chapter 4 describes in detail the functions of ORK and the way it is linked with Ada and C programs.
- Appendix A describes the Ravenscar profi le in detail.
- Appendix B contains an analysis of how conformance to the Ravenscar profi le can be hack at compile time with GNAT-ORK.
- Appendix C contains a comprehensive example of a Ravenscar-compliant program and its compilation with GNAT-ORK.
- Appendix E contains a copy of the GNU General Public License (GPL).

[2]King Juan Carlos University.

Chapter 2

Software overview

2.1 The Open Ravenscar Real-Time Kernel

The Open Ravenscar Real-Time Kernel (ORK) is a small, high performance real-time kernel that provides restricted tasking support for Ada 95 programs. The kernel is also usable from C programs.

The kernel is intended to support mission critical real-time software systems. In order to ensure that the software is highly reliable, and even in some cases go through a certifi - cation process, program constructs which are not verifi able should not be used. The exact set of language features to be avoided depends on the degree of integrity that is desired for the software and the verifi cation methods that are to be used. A detailed account of the Ada language issues in high integrity systems can be found in the ISO technical report *Guide for the use of the Ada Programming Language in High Integrity Systems* [29]. Based on these considerations, language subsets for building software with different levels of integrity can be defi ned. In the case of Ada, there is a standard mechanism to enforce that only the required subset of the language is used by means of the pragma Restrictions and the restriction identifi ers that are defi ned in the ALRM "Safety and Security" annex ([6], annex H).

Tasking has often been considered not safe for high integrity systems, mainly due to the diffi culty of analysing and verifying tasking programs. However, recent advances in response time analysis for fi xed priority preemptive scheduling [11] enable limited tasking mechanisms to be used even in this kind of systems.

One of the goals of the 8th International Real-Time Ada Workshop, which was held in 1997 in Ravenscar, Yorkshire, England, was to defi ne a safe tasking model for Ada. The outcome of this work is known as the "Ravenscar profi le" [10]. The profi le was slightly modifi ed in the following meeting [8], after some experience was gained on its implementation and use. It is also inlcuded in the ISO Ada 95 HIS report [29].

The profi le defi nes a subset if Ada tasking that includes static tasks (with no entries) and protected objects (with at most one entry), a real-time clock and delay until statements, as well as protected interrupt handler procedures and other tasking related features. A detailed description can be found in appendix A.

The restrictions in Ada tasking defi ned in the Ravenscar profi le enable tasking to be supported by a small, reliable kernel instead of a full operating systems. ORK is one such kernel, which enables critical real-time systems to be executed on a bare processor with no underlying operating system.

The kernel is integrated with the GNAT compilation system. A special cross-compilation version of GNAT is included in the ORK distribution. Real-time programs are written in a subset of Ada 95 which is consistent with the Ravenscar profi le and with other, non tasking restrictions, as desired according to the degree of integrity that is required for the program. The restrictions can be enforced at compilation time by means of appropriate restriction pragmas.

The code generated by the special version of GNAT can be loaded on the target hardware by means of appropriate bootstrap loaders. It can also be executed on a target simulator for testing purposes. Debugging support is available with an enhanced version of GDB which is also part of the ORK distribution. A graphic debugging interface is provided by means of DDD.

The kernel has been designed for effi cient support of Ada tasking constructs, but can also be used with C programs. A C interface package is provided for this purpose.

2.2 Architecture of ORK

The kernel consists of the following Ada packages:

- Kernel: Root package (empty interface).
- Kernel.Threads: Thread management, including synchronization and scheduling control functions.
- Kernel.Time: Clock and delay services.
- Kernel.Memory: Storage management.
- Kernel.Interrupts: Interrupt handling.
- Kernel.Parameters: Confi guration parameters.
- Kernel.CPU_Primitives: Processor-dependent defi nitions and operations.
- Kernel.Peripherals: Support for peripherals in the target board.
- Kernel.Peripherals.Registers: Defi nitions related to input-output registers of the peripheral devices.
- Kernel.Serial_Output: Support for serial output to a console.

Figure 2.1 shows the structure of the kernel.

The kernel is not intended to be directly used from Ada programs. Instead, an interface to the GNU Ada Runtime Library (GNARL) is used so that Ada 95 tasking constructs can be directly used by the real-time application programmer. This interface is described in the next section.

2.3 Using the kernel with GNAT

Ada tasking is implemented in GNAT by means of the run-time library, called GNARL [26]. The parts of GNARL which are dependent on a particular machine and operating system are known as GNULL, and its interface to the platform-independent part of the GNARL is called GNULLI. GNULL is built on top of ORK, which provides all the low-level tasking support functionality required by the Ravenscar profi le subset of Ada tasking (fi gure 2.2).

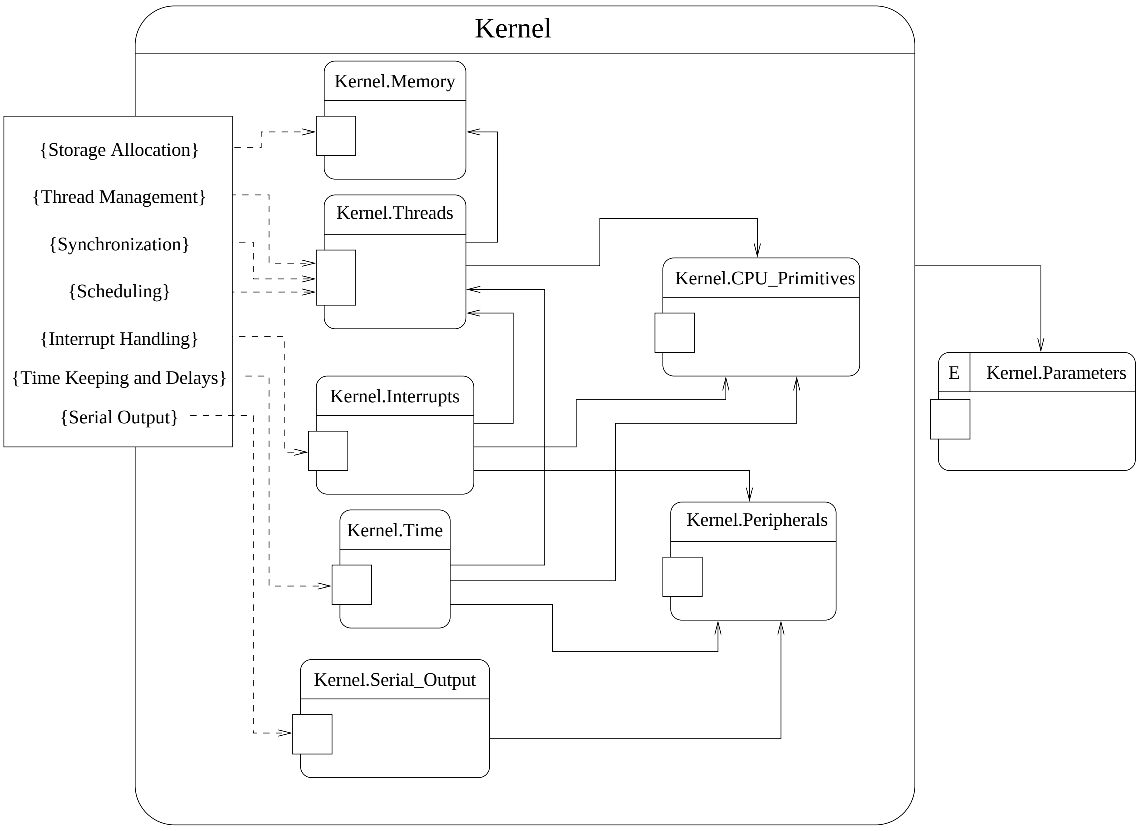

Figure 2.1: Architecture of ORK

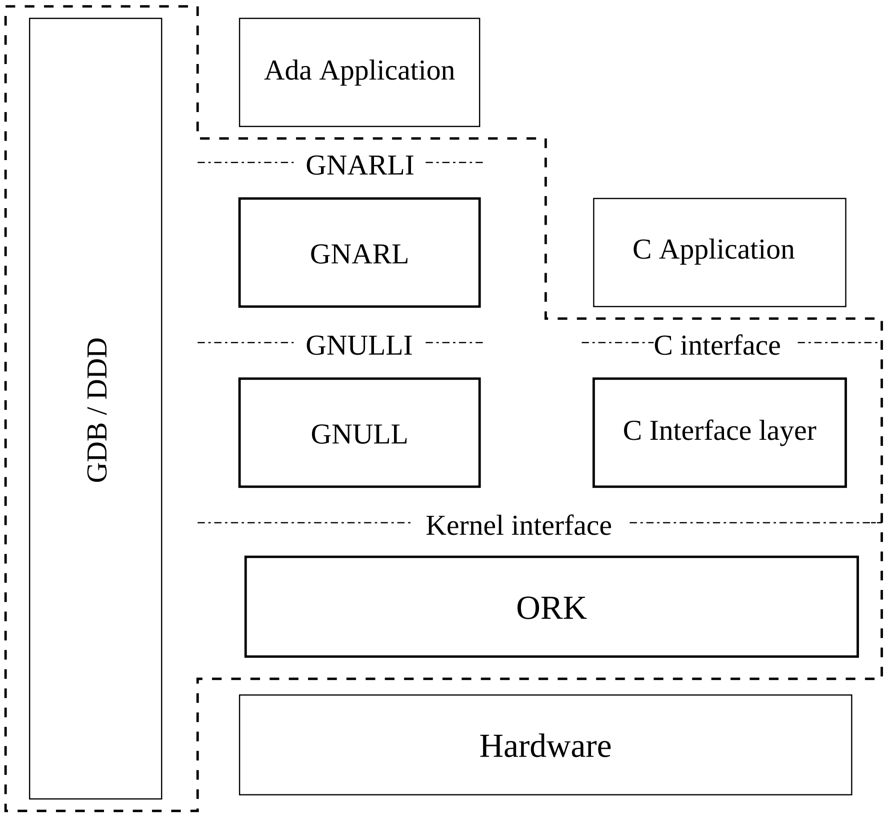

Figure 2.2: Architecture of the GNAT/GCC run-time system based on ORK

Ravenscar-compliant Ada programs are developed in GNAT by enforcing a set of restrictions (see chapter 3 and appendix B) by means of a confi guration pragma (pragma Ravenscar). This pragma also selects a restricted version of GNARL with reduced size and complexity. A special version of GNULL is also used, which interfaces directly with ORK. All this complexity is hidden to the programmer, who only needs to insert the pragma Ravenscar in the GNAT confi guration fi le (commonly named gnat.adc).

2.4 Using the kernel with C programs

The kernel can also be used with programs written in C (see section 3.7). Since C has no tasking constructs, the kernel functions for creating and handling threads have to be explicitly called from C. A C API (see fi gure 2.2) is provided for this purpose. The C interface is provided by a C interface layer (CIL), which is integrated with the GCC compilation system.

2.5 Hardware and software environment

ORK is intended to be used with a GNAT or GCC compilation system targeted to an ERC32 computer. In order to use it effectively, the following components are required.

2.5.1 Development platform

Real-time software based on openravenscar 2.2b can be developed on the following platforms:

- PC-compatible systems with a GNU/Linux operating system. The software has been tested with the Red-Hat 6.2 distribution of GNU/Linux.
- SPARC computers with a Solaris oerating system. The software has been tested with Solaris 2.8.

The cross development system consists of the following packages targeted to ELF-32 ERC32:

- GNAT 3.13: GNU Ada 95 compilation system.
- GCC 2.8.1: GNU C compiler.
- GDB 4.17: GNU debugger.
- DDD 3.2: graphical front-end for GDB.
- Newlib 1.8.2: Cygnus C-library.
- Binutils 2.9.1: GNU binary utilities.
- MKPROM: boot-PROM builder for ERC32 targets.
- REM-COM: remote target monitor for ERC32 targets.
- ORK-ERC32 2.2b: the ORK kernel, the ORK C Interface Library, GNAT patches, and GDB and DDD scripts and patches.

You also may find it useful to have the Emacs editor with Ada mode. You can download all the above mentioned packages from `http://www.openravenscar.org/`. Section 4.1 shows how to install an openravenscar 2.2b compilation system from both the binary and source distributions.

2.5.2 Execution platform

The execution platform for ORK based programs is an ERC32 computer with at least 110 KB memory. Programs can be loaded into an ERC32-based computer memory by means of appropriate tools. Once loaded, the software can be debugged by running GDB on the development computer, which communicates with the target computer by means of a communication line (e.g. serial line or ethernet).

Programs can also be tested and debugged on the development platform usin the TSIM simulator, which can be connected to GDB using a socket connection.[1] Another possibility for debugging and executing ORK-based applications is to use SIS (SPARC Instruction set Simulator).[2]

[1]TSIM is not free software. It is not part of ORK, and it is not available at the ORK site. See `http://www.gaisler.com/` for further details.

[2]SIS is not free software. It is not part of ORK, and it is not available at the ORK site.See `http://www.estec.esa.nl/wsmwww/erc32/freesoft.html` for further details.

Chapter 3

How to use the Open Ravenscar Real-Time Kernel

3.1 Software development

In order to develop programs based on ORK you should perform the following activities:

1. Write the source code for the program.
2. Compile and link the program.
3. Debug the program on the development platform.
4. Debug the program on the target platform.
5. Make a boot PROM for the program.

Two kinds of programs are supported by ORK:

- Ada 95 programs, restricted as defi ned by the Ravenscar profi le.
- C programs directly using kernel threads.

Figure 3.1 describes the data fbw for the GNAT compilation system.

Purely sequential Ada or C programs do not require ORK support, and can be compiled using GNORT[1] or GCC.

3.2 Writing Ada 95 programs

The fi rst step in compiling an Ada application is to write the source code for the program units which make up the application. You can use your favourite text editor for this purpose.

[1]For details about GNORT contact Ada Core Technologies, Inc. at `http://www.act.com` or `http://www.act-europe.fr`.

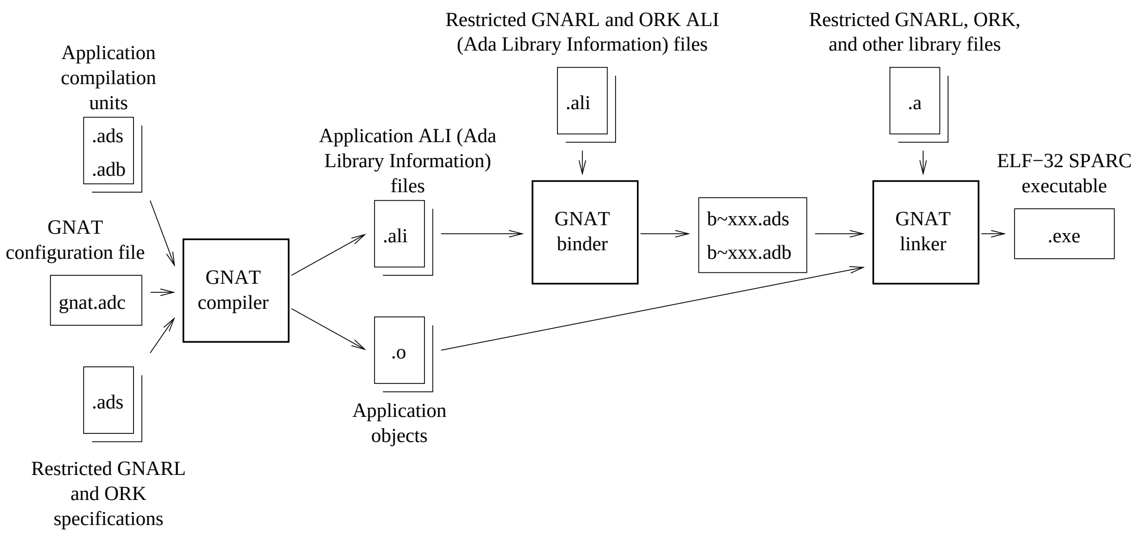
Application compilation units
.ads
.adb
GNAT configuration file
gnat.adc
.ads
Restricted GNARL and ORK specifications
GNAT compiler
Application ALI (Ada Library Information) files
.ali
.o
Application objects
Restricted GNARL and ORK ALI (Ada Library Information) files
.ali
GNAT binder
b~xxx.ads
b~xxx.adb
Restricted GNARL, ORK, and other library files
.a
GNAT linker
ELF−32 SPARC executable
.exe

Figure 3.1: Compilation fbw for GNAT/ORK applications

3.2.1 Ravenscar profile restrictions

When writing your Ada 95 application code, you should bear in mind that only the Ada subset defi ned by the Ravenscar profi le can be used. This means that you should not use any of the following features (see appendix A for a full description of the Ravenscar profi le):

- Task types and object declarations other than at the library level.
- Dynamic allocation and unchecked deallocation of protected and task objects.
- Requeue statement.
- ATC (asynchronous transfer of control via the asynchronous_select statement.)
- Abort statements, including Abort_Task in package Ada.Task_Identification.
- Task entries.
- Dynamic priorities.
- Ada.Calendar package.
- Relative delays.
- Protected types and object declarations other than at the library level.
- Protected types with more than one entry.
- Protected entries with barriers other than a single boolean variable declared within the same protected type.
- Entry calls to a protected entry with a call already queued.
- Asynchronous task control.
- All forms of select statements.
- User-defi ned task attributes.

- Dynamically attached interrupt handlers
- Task termination

If your program has strong integrity requirements, you may also wish to restrict some of the sequential constructs of Ada as well (See the reference [29] for guidelines on the Ada features you may wish to limit.)

ORK assumes that the following restrictions are also applicable to your program:

- No allocators (this means that there are no new statements). This is required as ORK supports only static storage.
- No Ada.Text_IO package. This is required as ORK does not directly support any input-output device but a serial output line, which is not accessible through Ada.Text_IO.

Notice that the above restrictions are common in embedded systems and do not impose any additional limitation on what could be considered as common practice.

WARNING 3.1
Programs that are not Ravenscar profile compliant should not be compiled and linked with the ORK version of GNAT. Using language features which are not allowed by the Ravenscar profile may result in unpredictable compilation or execution errors.

WARNING 3.2
ORK users are recommended to assign distinct priorities to all tasks and protected objects.[2]

3.2.2 The GNAT configuration file

You can have GNAT check all the above restrictions for you by compiling the program with a Ravenscar configuration pragma. Configuration pragmas are put in a special source file called gnat.adc (see [5]). Some additional configuration pragmas should also be included, as shown in the following template:

```
-- gnat.adc - configuration file template for the Ravenscar profile
pragma Ravenscar;
pragma Restrictions (Max_Tasks => N);
-- N must be equal to the number of tasks of the application
-- pragma Restrictions(No_Allocators); - does not work properly in GNAT 3.13
pragma Restrictions(No_IO);
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy   (Ceiling_Locking);
```

A copy of this file is included in the examples directory for your convenience.[3] Of course, you should add any additional restrictions you would like to enforce on your program.

Notice that the maximum number of application tasks is bounded by a kernel configuration parameter (see section 4.4 for the details).

[2]ORK allows priorities to be shared —as long as in keeping withe ceiling priority protocol—but this is not a commendable practice unless the task and protected object population exceeds the allowable range of priorities. See section 4.4 on how to configure the range of priorities and the maximum number of tasks.

[3]The directory `examples` is located directly under the directory where you have installed the openravenscar distribution (`/usr/local/openravenscar` in a standard installation).

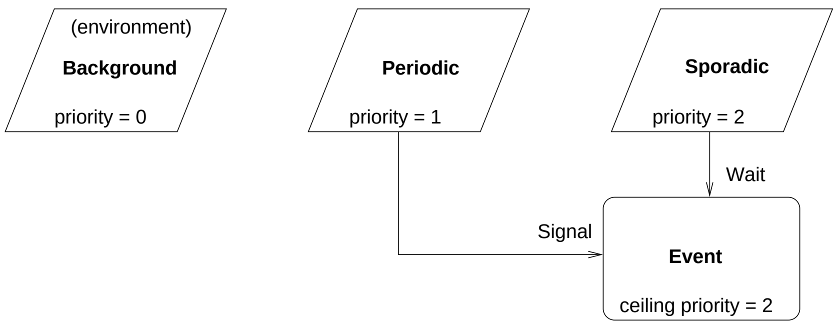

Figure 3.2: Task structure of the Hello program. Parallelograms represent tasks, and rounded rectangles represent protected objects. The arrows denote procedure or entry calls.

3.2.3 A first example

Let us now see a simple Ravenscar-compliant Ada program. The program consists of two compilation units: the main procedure (file hello.adb), and a package with all the application code, including two tasks, a protected object, and a background procedure (files tasking.ads and tasking.adb). Notice that GNAT requires that each compilation unit is in a separate file with the same name as the unit (see the *GNAT User's Guide* [5] for the details).

Figure 3.2 shows the task structure of the program.

```
-- hello.adb - Main procedure for the 'hello' example
with Tasking;
-- used for Background
procedure Hello is
  pragma Priority (0);
begin
  -- do some background work - must not terminate
  Tasking.Background;
end Hello;
```

Notice that the main procedure does nothing but start a background procedure. The Priority pragma specifies the lowest possible priority for the environment task (i.e. the initial task that does all initialization and then calls the main procedure). In this way, we ensure that the background procedure is only executed when there are no other executable tasks.

The environment task is not allowed to terminate in GNAT with the Ravenscar pragma.In order to prevent this to happen, the background procedure must never return. This is checked at compile time by writing a No_Return pragma near the procedure specification (in file tasking.ads):

```
-- tasking.ads - application tasks for the 'hello' example
package Tasking is
  procedure Background;
  pragma No_Return (Background);
  -- background activity - does not terminate
end Tasking;
```

The tasking package specification contains no other declarations. All the application activities are included in the package body:

```
-- tasking.adb - application tasks for the 'hello' example
with Ada.Real_Time;
-- used for Clock, Time_Span, Milliseconds
with Kernel.Serial_Output; use Kernel.Serial_Output;
-- used for Put_Line;
package body Tasking is
  use Ada.Real_Time;

  -- Task and protected object declarations -
  type Cycle_Count is mod 10;

  task Periodic is
    pragma Priority (1);
  end Periodic;

  task Sporadic is
    pragma Priority (2);
  end Sporadic;

  protected Event is
    pragma Priority (2);
    procedure Signal (C : Cycle_Count);
    entry  Wait  (C : out Cycle_Count);
  private
    Occurred : Boolean := False;
    Cycle   : Cycle_Count := 0;
  end Event;

  -- Background procedure

  procedure Background is
    C : Cycle_Count := 0;
  begin
    loop
      C := C + 1;
    end loop;
  end Background;

  -- Task and protected object bodies

  task body Periodic is
    Period : Time_Span := Milliseconds (1_000); -- 1s
    Next  : Time := Clock;
    Cycle : Cycle_Count := 1;
  begin
    loop
      delay until Next;
      Put_Line("Hello periodic");
      if Cycle = 0 then
        Event.Signal(Cycle); -- signal once every 10s
      end if;
      Cycle := Cycle + 1;
      Next := Next + Period;
    end loop;
  end Periodic;

  task body Sporadic is
    Cycle : Cycle_Count;
  begin
    loop
      Event.Wait(Cycle);
      Put_Line("Hello sporadic");
    end loop;
  end Sporadic;

  protected body Event is

    procedure Signal (C: Cycle_Count) is
    begin
      Occurred := True;
      Cycle  := C;
    end Signal;

    entry Wait(C : out Cycle_Count)
      when Occurred is
    begin
      Occurred := False;
      C      := Cycle;
    end Wait;

  end Event;

end Tasking;
```

Notice that the background procedure actually does nothing but increment a count in an endless loop. In real applications it might include some useful work to be executed at the lowest priority.

The tasking package contains two tasks: a periodic task, and a sporadic task. The latter is activated by the periodic task by means of a synchronization protected object (event). This is a common way to implement software-activated sporadic tasks [15]. The periodic task activates the sporadic task once every ten cycles. Each task writes a string to the serial output every time it is activated.

There is a copy of the above files in the `examples/hello` distribution directory. In order to compile and link the example files, you should copy them to a working directory and follow the steps that are described in the next section.

3.3 Compiling and linking Ada 95 programs

You can compile and link your program with gnatmake. For instance:

```
$ sparc-ork-gnatmake hello \
   -largs -k -mcpu=cypress -specs ork_specs
```

The command line switches are described in the *GNAT User's Guide* [5].

You can also compile, bind, and link separately:

```
$ sparc-ork-gcc -c hello.adb
$ sparc-ork-gcc -c tasking.adb
$ sparc-ork-gnatbind -x hello.ali
$ sparc-ork-gnatlink -k -mcpu=cypress -specs ork_specs  hello.ali
```

See the *GNAT User's Guide* [5] for details on the switches and library files.

A link diagnostic information file with the symbols which are mapped by the linker together with information on global common storage allocation can be obtained by using the following switch for gnatlink:

```
$ sparc-ork-gnatmake -g hello.adb -largs -Xlinker -Map hello.map \
  -k -mcpu=cypress -specs ork_specs
```

As a result, a link diagnostic file called hello.map is created. This kind of map files tend to be useful in embedded software development.

After all these compilation steps an ELF-32 SPARC executable called hello is obtained.

3.4 Debugging Ada 95 programs on the development platform

The simplest way you can test your program is by using an ERC32 simulator on your development platform. If you have the TSIM simulator[4] you can do:

```
$ tsim hello
```

And typing "go" from the command prompt you will get the following output:

```
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello periodic
Hello sporadic
Hello periodic
...
```

There is another simulator, called SIS,[5] which is not available for ELF-32 SPARC executable format. Therefore, this simulator requires an additional step to change the format of the executable. You can do:

```
$ sparc-ork-objcopy -O srec hello hello.srec
$ sis hello.srec
```

And typing "go" from the command prompt you will get the same output as with TSIM.

[4]TSIM is not free software, and it is not part of ORK. See `http://www.gaisler.com/` for futher details.

[5]SIS is not free software. It is not part of ORK, and is not available at the ORK site. See `http://www.estec.esa.nl/wsmwww/erc32/freesoft.html` for further details.

WARNING 3.3
SIS uses a 32 bit counter for CPU cycles. As a result, the execution stops after 2^{32} *CPU cycles, which is about 7 minutes of simulated time for a 10 MHz ERC32. You may use the SIS64 version of SIS, which uses a 64 bit counter and has virtually unlimited simulation time. Be aware, however, that it runs about 20% slower than SIS.*

Running the program on TSIM or SIS by themselves does not provide enough information on the behaviour of the program. You can have a better view of the program execution by using the GDB debugger, connected to the TSIM ERC32 simulator. In this case, TSIM must be started with the -gdb option, so that it waits for a connection from GDB:

```
$ tsim -gdb
   ...
gdb interface: using port 1234
```

After that, GDB can be started in the usual way (for instance, in another window). Before loading the program to debug, GDB must be connected to the simulator (using the extended-remote target):

```
$ sparc-ork-gdb hello \
   --command=/usr/local/openravenscar/lib/ork_tasking-tsim.gdb
(gdb) target extended-remote 127.0.0.1:1234
      ...
(gdb) load
      ...
(gdb) cont
      ...
(gdb) detach
      ...
```

You can have a better view with the DDD graphich front-end to GDB:

```
$ ddd hello --debugger sparc-ork-gdb \
       --command /usr/local/openravenscar/lib/ork_tasking-tsim.gdb
```

The GDB script ork_tasking.gdb provides debugging facilities for Ada tasks on top of ORK. See the section on Ada tasking of the manual *Debugging with GDB* to learn more about how to use it, and the facilites actually provided.

WARNING 3.4
This debugging facilities for Ada tasks can be used through the additional buttoms and menus of the improved ORK version of DDD.

If the ORK version of DDD is used then the target can be set and the program can be load in TSIM by pushing the special buttom which is labeled load.

For a complete description of GDB and DDD commands, see the documents *Debugging with GDB* [34] and *Debugging with DDD* [44].

3.5 Executing and debugging Ada 95 programs on the target platform

The remote target monitor needs to be running on the target board to allow remote target debugging with sparc-ork-gdb. The remote target monitor uses UART B as debugging link and UART A as console.

A boot PROM must be make to load and run the remote target monitor on a standalone target.

3.5.1 Making boot PROMS

ORK applications are linked to run from beginning of RAM at address 0x2000000. To make a boot-PROM that will run on a standalone target, use the mkprom utility. This will create a compressed boot image that will load the application to the beginning of RAM, initiate various MEC register, and finally start the application. mkprom will set all target dependent parameters, such as memory sizes, number of memory banks, waitstates, baudrate, and system clock. The applications do not set these parameters themselves, and thus do not need to be relinked for different board architectures. The example below creates a boot PROM for a system with 1 Mbyte RAM, one waitstate during write, 3 waitstates for ROM access, and a 10 MHz system clock. For more details see the mkprom manual.

```
$ mkprom -ramsz 1024 -ramwws 1 -romws 3 hello -freq 10 hello.srec
```

A file called `hello.srec` is created. The format of that file is Motorola S-record which is usually accepted by PROM recorders. It is possible to use an ERC32 simulator (SIS or TSIM) to load and test the resulting file.

3.5.2 Remote target monitor

The directory `/usr/local/openravenscar/src/rem-com` contains the remote monitor. The monitor supports “break-in” into a running program by pressing Ctrl-C in GDB or interrupt in DDD. The two timers are stopped during monitor operation to preserve the notion of time for the application. Note that the remote debugger monitor only works with programs compiled with ORK-ERC32 2.2b.

Type make, in the `/usr/local/openravenscar/src/rem-com` directory, to build the monitor. Before building setup the `Makefile` for your board. Record the resulting `*.srec` file to your boot-PROM. The remote debugger must be attached via UART B to the host, and a console can be attached via UART A.

The monitor installs itself into the top 64K of RAM. It therefore needs to know how large the RAM is. The default RAM size for the monitor is 2 Mbyte. You will have to adjust the Makefile in the `/usr/local/openravenscar/src/rem-com` directory if your system has a different memory size.

3.5.3 Using GDB with a remote target

You can use the debugger while your program runs on the target platform. To do so you need a serial link between UART B of the remote target and a serial device on your host station. If your program produces some output, you need another serial link connection from UART A to a terminal emulator on your host to display it. Terminal emulators such as tip, minicom and kermit will do.

For example, on a i386/GNU Linux workstation, UART A can be connected to /dev/ttyS0 (usually labeled as COM1), and UART B can be connected to /dev/ttyS1 (usually labeled as COM2). On a SPARC/Solaris workstation /dev/ttya and /dev/ttyb can be used.

The remote target has a boot PROM with a remote target monitor which operates the serial lines at 115200 bauds.

If under this example confi guration you type

```
$ xterm -e tip -115200 /dev/ttyS0 &
```

An X-terminal is created, and the terminal emulator program TIP is attached to /dev/ttyS0, which is confi gured at 115200 bauds. The initial message of the remote target monitor will then be displayed on the X-terminal by powering-on or resetting the remote target. Now, the debugging session can be started:

```
$ sparc-ork-gdb hello \
   --command /usr/local/openravenscar/lib/ork_tasking-rem-tar.gdb
(gdb) set remotebaud 115200
(gdb) target erc32 /dev/ttyS1
(gdb) load
(gdb) continue
      ...
```

You can also use DDD for a better interface:

```
$ ddd hello --debugger sparc-ork-gdb \
   --command /usr/local/openravenscar/lib/ork_tasking-rem-tar.gdb
```

If you use the ORK version of DDD, then you may use the "Load" menu button to set the target and load the program on the remote target. The default GBD script ork_tasking-rem-tar supplied with the ORK distribution uses /dev/ttyS1 at 115200 bauds for the host-target communication, but you can change this setting by editing the script.

3.5.4 Using GBD with an ERC32 simulator with the remote target monitor

It is possible to use a simulator as the remote target, and connect it with sparc-ork-gdb.

For example, let us suppose that UART B is connected to pseudo-device /dev/ttypb. The you can type:

```
$sis -freq 10 -fast_uart -uart2 /dev/ptypb mon.srec
> go
```

SIS [6] simulates a remote target with a 10 MHz ERC32, UART B connected to pseudo-device /dev/ttypb, and running the remote target monitor. The initial message of the remote target monitor will be displayed.

The fast_uart flag, which can be omitted, runs UARTS at an infi nite speed, rather than with the correct slow timing. As a result the communication via the pseudo-serial lines will be as fast as possible.

Now, the debugging session can be started:

[6] Either SIS or TSIM can be used for this purpose.

```
$ sparc-ork-gdb hello \
   --command /usr/local/openravenscar/lib/ork_tasking-rem-sim.gdb
(gdb) target erc32 /dev/ttypb
(gdb) load
(gdb) continue
      ...
```

Of course, DDD can also be used:

```
$ ddd hello --debugger sparc-ork-gdb \
   --command /usr/local/openravenscar/lib/ork_tasking-rem-sim.gdb
```

If you use the ORK version of DDD, then you may use the “Load” menu button to set the target and load the program on the remote target. The default GBD script ork_tasking-rem-sim, which is supplied with the ORK distribution, uses `/dev/ttypb` for the host-simulator communication, but you can change this setting by editing the script.

It is also possible to connect UART A to other pseudo-device (/dev/ttypa) and to attach to the pseudo-device a terminal emulator program. Then, the initial message of the remote target monitor will be displayed on the terminal emulator program.

WARNING 3.5
Occasionally, sparc-ork-gdb hangs when attempting to load the program on the simulators (SIS or TSIM). If you experience this problem, type Crtl-C and then type load again. This problem has never arisen with actual remote targets. Therefore, it can be concluded that it is a simulator or a Unix pseudo-device bug. The bug has not been fi xed by the ORK team as the source code of the simulators is not available.

3.6 Interrupt handlers

3.6.1 Protected procedure handlers

The Ravenscar profi le allows the use of protected procedures as interrupt handlers. Interrupt handlers are declared as parameterless protected procedures, attached to an interrupt source. Interrupt sources are identifi ed in the Ada.Interrupts.Names package. This package contains the identifi ers of all the ERC-32 interrupts.

WARNING 3.6
The Ravenscar profi le only allows the static attachment of protected procedures as interrupt handlers. The sources of GNAT Run-Time Library have been modifi ed to take into account that restriction by the ORK team. As a result, a pragma Interrupt_Handler without the corresponding pragma Attach_Handler gives the following compilation error:

```
fatal error: run-time library configuration error
cannot locate "Dynamic_Interrupt_Protection" in file "s-interr.ads"
(entity not in package)
```

Moreover, a call to Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference will raise Program_Error. Although, a call to Is_Reserved does not raise Program_Error in the current version of ORK.

A general template is:

```
with Ada.Interrupt.Names; use Ada.Interrupt.Names;
-- used for External_Interrupt_0, External_Interrupt_0_Priority

protected Interrupt is

  -- public protected operations

private
  -- the handler need not be visible outside the protected object
  pragma Interrupt_Priority(External_Interrupt_0_Priority);
  procedure Handler;
  pragma Attach_Handler (Handler, External_Interrupt_0);
  -- other private operations and data
end Interrupt;
```

Notice that you should assign a priority to the protected object with a pragma Priority. You should use a priority level equal to the hardware priority of the interrupt source. Notice that hardware priority levels are the values of interrupt names as declared in the package Ada.Interrupt.Names.

WARNING 3.7
You should only use priorities in the Interrupt_Priority range for protected objects that contain interrupt handlers (ALRM C.3.1).

3.6.2 An example with interrupts

Appendix C describes an example application with interrupt handlers.

3.7 Working with C programs

3.7.1 The C interface

ORK may be used from C programs through the CIL (C Interface Layer, figure 2.2). The CIL consists of some header files that contain the appropriate types and function definitions which are needed to interface with the ORK kernel. These files are:

- ork.h
- types.h
- kernel-interrupts.h
- kernel-memory.h
- kernel-threads.h
- kernel-time.h

3.7.2 ORK & CIL limitations

A C program for the ORK kernel can be divided into any number of .c and .h files, just like any ordinary C program. However, the following issues should be taken into account:

- The Ravenscar profile restrictions cannot be enforced at compilation time by any means, so the conformance of the application with the Ravenscar profile is left up to the programmer.
- Concurrency is handled by the kernel. No processes can be created, only threads, and only through the CIL capabilities. Moreover, the main function must not terminate, as the Ravenscar profile states clearly.

As a result, the main program will have the following structure:

```
void main (void)
{
     /* C variable declaration */
     /* Ada packages elaboration */
     /* ORK kernel initialization */
     /* C variable initialization */
     /* Tasks creation statements */
     /* Environment Task's infinite loop */
}
```

Tasks must not terminate either, so the structure of task bodies will be as follows:

```
void task_body (void)
{
     /* Task's local variable declaration and initialization */
     while (1) {
         /* task code */
     }
}
```

- Ada packages must be elaborated before functions exported to C can be used. This means that the function adainit() must be called as soon as possible from the C program. Although the program must not terminate, a call to adafinal() at the end would be desirable, just in case.
- No dynamic memory can be used, thus no malloc calls can be made. All variables must be declared statically, and references must be used instead of pointers.

3.7.3 A simple example

The example that follows is a C implementation of the Ada program from section 3.2.3, with a condition variable being used instead of a protected object. The output will be the same as the one on section 3.4

```
###########################################################
# $Id: c-hello.c,v 1.6 2002/03/01 11:52:58 ork Exp $
###########################################################
#include "ork.h"

ork_cond_t event_c;
ork_mutex_t event_m;
```

```
int periodic (void) {

        ork_time_t period = 1*SECOND;
        ork_time_t next = kernel_clock();
        int c = 10;

        while (1) {
                kernel_delay_until(next);
                c−−;

                printf("Hello periodic\n");
                if (c == 0) {
                        kernel_condition_signal (&event_c);
                        c = 10;
                }
                next = next + period;
        }
}

int sporadic (void) {

        int ceiling_violation = 0;

        while (1) {
                kernel_mutex_lock (&event_m, &ceiling_violation);
                kernel_condition_wait (&event_c, &event_m);
                kernel_mutex_unlock (&event_m);

                printf("Hello sporadic\n");
        }
}

int main (void)
{
        ork_thread_t periodic_task;
        ork_thread_t sporadic_task;
        ork_thread_t env_task;

        ork_prio_t p_periodic;
        ork_prio_t p_sporadic;
        ork_prio_t p_env;

        ork_prio_t p_mutex;

        int stacksize = 5120;
        int ok;
        int count = 0;

        adainit();
        kernel_initialize();

        p_periodic = 200;
        p_sporadic = 201;
        p_mutex = 202;

        kernel_mutex_init (&event_m, p_mutex);
        kernel_condition_init (&event_c);
        // How to change environment thread's priority
        env_task = *kernel_thread_self ();
        p_env = 100; // New priority
        kernel_set_priority (&env_task, p_env);

        kernel_thread_create(&periodic_task, periodic, 0, p_periodic, stacksize, &ok);
        kernel_thread_create(&sporadic_task, sporadic, 0, p_sporadic, stacksize, &ok);

        // Environment task's code
        while (1) {
                if (count++ >= 10)
                        count = 0;
        }

        adafinal();
}
```

As it follows from the example, a single header file must be included: ork.h, which includes the other CIL header files required by the program.

Notice that the main procedure never terminates. This is due to the GNAT requirement that the environment task never ends under the Ravenscar profile (the same happens with Ada example in 3.4).

The default priority for the environment task is 120. This can be changed by the environment task itself, as can be seen in the example (line 65), where the environment task's priority is set to 100.

3.7.4 Compiling a C program

The compiler must know which Ada packages have to be elaborated. As such information cannot be provided in the C program nor in the Makefile, a dummy Ada procedure that uses the ORK packages has to be added to the program. The code of this procedure is:

```
with Kernel.Threads;
with Kernel.Time;
with Kernel.Interrupts;
with Kernel.Memory;
procedure Stub is
begin
  null;
end Stub;
```

The steps that should be followed to produce an ORK-based application written in C are:

1. Compile your C code:

   ```
   $ sparc-ork-gcc -I/usr/local/openravenscar/include -g -c \
      c-program.c
   ```

2. Compile the Ada code:

   ```
   $ sparc-ork-gnatmake -gnata -gnatE -gnato -gnatr -g -c stub.adb
   ```

3. Bind the Ada program:

   ```
   $ sparc-ork-gnatbind -n -t -Mmain stub.ali
   ```

4. Link everything together:

   ```
   $ sparc-ork-gnatlink -k -mcpu=cypress -specs ork_specs \
      stub.ali c-program.o -lgnarl -o c-application
   ```

Chapter 4

ORK reference

4.1 Installation and directory structure

4.1.1 Getting ORK

ORK is only distributed via anonymous ftp. The primary home of ORK is `http://www.openravenscar.org`. The ORK cross-compilation system and related tools can be found by clicking on the link *"software"*. The sources used to build ORK cross-compilation system can be also found at the same location. The ORK distribution includes:

openravenscar-2.2-i686-pc-linux-gnu-bin.tar.gz : gzipped tarfile which contains the binary distribution for GNU/Linux. The current distribution has been compiled on Red-Hat 6.2using glibc2 libraries. To avoid problems with different versions of libc, all binaries are statically linked.

openravenscar-2.2-sparc-solaris2.8-bin.tar.gz : gzipped tarfile which contains the binary distribution for Solaris. The current distribution has been compiled on Solaris 2.8. To avoid problems with different versions of libc, all binaries are statically linked.

openravenscar-2.2-src.tar.gz : gzipped tarfile which contains the sources as well as the procedures for building the ORK.

PC-compatible computers with a GNU/Linux operating system and SPARC computers with Solaris are supported as development platforms. The installation procedure for a Linux computers is detailed but the installation procedure for a Solaris computers is the same. Although, the binary distribution for Solaris must be used instead.

4.1.2 Installing ORK

The openravenscar 2.2b directory tree has been compiled to reside in the `/usr/local/openravenscar` directory. After obtaining the gzipped tarfile `openravenscar-2.2-i686-pc-linux-gnu` `tar.gz`, which includes the binary distribution, uncompress and untar it onto `/usr/local/openravenscar`, or create a link to the `openravenscar` location.

The Open Ravenscar distribution can be installed with the following commands (assuming the gzipped tar file is located at `/tmp/openravenscar-2.2-i686-pc-linux-gnu-bin.tar.gz`):

```
$ cd /usr/local
$ tar -zxvf /tmp/openravenscar-2.2--i686-pc-linux-gnu-bin.tar.gz
```

After the cross-compilation system is installed, `/usr/local/openravenscar/bin` must be added to the search path (usually, environment variable PATH in your shell).

4.1.3 Installing the ORK sources

In this chapter it is assumed that the sources are installed in `/usr/local/openravenscar/src`, although they can be installed at any location. After obtaining the gzipped tarfile `openravenscar-2.2-src.tar.gz` which contains the sources of ORK cross-compilation system, uncompress and untar it to `/usr/local/openravenscar/src`.

The Open Ravenscar distribution can be installed with the following commands (assuming the gzipped tar file is located at `/tmp/openravenscar-2.2-src.tar.gz`):

```
$ cd /usr/local
$ tar -zxvf /tmp/openravenscar-2.2-src.tar.gz
```

The sources have been adapted using ACT patches, RTEMS patches, and specific ORK patches. The sources provided are ready for building the ORK cross-compilation system. The directory `/usr/local/openravenscar/src` also contains procedures for building the whole ORK cross-compilation system and the ORK adalib (see sections 4.4 and 4.4.3).

4.1.4 Directory structure

Contents of /usr/local/openravenscar

bin: executables.

demo: demo application which shows ORK functionality (see appendix C).

include: include files.

lib: gcc libraries which include ORK adalib for erc32 target.

info: gcc documentation in info format.

man: man pages.

sparc-ork-elf: newlib (libc) library for SPARC family.

src: sources of ORK and related tools.

Contents of /usr/local/openravenscar/src

binutils-2.9.1: Adapted sources of binutils for ORK.

newlib-1.8.2: Adapted sources of newlib for ORK.

gcc-2.8.1: Adapted sources of gcc for ORK.

gcc-2.8.1/ada: Adapted sources of gnat-3.13a for ORK including ORK itself.

ORK: Startup code and miscellaneous support routines.

4.1.5 Tools

ORK includes the following tools in the `/usr/local/openravenscar/bin` directory:

sparc-ork-addr2line: utility to translate program addresses into fi le names and line numbers.

sparc-ork-ar: library archiver.

sparc-ork-as: cross-assembler.

sparc-ork-c++filt: utility to demangle C++ symbols.

sparc-ork-gasp: assembler pre-processor.

sparc-ork-gcc: C cross-compiler.

sparc-ork-gnat: utility to list GNAT commands, qualifi ers and options.

sparc-ork-gnatbind: Ada binder.

sparc-ork-gnatbl: Ada bind and link.

sparc-ork-gnatchop: Ada source code splitter.

sparc-ork-gnatfind: Ada utility for locating defi nitions and/or references to a specifi ed entity or entities.

sparc-ork-gnatkr: Ada fi le name kruncher.

sparc-ork-gnatlink: Ada linker.

sparc-ork-gnatls: Ada library lister.

sparc-ork-gnatmake: Ada make utility.

sparc-ork-gnatmem: Ada utility to monitors dynamic allocation and deallocation activity in a program.

sparc-ork-gnatprep: Ada pre-processor.

sparc-ork-gnatpsta: utility to print the Standard package.

sparc-ork-gnatpsys: utility to display the System package.

sparc-ork-gnatxref: Ada utility to generating a full report of all cross-references.

sparc-ork-ld: linker.

sparc-ork-nm: utility to print symbol table.

sparc-ork-objcopy: utility to convert between binary formats.

sparc-ork-objdump: utility to dump various parts of executables.

sparc-ork-ranlib: library sorter.

sparc-ork-size: utility to display segment sizes.

sparc-ork-strings: utility to dump strings from executables.

sparc-ork-strip: utility to remove symbol table.

4.1.6 Documentation

An extensive set of documentation for all the tools can be found in the `/usr/local/openravenscar/info` and `/usr/local/openravenscar/man` directories.

Data sheets for the ERC32 as well as GNU tools such as ld, as, bdf, etc. can be found at the ESA ERC32CCS site located at `ftp://ftp.es-tec.esa.nl/pub/ws/wsd/erc32/erc32ccs`.

4.2 Kernel interface

4.2.1 Introduction

The kernel provides all the required functionality to support real-time programming on top of the ERC32 hardware architecture. The kernel functions are 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.
4. Interrupt handling.

All these functions are described in the following subsections.

The kernel is normally used as a low-level layer providing the basic functionality to the upper GNAT run-time system. However, it can be used directly from an application program, written in either Ada or C.

4.2.2 Threads and synchronization

The operations related with the initialization of the kernel, thread management, synchronization, and scheduling are implemented in the package Kernel.Threads.

```
with System;
-- used for Address
package Kernel.Threads is

  procedure Initialize;

  -- Thread support

  type Thread_Id is private;
  Null_Thread_Id : constant Thread_Id;

  procedure Thread_Create (Id : in out Thread_Id;
                           Code       :      System.Address;
                           Args       :      System.Address;
                           Priority :        Integer;
                           Stack_Size :      Integer;
                           Succeeded :   out Boolean);

  function Thread_Self return Thread_Id;
```

```
   -- Mutual exclusion locks

   type Mutex is limited private;

   procedure Mutex_Init (Id        : access Mutex;
                         Ceiling_Priority :   Integer);
   procedure Mutex_Lock (Id        : access Mutex;
                        Ceiling_Violation : out Boolean);
   procedure Mutex_Unlock (Id      : access Mutex);

   -- Condition variables

   type Condition is limited private;

   procedure Condition_Init (Id : access Condition);
   procedure Condition_Wait (Id : access Condition;
                             Mutex_Id : access Mutex);
   procedure Condition_Signal (Id : access Condition);

   -- Scheduling -

   procedure Set_Priority (Id : Thread_Id;
                           Priority : Integer);
   function Get_Priority (Id  : Thread_Id) return Integer;

   procedure Yield;

   -- Debugging support -

   function Check_No_Mutexes (Self_ID : Thread_Id) return Boolean;

end Kernel.Threads;
```

Before calling any kernel operation, the initialization routine (Initialize) must be explicitly invoked. Its purpose is to initialize the ready queue, as well as the descriptors of the environment thread (which executes the main procedure) and the dummy thread (which is executed when there is no ready threads in the system).

Once the kernel has been initialized, threads can be created invoking the procedure Thread_Create. This procedure needs to know the pointer to the function to execute (and its argument), its priority, and its required stack size. With this information a new thread is created. Both the thread descriptor and the new stack for the thread are obtained from a preallocated pool, so that no dynamic memory allocation is needed. The function Thread_Self is used to obtain the identity of the currently running thread.

The synchronization of threads can be achieved using both mutexes and condition variables. Mutexes implement the *ceiling locking* protocol, as required by the Ravenscar profi le. Condition variables must always be used with an associated mutex to guarantee the mutual exclusion.

The procedure Mutex_Init initializes the mutex, setting the value of its ceiling priority. Once the mutex is initialized, in can be acquired calling Mutex_Lock. The effect of seizing the mutex is that the active priority of the thread is raised to the ceiling priority of the mutex.

Several implementations of threads support acquiring a mutex just for reading or with read/write permission. This choice enhances the level of concurrency in multiprocessor systems, but it presents an unnecessary overhead with no benefi ts in monoprocessor systems.

The procedure Mutex_Unlock releases the mutex. The active priority of the thread is restored to the value held by the thread just before acquiring the mutex.

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

Seizing of mutexes can be nested, so that a thread which holds a mutex can acquire another one; of course, the ceiling priority of the latter must be greater or equal to the ceiling of the former, to avoid the violation of the *ceiling locking* protocol. When nesting mutexes, LIFO order of unlocking is required.

Condition variables are used to suspend the current thread until the condition is signalled. Conditions have a procedure called Condition_Init which initializes the condition. When a thread wants to suspend itself until the condition is signalled by another thread, Condition_Wait is called. Waiting on a condition is always associated to a mutex, which must be held by the thread before calling the Condition_Wait operation. The effect of this call is to atomically release the mutex and suspend the thread. The kernel guarantees that the thread is granted the mutex again when it resumes execution.

The Ravenscar profile does not allow more than one thread to wait on a condition at the same time. If a thread tries to wait on a condition that already has another thread waiting on, a bounded error occurs. The default kernel action upon this bounded error is to suspend the calling thread forever. In fact, this action should never be taken for Ada tasks, because this error is caught by the GNARL code that deals with protected entry call. The GNARL action for this bounded error (violation of the Ravenscar profile restriction Max_Queue_Depth => 1) is to raise Program_Error, as specified by the Ravenscar profile definition (see appendix A).

WARNING 4.1
The GNAT 3.13 implementation of this specification does not work correctly. It is therefore possible that the kernel suspends a task forever as a result of calling an entry with a queued call in the current openravenscar 2.2b cross-compilation system.

The procedure Condition_Signal signals the condition, so that if there is a thread waiting on the condition, the task resumes its execution. If there are no waiting threads, this call has no effect.

The base priority of the thread, which is the priority of the thread without taking into account the dynamic priority changes which may be caused by the *ceiling locking* policy, can be changed by calling the procedure Set_Priority. The current base priority of a thread can be obtained calling the procedure Get_Priority.

The scheduling of threads is performed according to the *FIFO within priorities* and the *ceiling locking* methods (see ALRM D.2-3). However, a context switch may be forced by calling the procedure Yield; when calling this procedure, the ownership of the processor is voluntarily transferred to the next ready thread with the currently active priority. If there is no other thread with this active priority this call has no effect.

4.2.3 Time management

The operations related with time are implemented in package Kernel.Time.

```
package Kernel.Time is

   type Time is range −(2**63) . . +(2**63 − 1);

   −− Clock
   function Clock return Time;
   Tick : constant Time;

   -- Delay until
   procedure Delay_Until (T : Time);

end Kernel.Time;
```

The time in ORK is represented 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 kernel provides 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.

The ERC32 hardware provides two timers (apart from the special *Watchdog* timer) which can be programmed to be either of single-shot type or periodical type [39]. 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. Therefore, the first provides the basis for a high resolution clock, while the second offers the required support for precise alarm handling.

The *Real Time Clock* is programmed by ORK to interrupt periodically by updating the most significant part of the clock. The less significant part of the clock is held in the hardware clock register. A software register is used to store the most significant part of the clock. 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 interrupt period.

Depending on the selected period for the clock interrupt, the overhead imposed to the system changes.

The current value of the real-time clock can be obtained calling the function Clock. This function returns the number of nanoseconds elapsed since system startup, providing a time zone independent, monotonically increasing, absolute time value.

The granularity of the real-time clock can be read from the constant called Tick.

When a thread needs to suspend until an absolute time, the procedure Delay_Until is called. The effect of this call is the suspension of the calling thread until the value of the clock is equal to or greater than the specified time. If the alarm time is not in the future, the ownership of the processor is transferred to the next ready thread with the currently active priority.

4.2.4 Memory management

The memory management is implemented by package Kernel.Memory. No heap management is provided, as dynamic data allocators are expected not to be found in Ravenscar compliant programs.

```
with System;
-- used for Address
with System.Parameters;
-- used for Size_Type
package Kernel.Memory is

   function New_Memory_Region (Bytes : System.Parameters.Size_Type)
     return System.Address;
   function New_Stack   (Bytes : System.Parameters.Size_Type)
     return System.Address;
```

```
end Kernel.Memory;
```

The procedure New_Memory_Region allocates a memory block with the desired size, returning the address of the beginning of this block.

Space for stacks is requested using the function New_Stack. The required stack size is set as the argument of the function, and the value returned by the function is the address of the top of the new stack.

Memory deallocation is not supported by ORK.

4.2.5 Interrupt handling

Interrupt names and operations are encapsulated by package Kernel.Interrupts.

```
with System;
-- used for Address
package Kernel.Interrupts is

  type Interrupt_ID is ...;

  procedure Attach_Handler (Handler : System.Address;
                            Id      : Interrupt_ID);
  procedure Detach_Handler (Id : Interrupt_ID);
  function Current_Handler(Id : Interrupt_ID)
   return System.Address;

end Kernel.Interrupts;
```

Interrupt handlers are always executed using an interrupt stack. The size of the interrupt stack can be modifi ed by the user changing the value of Kernel.Parameters.Interrupt_ Stack_Size. Interrupt handlers are called directly from the hardware, and are executed as if they were directly invoked by the interrupted thread (but using the interrupt stack).

The procedure Attach_Handler must be called to attach a handler to an interrupt. The required arguments for this procedure are:

- Id. The interrupt identifi er.
- Handler. The address of the procedure used as interrupt handler.
- Priority. The priority at which the interrupt handler is executed. This value must be at least equal to the priority of the interrupt.

If the active priority of a running thread is equal to or greater than the one of an interrupt, the 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.

An important implication of this interrupt model is that users should always use distinct priorities for threads and interrupt handlers; otherwise, tasks could delay the interrupt handling. The implication of this (correct and important) recommendation is that the user should not assign priorities in the Interrupt_Priority range to software tasks.

The procedure Detach_Handler detaches a previously attached interrupt handler. Subsequent deliveries of the interrupt are handled by a default handler which is part of the kernel itself. This call has no effect if an interrupt handler was not previously attached.

The function Current_Handler gets the address of the handler which is currently attached to an interrupt.

4.3 Errors

Errors in the kernel are signalled to the application program by means of the Ada exception mechanism.

Dynamic memory should only be allocated (from a preallocated pool) in the initialization of the kernel, as a result of task creation (ATCBs, stacks, . . .). If the preallocated pool is completely full, any request for new space raises Tasking_Error.

The stack allocated for each task is protected using the memory access protection functionality provided by the MEC in ERC32. When the stack for any task is not enough for its computation, a Storage_Error is raised (with a “stack overflow” message) when a task tries to write outside the bounds of its stack.

When attaching protected procedures to interrupts, the ceiling priority of the protected object should be carefully chosen. The compiler checks that the interrupt for the protected object is in the range of System.Interrupt_Priority. This range of priorities is mapped to the 15 different interrupt levels provided by the SPARC architecture. Therefore, when assigning priorities to protected objects, the priority value must be at least equal to the priority of the hardware interrupt. Otherwise, the execution of the program is erroneous (ALRM C.3.1). The kernel cannot automatically detect this wrong priority assignment, and it must be done by the user.

One important restriction of ORK is related to the implementation of protected objects. Calling an operation that is potentially blocking during a protected action is a bounded error (see ALRM 9.5.1). The implementation requires the user not to make potentially blocking calls within protected subprograms, as this greatly simplifies the implementation of the underlying mutexes required by protected objects.

WARNING 4.2
The user must carefully write protected subprograms to avoid bounded errors when calling potentially blocking operations during protected actions.

Potentially blocking operations are (ALRM 9.5.1):

- *Protected entry calls;*
- *delay until;*
- *Ada.Synchronous_Task_Control.Suspend_Until_True (ALRM D.10);*

Notice that an external call on a protected subprogram with the same target object as that of the protected action, or a call on a subprogram whose body contains a potentially blocking operation is also a blocking operation (ALRM 9.5.1).

4.4 Tailoring the kernel

ORK has configuration parameters to tailor it to different applications. This parameters are mainly in kernel-parameters.ads. That file, as well as other ORK files, can be found at `/usr/local/openravenscar/src/gcc-2.8.1/ada`.

You can modify that file and rebuild the whole cross-compilation system in order to build an ORK that satisfies your requirements.

It is recommended that the file kernel-parameters.ads be previously compiled:

```
$ sparc-ork-gcc -c -gnatpg kernel-parameters.ads
```

After updating kernel-parameters.ads, the ORK cross-compilation system can be rebuilt (see section 4.4.3).

4.4.1 Configurable parameters

The configurable parameters included in the Kernel.Parameters package are:

Max_Tasks: Maximum number of threads supported by the kernel.

Stack_Area_Size: Maximum space to allocate stacks for threads.

Default_Stack_Size: Default size for the stack of a thread.

Interrupt_Stack_Size: Size of the interrupt stack.

Max_Priority: Last value of subtype System.Priority.

Clock_Frequency: Frequency of the Real Time Clock and General Purpose Timer input clock.

Clock_Interrupt_Period: Period of Real Time Clock interrupt. The range of this constant depends on Clock_Frequency.

The configurable parameter which defines the memory space available in the board is included in the linker script file `commands.ld`, which can be found in `/usr/local/openravenscar/sparc-ork-elf/lib/` directory. You can edit that file and change the `RAM_SIZE` value. It is not needed to rebuild the ORK cross-compilation system when this parameter is changed.

4.4.2 Interrupt names

ORK interrupt names are as close as possible to names given in MEC revision A Device Specification document [22]. For example, the ORK name Watch_Dog_Time_Out denotes the Watch Dog time-out interrupt.

ORK interrupt names are defined in Kernel.Peripherals. These interrupt names are available to GNARL by appropriate renames in the GNULL package System.OS_Interface. As a result, the standard Ada package Ada.Interrupts.Names contains the following interrupt names:

Watch_Dog_Time_Out: This interrupt occurs if the watchdog timer times out.

Real_Time_Clock: This interrupt is issued by the real time clock timer tick.

General_Purpose_Timer: This interrupt is issued by the general purpose timer.

DMA_Time_Out: This interrupt occurs if the DMA session exceeds permitted time.

DMA_Access_Error: This interrupt occurs if the DMA performs an access violation or illegal access.

UART_Error: This interrupt is generated by the UARTs if an error is detected.

UART_A_Ready: This interrupt is generated by the UART channel A each time a data word has been correctly received and each time a data word has been sent.

UART_B_Ready: This interrupt is generated by the UART channel B each time a data word has been correctly received and each time a data word has been sent.

Correctable_Error_In_Memory: This interrupt occurs if the EDAC (Error Detection And Correction) detects and corrects an error.

Masked_Hardware_Errors: This occurs when there is a hardware error set in the Error and Reset Status Register and the error is masked.

External_Interrupt_(4,3,2,1,0): The sources of these interrupts are located outside the ERC32. Consequently these interrupts are inputs to the MEC through pins ExtINT(4,3,2,1,0).

4.4.3 Compiling the kernel

Procedures for rebuilding the whole ORK cross-compilation system and adapting ORK are the in `/usr/local/openravenscar/src` directory.

In order to rebuild the whole ORK cross-compilation system, you need to do the following:

1. Edit the fi le `user.cfg` in order to change the location of ORK, if so desired.

2. Enter the command

   ```
   $./build_ada HOST
   ```

   in the `/usr/local/openravenscar/src` directory where HOST can be either i[3456]86-pc-linux-gnu or sparc-sun-solaris2.*

The last procedure will take about 15-30 minutes on a modern computer (e.g. 13 minutes on a 600 MHz Pentium III with 128 Mbytes and IDE disks, and 38 minutes on a 360 MHz UltraSparc II with 128 Mbytes and IDE disks) As a result, the ORK cross-compilation system will be installed.

As this procedure takes a long time, it is possible to selectively rebuild only the ORK adalib. However, ORK must have been previously compiled in order to do this.

If the ORK cross-compilation system has been built previously, the subdirectories called `build-sparc-tools` and `src` must already exist in `/usr/local/openravenscar/src`. It is then possible to rebuild only the ORK adalib by typing

```
$./build_adalib
```

in the `/usr/local/openravenscar/src` directory.

WARNING 4.3
It is necessary to have a gnat-3.13 installed for rebuilding or adapting ORK.

Appendix A

The Ravenscar profile

A.1 Introduction

The Ravenscar Profile (Reliable Ada Verifiable Executive Needed for Scheduling Critical Applications in Real-Time) is the best known result of the 8th International Real-Time Ada Workshop (IRTAW'8), which was held in April 1997 in Ravenscar, Yorkshire [10, 16, 14]. The purpose of the profile is to identify a subset of the tasking features of Ada which can be implemented using a small, reliable kernel. The expected benefits of this approach are:

- Improved memory and execution time efficiency, by removing features with a high overhead.
- Improved reliability, by removing non-deterministic and non analysable features.
- Improved timing analysis, by removing non-deterministic and non-analysable features.

The profile was revised at the 9th International Real-Time Ada Workshop (IRTAW'9) that was held in March 1999 near Tallahassee, Florida [8]. The revision took into account early experience with a commercial Ravenscar real-time kernel called Raven [20, 19], as well as some other experiences with small size kernels with a similar orientation [30, 31, 17, 41, 33, 32]. The next meeting, IRTAW'10, was held near Avila, Spain, in September 2000. The meeting discussed up-to-date experience in implementing the profile, including ORK [18, 40], and made some recommendations about exceptions, memory management, and interrupt support [42].

The profile is now part of the ISO report *Guide for the use of the Ada Programming Language in High Integrity Systems* (HIS) [29]. The summary presented here is based on Alan Burns' comprehensive account of the definitions in IRTAW'8, IRTAW'9, and IRTAW'10 [13].

The definition of the Ravenscar profile is based on Ada 95, including the Systems Programming and Real-Time annexes. It only addresses tasking constructs, as the reliability aspects of the sequential part of Ada are covered in other sections of the HIS report [29].

The profile is based on a computation model with the following features:

- A single processor.
- A fixed number of tasks.

- A single invocation event for each task. The invocation event may be generated by the passing of time (for time-triggered tasks) or by a signal from either another task or the environment (for sporadic tasks).
- Task interaction only by means of shared data with mutually exclusive access.

This set of features effectively supports building systems with the following kinds of components:

- Periodic tasks.
- Program driven sporadic tasks.
- Interrupt driven sporadic tasks.
- Protected objects implementing shared data (typically with no entries).
- Protected objects for event synchronization (with at most one entry called by a single signalling task).

These components are considered to be expressive enough for implementing high integrity systems for space applications on a single processor.

A.2 Definition

The Ravenscar profile is defined by the following restrictions [13]:

A.2.1 Forbidden features

RP1 Task types and object declarations other than at the library level. Thus, there is no hierarchy of tasks.

RP2 Dynamic allocation and unchecked deallocation of protected and task objects.

RP3 Requeue.

RP4 ATC (asynchronous transfer of control via the asynchronous_select statement.)

RP5 Abort statements, including Abort_Task in package Ada.Task_Identification.

RP6 Task entries.

RP7 Dynamic priorities.

RP8 Calendar package.

RP9 Relative delays.

RP10 Protected types and object declarations other than at the library level.

RP11 Protected types with more than one entry.

RP12 Protected entries with barriers other than a single boolean variable declared within the same protected type.

RP13 An entry call to a protected entry with a call already queued.

RP14 Asynchronous task control.

RP15 All forms of select statements.

RP16 User-defi ned task attributes.

RP100 Dynamic interrupt handler attachments.

In addition to these restrictions, implementations can make the following assumption:

RP17 Tasks do not terminate.

A.2.2 Supported features

The above restrictions still support a wide range of tasking features, such as:

RP18 Task objects, restricted as above.

RP19 Protected objects, restricted as above.

RP20 Atomic and Volatile pragmas.

RP21 Delay until statements.

RP22 Ceiling Locking policy and FIFO within priorities dispatching.

RP23 Count attribute (but not within entry barriers).

RP24 Task identifi ers, e.g. T'Identity, E'Caller.

RP25 Synchronous task control.

RP26 Task discriminants.

RP27 Real_Time package.

RP28 Protected procedures as statically bound interrupt handlers.

A.2.3 Dynamic semantics

Two aspects of the profi le require their dynamic semantics to be defi ned:

RP29 If an entry call is made on an entry that already has a queued call (i.e. the queue length would become 2), then Program_Error is raised.

This is consistent with the use of Program_Error in the defi nition of synchronous suspension objects [6] D.10(10).

RP30 If a task attempts to terminate, this is classifi ed as a bounded error (i.e. there is a documentation requirement on the implementation to defi ne its effect), one allowed outcome being the permanent suspension of the task.

RP101 If a task executes a potentially blocking operation from within a protected object then Program_Error *must* be raised (unless a subprogram call is made to a foreign language domain). The full language defi nes this as a bounded error, Program_Error being just one of the allowed outcomes.

A.3 Denoting the restrictions

Most of the Ravenscar profi le restrictions can be checked at compile time in Ada 95 by means of standard identifi ers that can be used with the pragma Restrictions [6, 13.12; D7; H.4]. The following identifi ers apply:

RP31 No_Task_Hierarchy

RP32 No_Abort_Statements

RP33 No_Task_Allocators

RP34 No_Dynamic_Priorities

RP35 No_Asynchronous_Control

RP36 Max_Task_Entries => 0

RP37 Max_Protected_Entries => 1

RP38 Max_Asynchronous_Select_Nesting => 0

RP39 Max_Tasks => N – defi ned by the application

However, these identifi ers are not suffi cient to enforce all the profi le restrictions. The following additional restriction identifi ers have been proposed to that end:

RP40 Simple_Barrier_Variables

RP41 Max_Entry_Queue_Depth => 1 – or, in general, N

RP42 No_Calendar

RP43 No_Relative_Delay

RP44 No_Protected_Type_Allocators

RP45 No_Local_Protected_Objects

RP46 No_Requeue

RP47 No_Select_Statements

RP48 No_Task_Attributes

RP49 No_Task_Termination

RP50 No_Dynamic_Interrupt_Handlers

The meanings of these restrictions are straightforward, and foloow directly from the above defi nitions.

The last restriction forbids calling the subprograms defi ned in Ada.Interrupts. The type Interrupt_Id may be used in application programs.

Appendix B

The Ravenscar restrictions in GNAT

This appendix is based on GNAT 3.13[4].

B.1 The pragma Ravenscar

Most of the Ravenscar profile restrictions can be enforced 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 restriction identifiers, and thus a number of additional restriction identifiers have been proposed at IRTAW8, [10], IRTAW9 [8], and IRTAW10 [42] for this purpose. The complete set of Ravenscar restrictions is listed in appendix A and in reference [13].

The approach adopted in GNAT is to use an implementation defined pragma (pragma Ravenscar) in order to establish the complete set of restrictions [4]. However, the restrictions established by the pragma Ravenscar are not exactly the same as those defined in the profile. This means that there some small differences between the GNAT Ravenscar pragma and the profile defined by IRTAW [13].

Table B.1 summarizes the restrictions enforced by this pragma and their relationship with the profile restrictions.[1]

From the table the following conclusions can be drawn:

1. Two restrictions which are part of the Ravenscar profile definition [13] are not enforced by the pragma Ravenscar:

   - RP39 Max_Tasks => N

     Notice that this restriction could not be included in the pragma as the value of N is application dependent. However, since Max_Tasks is a standard restriction parameter (ALRM D.7), this restriction can be enforced at compile time using the pragma Restriction.

   - RP50 No_Dynamic_Interrupt_Handlers

     This a new restriction which was not required until the IRTAW10 revision of the profile. Therefore it was not possible to include it in GNAT 3.13, which was released before that event. Although it should be included in future versions of GNAT, some kind of workaround, such as raising an exception if the application calls an Ada.Interrupts subprogram, is the only thing that can be done for the moment.

[1]The RP codes refer to appendix A.

Table B.1: The pragma Ravenscar (PR) and the Ravenscar profi le (RP) restrictions

Code	**Restriction**	**RP restriction**	**Comments**
PR1	No_Task_Hierarchy	RP31	
PR2	No_Abort_Statements	RP32	
PR3	No_Task_Allocators	RP33	
PR4	No_Dynamic_Priorities	RP34	
PR5	No_Asynchronous_Control	RP35	
PR6	Max_Task_Entries => 0	RP36	
PR7	Max_Protected_Entries => 1	RP37	
PR8	Max_Asynchronous_Select_Nesting => 0	RP38	
—	Max_Tasks => N	RP39	Not included in PR
PR10	Boolean_Entry_Barriers	RP40	Different identifi er
PR11	No_Entry_Queue	RP41	Different semantics (yet acceptable for N=1)
PR12	No_Calendar	RP42	
PR13	No_Relative_Delay	RP43	
PR14	No_Protected_Type_Allocators	RP44	
PR15	No_Local_Protected_Objects	RP45	
PR16	No_Requeue	RP46	
PR17	No_Select_Statements	RP47	
PR18	No_Task_Attributes	RP48	
PR19	No_Task_Termination	RP49	
—	No_Dynamic_Interrupt_Handlers	RP50	Not included in PR
PR20	Static_Storage_Size	Not required by RP	
PR21	No_Dynamic_Interrupts	—	Contradicts RP
PR22	No_Terminate_Alternatives	—	Implied by RP47
PR23	Max_Select_Alternatives => 0	—	Implied by RP47

2. Four restrictions which are not part of the Ravenscar profile proper, but may be considered useful [8], are not enforced by the pragma Ravenscar:

   - RP51 No_Exception_Handlers
   - RP52 No_Standard_Storage_Pools
   - RP53 No_IO
   - RP54 No_Nested_Finalization

   However, since all of them are available in GNAT as restriction identifiers,[2] they can be enforced at compile time using the pragma Restriction.

3. One restriction,

   - PR10 Boolean_Entry_Barriers

   has a different name, but the same semantics as the identifier proposed in the Ravenscar profile definition:

   - RP40 Simple_Barrier_Variables

   This difference may cause only portability problems, but does not prevent the compiler form enforcing the right restriction.

4. One restriction

   - PR11 No_Entry_Queue

   has a different name, and a slightly different semantics from the restriction proposed in the Ravenscar profile definition:

   - Max_Entry_Queue_Depth => N

   However, since N = 1 in the Ravenscar profile, the restriction of the GNAT pragma Ravenscar is acceptable. Moreover, Max_Entry_Queue_Depth => N is implemented in GNAT as a restriction parameter, and can thus be explicitly set with the Restrictions pragma.

5. Two of the restrictions enforced by the pragma Ravenscar are redundant:

   - PR22 No_Terminate_Alternatives
   - PR23 Max_Select_Alternatives => 0

   Both restrictions are implied by RP47. However, it does no harm to have them included in the pragma Ravenscar.

6. One restriction is enforced by the pragma Ravenscar, but is in contradiction with the Ravenscar profile definition;

   - PR21 No_Dynamic_Interrupts

[2]Notice that RP51 and RP52 are not part of the Ada 95 standard, but are provided in GNAT as implementation-defined restrictions [4, section 4, 57].

The current profi le defi nition [13] specifi es the restriction No_Dynamic_Interrupt_Handlers instead (RP50). The problem with No_Dynamic_Interrupts is that it forbids any dependence from Ada.Interrupts [4], thus preventing the use of type Interrup_Id which is required by pragma Attach_Handler. The result is that the current defi nition of pragma Ravenscar in GNAT 3.13 does not support static interrupt handlers, which are explicitly allowed by the Ravenscar profi le. The only workaround for the moment is to modify the compiler in order to ignore such restriction.

The conclusion is that, although there are some differences between the restrictions imposed by the pragma Ravenscar and the IRTAW defi nition of the Ravenscar profi le, the pragma can be used in its present form, together with other confi guration pragmas (see below), to check Ravenscar compliance of Ada programs compiled with GNAT.

B.2 Other pragmas

B.2.1 Pragma Restrictions

Some of the Ravenscar restrictions can be enforced with restriction identifi ers. The Ravenscar restriction identifi ers that are accepted by GNAT 3.13 are listed in table B.2.

Table B.2: Ravenscar restriction identifi ers in GNAT

Code	GNAT restriction	RP restriction	Comments
RI1	No_Task_Hierarchy	RP31	
RI2	No_Abort_Statements	RP32	
RI3	No_Task_Allocators	RP33	
RI4	No_Dynamic_Priorities	RP34	
RI5	No_Asynchronous_Control	RP35	
RI6	Max_Task_Entries => 0	RP36	
RI7	Max_Protected_Entries => 1	RP37	
RI8	Max_Asynchronous_Select_Nesting => 0	RP38	
RI9	Max_Tasks => N	RP39	N is application dependent
RI10	Boolean_Entry_Barriers	RP40	Different identifi er
RI11	Max_Entry_Queue_Depth => 1	RP41	
	No_Entry_Queue	RP41	Different semantics
RI12	No_Calendar	RP42	
RI13	No_Relative_Delay	RP43	
RI14	No_Protected_Type_Allocators	RP44	
RI15	No_Local_Protected_Objects	RP45	
RI16	No_Requeue	RP46	
RI17	No_Select_Statements	RP46	
RI18	No_Task_Attributes	RP48	
RI19	No_Task_Termination	RP49	

The set of restriction identifi ers covers the Ravenscar profi le specifi cations (except for interrupt handler support), and thus can be used to enforce the profi le at compile time, with that only exception.

B.2.2 Scheduling related pragmas

The following standard pragmas should be used with Ravenscar profile programs:

- pragma Task_Dispatching_Policy (FIFO_Within_Priorities)
- pragma Locking_Policy (Ceiling_Locking)

The pragma Queuing_Policy is not needed as the maximum length of protected entry queues is restricetd to 1 (RP41).

B.3 The gnat.adc file for the Ravenscar profile

GNAT requires configuration pragmas to be put in a separate file, called gnat.adc [5]. A template gnat.adc for Ravenscar compliant programs follows.

```
-- gnat.adc - configuration file template for the Ravenscar profile
pragma Ravenscar;
pragma Restrictions (Max_Tasks => N);
-- N must be equal to the number of tasks of the application
-- pragma Restrictions(No_Allocators); - does not work properly in GNAT 3.13
pragma Restrictions(No_IO);
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy   (Ceiling_Locking);
```

The maximum number of tasks is application-dependent. There is an upper bound on this number, which depends on the underlying kernel. See the *Operation Manual* for the details.

Appendix C

Example program

C.1 Description

The goal of this example is to show the functionality of the Open Ravenscar Real-Time Kernel.

The example program has three tasks which spend their computation time calling the Whetstone benchmark. This benchmark performs floating point operations developed for the Performance Issues Working Group (PIWG) test suite.

In order to exercise communication among tasks, two of the three tasks interact through a protected object. One of the tasks is sporadic and the other one is periodic. The third task is an independent periodic task.

The sporadic task is activated by a hardware interrupt for which it waits on an protected entry with a simple boolean barrier. A protected procedure is used to handle the interrupt, in accordance with the ORK interrupt model. The protected procedure opens the barrier and then the sporadic task becomes runnable. After the task executes its code the barrier is closed again.

A periodic task activates the hardware interrupt. The Memory Controller (MEC) device of the ERC32 core computer has special registers which allows the user to force hardware interrupts by software. Such registers are used by the periodic task to activate the hardware interrupt at regular intervals.

All the tasks print the value of Real_Time.Clock whenever they start and finish executing their body. Only absolute delays and the monotonic clock of the Real_Time package are used. In order to avoid undesirable interactions between input-output and task scheduling, the special Put operation of the Kernel.Serial_Output package is used.

The example program is designed to cover all the features which are needed in space embedded applications. In particular, the example includes:

- Task management
- Task synchronization
- Time keeping and absolute delays
- Ada interrupt management
- Floating point calculations

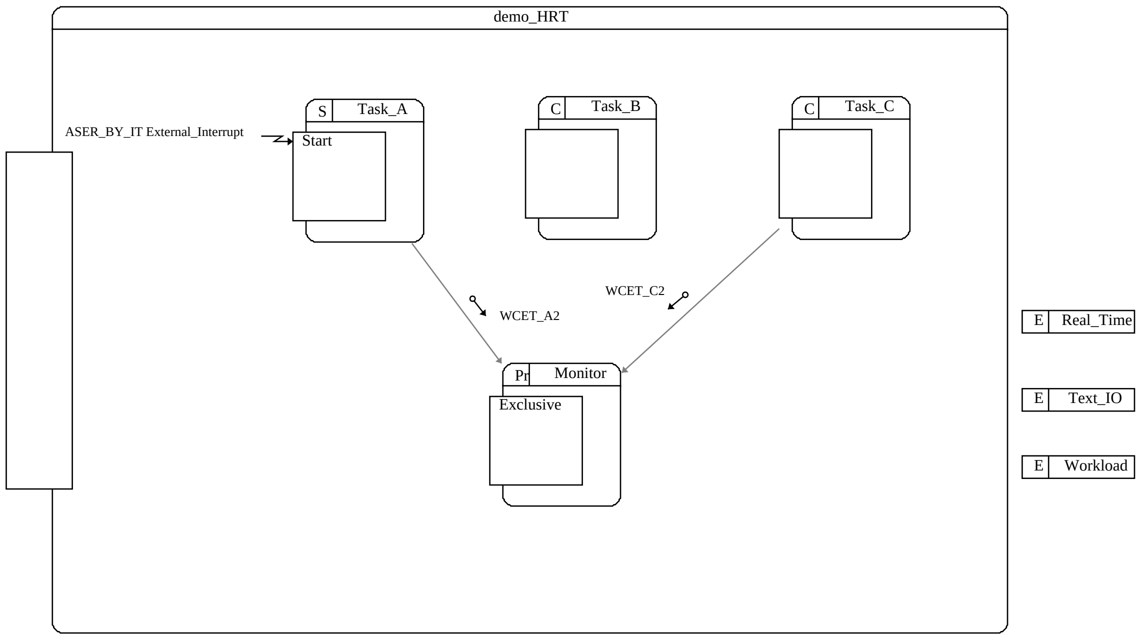

Figure C.1: Example task set

C.2 Temporal requirements of the tasks

Figure C.1 shows the structure of the task set. The task set will be analysed for the temporal requirements of the tasks as shown in table C.1. The period for task A is interpreted as a minimum inter-arrival time.

Task	Period	Activities
A	14	a_1, a_2
B	20	b_1
C	36	c_1, c_2

Table C.1: Temporal requirements of the example tasks

Tasks A and C contain two logical blocks of activities, while task B has only one. Activity a_1 corresponds to internal computation of task A, and a_2 to the execution time of task A inside resource Monitor. Similarly, c_1 corresponds to the internal execution time of task C, and c_2 to the execution time of task C inside resource Monitor. Finally, b_1 corresponds to the whole execution of task B. By extension, the same set of symbols denote the WCET of the corresponding block of activity.

Task (block)	Priority	WCET	Resource
A(a_1)	Priority'Last	1	None
A(a_2)	Priority'Last	2	Monitor
B (b_1)	Priority'Last - 1	6	None
C (c_1)	Priority'Last - 2	2	None
C (c_2)	Priority'Last	6	Monitor

Table C.2: Priority assignment and Worst Case Execution Time of activities

Table C.2 shows the priorities assigned to the tasks. Task A has the highest priority, task C has the lowest priority, task B has a medium priority.

C.3 Schedulability analysis

The Ravenscar profi le requires the program to be compiled with pragma Task_Dispatching _Policy(FIFO_Within_Priorities) and pragma Locking_Policy (Ceiling_Locking) (see 3.2.2). Therefore, the maximum response time of every task can be evaluated using equation C.1.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{C.1}$$

Which is solved using a recurrence relation:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times C_j$$

As immediate ceiling locking is used, the maximum blocking time can be evaluated for every task.

Task A: can suffer a blocking time equal to WCET of activity c_2, i.e. $B_a = 6$.

Task B: can suffer a blocking time equal to WCET of activity c_2, i.e. $B_b = 6$.

Task C: is the lowest priority task and so can not suffer blocking, i.e. $B_c = 0$.

The maximum response time of every task can now be calculated. The minimum inter-arrival time will be used as the period in order to calculate the worst case response time for the low priority task.

Following common practice, an initial value w_i^0 equal to the sum of the WCET of higher priority task plus the WCET of the task itself is used:

$$w_a^1 = 3 + 6 = 9$$

$$\begin{aligned} w_b^1 &= 6 + 6 + \left\lceil \frac{9}{14} \right\rceil \times 3 = 15 \\ w_b^2 &= 6 + 6 + \left\lceil \frac{15}{14} \right\rceil \times 3 = 18 \\ w_b^3 &= 6 + 6 + \left\lceil \frac{18}{14} \right\rceil \times 3 = 18 \end{aligned}$$

$$\begin{aligned} w_c^1 &= 8 + \left\lceil \frac{17}{14} \right\rceil \times 3 + \left\lceil \frac{17}{20} \right\rceil \times 6 = 20 \\ w_c^2 &= 8 + \left\lceil \frac{20}{14} \right\rceil \times 3 + \left\lceil \frac{20}{20} \right\rceil \times 6 = 20 \end{aligned}$$

Figure C.2 shows the schedule of the tasks starting at time zero for 60 time units of 100ms each. Up arrows denote activation time and down arrows denote deadlines. Filled boxes denote sections executed at ceiling priority.

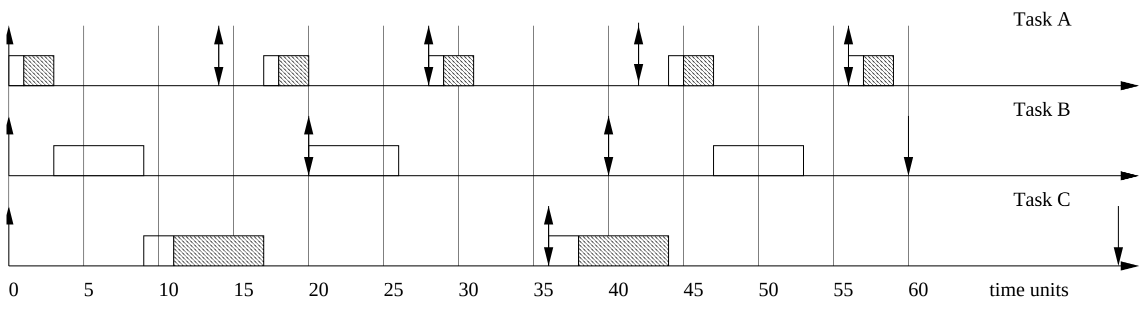

Figure C.2: Schedule of tasks

C.4 Example program output

The output of the example program shows the start and termination time of each task cycle. You can use TSIM for executing it, the options `-freq 10 -fast_uart` can be used to set the clock frequency defined in Kernel.Parameters and to set “infinite” speed in the UART channel.

With a time unit of 100ms the actual output[1] is:

```
$ tsim -freq 10 -fast_uart demo

TSIM - remote SPARC simulator 0.1 (demo version)

serial port A on stdin/stdout
allocated 4096 K RAM memory
allocated 2048 K ROM memory
section: .text at 0x2000000, size 183680 bytes
section: .data at 0x202cd80, size 6148 bytes
section: .bss at 0x202e588, size 408660 bytes
tsim> go
Task A running   RT.Clock =   0.001300500
Task A finishing RT.Clock =   0.306737700
Task B running   RT.Clock =   0.308817300
Task B finishing RT.Clock =   0.915951300
Task C running   RT.Clock =   0.917942600
Task C finishing RT.Clock =   1.728120300
Task A running   RT.Clock =   1.730182000
Task A finishing RT.Clock =   2.035818800
Task B running   RT.Clock =   2.037934000
Task B finishing RT.Clock =   2.645087400
Task A running   RT.Clock =   2.801267200
Task A finishing RT.Clock =   3.106838900
Task C running   RT.Clock =   3.600678000
Task C finishing RT.Clock =   4.410930900
Task A running   RT.Clock =   4.413009400
Task A finishing RT.Clock =   4.718632300
Task B running   RT.Clock =   4.720762600
Task B finishing RT.Clock =   5.327842300
Task A running   RT.Clock =   5.601311000
Task A finishing RT.Clock =   5.906893700
```

[1]The start and termination time of each task cycle can vary depending of the ORK version

```
...
```

There are small variations with respect to the timetable of figure C.2 which are due to:

1. Kernel overhead.
2. The code of the tasks includes calls to the Whetstone benchmark with an actual parameter which suits the WCET defined in table C.2. As a result, the execution time of protected operations, delay settings, clock readings, and other operations increases the WCET defined in table C.2.

C.5 Example code

demo.adb

```
with Tasks;
with System;

procedure Demo is

  pragma Priority (System.Priority'First);

begin

  Tasks.Background;

end Demo;
```

tasks.ads

```
package Tasks is

  procedure Background;

end Tasks;
```

tasks.adb

```
with Kernel.Serial_Output;

with Ada.Interrupts.Names;

with Ada.Real_Time;
use type Ada.Real_Time.Time_Span;

with System;
with Workload;
with Force_External_Interrupt_2;

package body Tasks is

  Time_Unit : constant Ada.Real_Time.Time_Span :=
                Ada.Real_Time.Milliseconds (100);

  -- This constant was meausured with sis -freq 10
  -- A program for measuring this constant can be built with
  -- make -f Makefile.measure
```

```
Time_per_Kwhetstones : constant Ada.Real_Time.Time_Span :=
                   Ada.Real_Time.Nanoseconds (539021);

procedure Execution_Time (Time : Ada.Real_Time.Time_Span) is

begin
  Workload.Small_Whetstone (Time / Time_per_Kwhetstones);
end Execution_Time;

-- 500 Milliseconds is the initial offset for the tasks
-- It is enough time to elaborate the program

Offset : constant Ada.Real_Time.Time_Span :=
       Ada.Real_Time.Milliseconds (500);

Time_Zero : constant Ada.Real_Time.Time :=
         Ada.Real_Time.Time_of (0, Ada.Real_Time.Time_Span_Zero) +
         Offset;

-- This procedure prints Real_Time.Clock - Time_Zero

procedure Print_RTClok is
  Seconds_Count_From_Time_Zero : Ada.Real_Time.Seconds_Count;
  Time_Span_From_Time_Zero : Ada.Real_Time.Time_Span;
  Duration_From_Time_Zero : Duration;

begin

  Ada.Real_Time.Split (Ada.Real_Time.Clock - Offset,
                 Seconds_Count_From_Time_Zero,
                 Time_Span_From_Time_Zero);
  Duration_From_Time_Zero := Duration (Seconds_Count_From_Time_Zero) +
             Ada.Real_Time.To_Duration (Time_Span_From_Time_Zero);
  Kernel.Serial_Output.Put (" RT.Clock = ");
  Kernel.Serial_Output.Put (Duration'Image(duration_From_Time_Zero));

end Print_RTClok;

-- Temporal parameters of Tasks

subtype Tasks is character range 'A' .. 'C';

WCET_A1 : constant Ada.Real_Time.Time_Span := 1 * Time_Unit;
WCET_A2 : constant Ada.Real_Time.Time_Span := 2 * Time_Unit;
Period_A : constant Ada.Real_Time.Time_Span := 14 * Time_Unit;

WCET_B : constant Ada.Real_Time.Time_Span := 6 * Time_Unit;
Period_B : constant Ada.Real_Time.Time_Span := 20 * Time_Unit;

WCET_C1 : constant Ada.Real_Time.Time_Span := 2 * Time_Unit;
WCET_C2 : constant Ada.Real_Time.Time_Span := 6 * Time_Unit;
Period_C : constant Ada.Real_Time.Time_Span := 36 * Time_Unit;

-- Priority of the interrupt used

Priority_Of_External_Interrupts_2 : constant System.Interrupt_Priority :=
                             System.Interrupt_Priority'First + 9;

procedure Background is
```

```
begin
  loop
    null;
  end loop;
end Background;

task A is
  pragma Priority (System.Priority'Last);
end A;

task B is
  pragma Priority (System.Priority'Last − 1);
end B;

task C is
  pragma Priority (System.Priority'Last − 2);
end C;

protected Monitor is

  pragma Priority (System.Priority'Last);

  procedure Exclusive (Time : Ada.Real_Time.Time_Span;
                       Running_Task : Tasks);

end Monitor;

−− This task forces a interrupt every Period_A

task Interrupt is
  pragma Priority (Priority_Of_External_Interrupts_2);
end Interrupt;

protected Interrupt_Semaphore is
  pragma Priority (Priority_Of_External_Interrupts_2);

  entry Wait;

  procedure Signal;
  pragma Interrupt_Handler (Signal);
  pragma Attach_Handler (Signal,
                         Ada.Interrupts.Names.External_Interrupt_2);

private

  Signaled : Boolean := False;

end Interrupt_Semaphore;

protected body Interrupt_Semaphore is

  entry Wait when Signaled is

  begin
    Signaled := False;
  end Wait;

  procedure Signal is
  begin
    Signaled := True;
```

```
   end Signal;

end Interrupt_Semaphore;

task body Interrupt is
   Next_Time : Ada.Real_Time.Time := Time_Zero;
begin
   loop
      delay until Next_Time;
      Force_External_Interrupt_2;
      Next_Time := Next_Time + Period_A;
   end loop;
end Interrupt;

protected body Monitor is

   procedure Exclusive (Time : Ada.Real_Time.Time_Span;
                        Running_Task : Tasks) is

   begin
      Execution_Time (Time);
      Kernel.Serial_Output.Put ("Task ");
      Kernel.Serial_Output.Put (Running_Task);
      Kernel.Serial_Output.Put (" finishing");
      Print_RTClok;
      Kernel.Serial_Output.New_Line;
   end Exclusive;

end Monitor;

task body A is
begin
   loop
      Interrupt_Semaphore.Wait;
      Kernel.Serial_Output.Put ("Task A running ");
      Print_RTClok;
      Kernel.Serial_Output.New_Line;
      Execution_Time (WCET_A1);
      Monitor.Exclusive (WCET_A2, 'A');
   end loop;
end A;

task body B is
   Next_Time : Ada.Real_Time.Time := Time_Zero;
begin
   loop
      delay until Next_Time;
      Kernel.Serial_Output.Put ("Task B running ");
      Print_RTClok;
      Kernel.Serial_Output.New_Line;
      Execution_Time (WCET_B);
      Next_Time := Next_Time + Period_B;
      Kernel.Serial_Output.Put ("Task B finishing");
      Print_RTClok;
      Kernel.Serial_Output.New_Line;
   end loop;
end B;

task body C is
   Next_Time : Ada.Real_Time.Time := Time_Zero;
```

```
  begin
    loop
      delay until Next_Time;
      Kernel.Serial_Output.Put ("Task C running ");
      Print_RTClok;
      Kernel.Serial_Output.New_Line;
      Execution_Time (WCET_C1);
      Monitor.Exclusive (WCET_C2, 'C');
      Next_Time := Next_Time + Period_C;
    end loop;
  end C;

end Tasks;
```

force_external_interrupt_2.adb

```
with Kernel.Peripherals.Registers;
-- to get definitions of MEC registers such as:
--      Test_Control
--      Interrupt_Mask
--      Interrupt_Force

procedure Force_External_Interrupt_2 is

  package KPR renames Kernel.Peripherals.Registers;

  -- The MEC registers must be accesses as a whole.
  -- The workaround used to force GNAT to generate proper instructions is:
  -- Registers type definition are cualified with pragma Atomic
  -- and auxiliary objects are used to write the MEC registers

  Test_Control_Auxiliary : KPR.Test_Control_Register :=
                       KPR.Test_Control;
  Interrupt_Mask_Auxiliary : KPR.Interrupt_Mask_Register :=
                         KPR.Interrupt_Mask;
  Interrupt_Force_Auxiliary : KPR.Interrupt_Force_Register :=
                          KPR.Interrupt_Force;

begin

  Test_Control_Auxiliary.Interrupt_Force_Register_Write_Enable := True;
  Interrupt_Mask_Auxiliary.External_Interrupt_2 := False;
  Interrupt_Force_Auxiliary.External_Interrupt_2 := True;

  KPR.Test_Control := Test_Control_Auxiliary;
  KPR.Interrupt_Mask := Interrupt_Mask_Auxiliary;
  KPR.Interrupt_Force := Interrupt_Force_Auxiliary;

end Force_External_Interrupt_2;
```

gnat.adc

```
pragma Ravenscar;

pragma Restrictions (No_Asynchronous_Control);
pragma Restrictions (Max_Tasks => 4 );
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
```

Appendix D

Known bugs and limitations

D.1 Introduction

This appendix lists three kinds of bugs and limitations of the openravenscar 2.2b compilation system:

1. Unsolved GNAT 3.13 problems related to the Ravenscar Profi le restrictions.

2. GNAT 3.13 problems which can be solved with some work-arounds.

3. Bugs and limitations of the ORK-ERC32 2.2b kernel itself.

The bugs are detailed in the next sections. The actions for bounded errors related to tasking are also included.

D.2 Unsolved GNAT 3.13 bugs

These bugs imply a limitation in the compile-time checking of the Ravenscar profi le restrictions.

1. The restriction Max_Task => N is not enforced at compilation time.

2. The restriction No_Protected_Type_Allocators is not enforced at compilation time.

3. The restriction No_Local_Protected_Objects is not enforced at compilation time when declaring protected objects within a task.

D.3 Solved GNAT 3.13 bugs

These bugs originally in GNAT 3.13 have been fi xed by minor changes in the compiler sources, and not present in openravenscar 2.2b.

1. The restriction Boolean_Entry_Barriers is not properly implemented in GNAT 3.13, which causes a spurious error when compiling some valid, RP-compliant programs.

2. GNAT 3.13 does not allow static protected interrupt handlers, as required by the Ravenscar Profi le defi nition.

D.4 ORK-ERC32 2.2b bugs related to the Ravenscar Profile

1. The Count attribute does not work for protected entries.
2. The restriction Max_Entry_Queue_Depth => 1 is not properly checked at run time, and Program_Error is not raised when a task makes a call to a protected entry which already has a queued call.

D.5 Other ORK-ERC32 2.2b bugs

1. **Stack Protection**. ORK uses the hardware facilities of the ERC32 to detect thread stack overruns. Thread stacks are allocated in a linear memory space, and forbidden blocks are inserted between adjacent stacks in order to detect stack overruns.

   The size of forbidden blocks is Kernel.Parameters.Protection_Stack_Size, which has a default value of 256 bytes.

   However, if a thread allocates an amount of stack larger than the forbidden block then it can access the next thread stack without hardware detection.

2. **sparc-ork-gnatpsys**. The standard Ada 95 package System is not included in the predefined GNAT Ada library which is located in the `adainclude` directory. Instead, this package is embedded in the GNAT compilation system. The GNAT tool gnatpsys can be used to display its specification.

   The name of the equivalent tool for the GNAT-ORK cross-compilation system is sparc-ork-gnatpsys. The purpose of this tool is to list the System specification for the cross-compilation system. However, due to a bug the current version displays the System specification for the native GNAT compiler.

D.6 ORK-ERC32 2.2b tasking bounded errors

1. **Task Termination**. Task termination is a bounded error in the Ravenscar Profile (RP30). The default action of a task termination in ORK is null.

   This default action can be changed by means of the procedure System.Task_Primitives.Operations.Set_Exit_Task_Procedure.

2. **Potentially blocking statements in protected procedures and functions**. Calling a potentially blocking operation in a protected procedure is a bounded error (ALRM 9.5.1).

   This is currently detected by ORK and Program_Error is raised.

3. **Task Identification**. It is a bounded error to call Task_Identification.Current_Task (ALRM C.7.1) from an interrupt handler. ORK does not detect this bounded error.

   The effect of calling Task_Identification.Current_Task from an interrupt handler is to obtain a non sense Task_Id.

Appendix E

GNU General Public License

Version 2, June 1991
Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent li-

These requirements apply to the modifi ed work as a whole. If identifi able sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifi cations to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface defi nition fi les, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under

this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source fi le to most effectively convey the exclusion of warranty; and each fi le should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program’s name and an idea of what it does.
Copyright © yyyy name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type ‘show w’.  This is free software, and you are welcome
to redistribute it under certain conditions; type ‘show c’
for details.
```

The hypothetical commands `‘show w’` and `‘show c’` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `‘show w’` and `‘show c’`; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program ‘Gnomovision’
(which makes passes at compilers) written
by James Hacker.
signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

nes

Appendix F

Metrics

F.1 Ada LRM Annex C and D Metrics

This appendix lists the additional characteristics of the ORK-ERC32 2.2bwhich are specifi ed in Annex C and D of the Ada Language Reference Manual and are allowed by the Ravenscar profi le.

The metrics are shown in processor cycles therefore to obtain seconds they must be divided by the processor clock frequency.

F.1.1 Protected Procedure Handlers (C-3.1)

15. The worst case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C - (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how log it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

```
=== interrupt_overhead_test ===============================
 Cycles>  1969334
 Cycles>  1970270
 cycles A+B:  1969334
 cycles C:  1970270
 cycles C-(A+B) =  936
==========================================================
```

The average overhead of a protected procedure as interrupt handler is 936 processor cycles.

This metric has been measured with the simulators. Otherwise, special hardware such as signal generators and logic analyzers must be used to measure this overhead on real target. It must be said that other metrics have been measured on the simulators as well as on real hardware and the differences were insignifi cant.

F.1.2 Monotonic Time (D-8)

33. Values of Time_First, Time_Last, Time_Span_Last, Time_Span_Unit and Tick

- *Time_First:* seconds: -633437445 Duration: -8589934591.854780000

- *Time_Last:* seconds: 633437444 Duration: 8589934592.854780000
- *Time_Span_First:* -9223372036.854780000
- *Time_Span_Last:* 9223372036.854780000
- *Time_Span_Unit:* 0.000000001
- *Tick:* 0.000000050

The value of Tick is equal to the period of the input clock signal. The frequency of this clock signal for the eVAB-695E-Rev.C Temic Evaluation Board is 20 MHertz.

34. Properties of the underlying time base used for the clock and for type Time: range of values supported and any relevant aspects of the underlying hardware or OS facilities. Time is represented internally 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. Time is a count of ticks since the clock was started.

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. One of them (the Real Time Clock) is used 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 processor clock, and they use a two-stage counter.

In order to provide a high resolution clock, the least signifi cant part of the clock is held in the Real Time Clock hardware register and the Real Time Clock is programmed to interrupt periodically, updating the most signifi cant part of the clock.

As a result, the clock tick is equal to the period of the input signal of the downcounter which is the processor clock period in the current implementation.

The underlying implementation is deeply described in the Software Design Document (Kernel.Time subsection) and in [43].

35. Synchronization with external sources The system does not synchronize with any external source.

36. External environment properties affecting the behavior of the clock. The clock is a count of ticks since the clock was started, therefore its behavior only is affected by the manufacturing drift of the hardware clock.

39. An upper bound of a clock tick. The tick of the clock is equal to the period of the processor input signal and the clock is incremented by the hardware every tick. As a result, the tick has always the same value.

40. Upper bound of the size of a clock jump The clock is a count of ticks and is not synchronize with external sources. As a result, the clock does not jump.

41. Clock Drift The implementation of the clock uses a timer in periodic mode therefore the clock has the accuracy of the hardware. In order to probe it, a test against canon.inria.fr NPT V3 primary server shows a drift of -2 ± 2[1] seconds for a 4.3×10^5 seconds.

The resulting drift is less than 10^{-5} which is in the range of the manufacturing drift of the clocks.

The test was made on a eVAB-695E-Rev.C Temic Evaluation Board.

44. An upper bound on the execution time of a call to the Clock function, in processor clock cycles. Line tested: " T := Clock; "

```
=== D8-44-clock_call ====================================
total processor cycles used by a 'Clock' call: 126
=========================================================
```

45. Upper bounds on the execution times of the operators of the types Time and Time_Span The results of the test give the following cycles for each operation:

- *Time + Time_Span:* Cycles: 18
- *Time_Span + Time:* Cycles: 18
- *Time - Time_Span:* Cycles: 18
- *Time - Time:* Cycles: 18
- *Time < Time:* Cycles: 16
- *Time <= Time:* Cycles: 24
- *Time > Time:* Cycles: 23
- *Time >= Time:* Cycles: 15
- *Time_Span + Time_Span:* Cycles: 18
- *Time_Span - Time_Span:* Cycles: 18
- - *Time_Span:* Cycles: 27
- *Time_Span * Integer* Cycles: 145
- *Integer * Time_Span* Cycles: 146
- *Time_Span / Time_Span* Cycles: 1603
- *Time_Span / Integer* Cycles: 277
- *abs(Time_Span)* Cycles: 16
- *Time_Span < Time_Span* Cycles: 39
- *Time_Span <=Time_Span* Cycles: 31
- *Time_Span > Time_Span* Cycles: 32

[1]This error margin in the test is because of the standard unix clock representation (1 second clock resolution)

- *Time_Span >=Time_Span* Cycles: 40
- *To_Duration(Time_Span)* Cycles: 13
- *To_Time_Span(Duration)* Cycles: 13
- *Split(Time, Seconds_Count, Time_Span)* Cycles: 716
- *Time_Of(Seconds_Count, Time_Span)* Cycles: 70
- *Nanoseconds(Integer)* Cycles: 135
- *Microseconds(Integer)* Cycles: 254
- *Milliseconds(Integer)* Cycles: 274

F.1.3 Delay Accuracy (D-9)

11. An upper bound on the execution time, in processor clock cycles, of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of Real_Time.Clock at the time of executing the statement. This execution time is equal to **475** processor clock cycles.

13. An upper bound on the lateness of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement

- *One task + background task*

  The delay until lateness upper bound for a call to a delay until statement is **139.5** μseconds, using a 20 MHz system clock. This lateness occurs when the time of the delay until matches with a second change. It must be noted that the clock interrupt occurs every second in the kernel tested.

  If the time of the delay until statement does not match with a clock interrupt, the lateness upper bound for a call to a delay until statement is **109.5** μseconds.

- *N tasks + background task*

  The delay until lateness for several tasks (N Tasks) is: **132.5** μseconds + **7** $\times$ N μseconds when the time of the delay until matches with a clock interrupt. If not: **102.5** μseconds + **7** $\times$ N μseconds

F.1.4 Other Optimizations and determinism rules (D-12)

7. The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. The execution time in processor clock cycles of a call to Set. The number of processor clock cycles used by a call to the protected procedure "Set" is **706**.

F.2 Other useful metrics

F.2.1 Context Switch

The context switch is measured between two tasks with the same priority, before this it measures the overhead of the Yield procedure.

The context switch time is equal to **523** processor clock cycles.

F.2.2 Interrupt Latency

This tests measures the interrupt latency. This test uses a special register of ERC32 which allow to force interrupts.

The external interrupt 2 is forced by this means and the time until the first statement of the protected interrupt handler is reached can be considered as the interrupt latency.

The interrupt latency is equal to **1708** processor clock cycles.

This elapsed time has been measured with the simulators otherwise special test instrument must be used. The example program detailed in appendix C has been used for the test.

Bibliography

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

[2] Ada Core Technologies. *GNAT User's Guide. Version 3.12a2*, 1999.

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

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

[5] Ada Core Technologies. *GNAT User's Guide. Version 3.13a*, March 2000.

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

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

[8] Lars Asplund, Bob Johnson, and Kristina Lundqvist. Session summary: The Ravenscar profi le and implementation issues. *Ada Letters*, XIX(25):12–14, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[9] 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.

[10] Ted Baker and Tullio Vardanega. Session summary: Tasking profi les. *Ada Letters*, XVII(5):5–7, 1997. Proceedings of the 8th International Ada Real-Time Workshop.

[11] Alan Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S.H. Son, editor, *Advances in Real-Time Systems*. Prentice-Hall, 1994.

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

[13] Alan Burns. The Ravenscar profi le. Technical report, University of York, 2000. Available at `www.cs.york.ac.uk/~burns/ravenscar.ps`.

[14] Alan Burns, Brian Dobbing, and George Romanski. The Ravenscar profi le for high integrity real-time programs. In Lars Asplund, editor, *Reliable Software Technologies — Ada-Europe'98*, number 1411 in LNCS. Springer-Verlag, 1998.

[15] Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2 edition, 1996.

[16] Alan Burns and Andy J. Wellings. Restricted tasking models. *ACM Ada Letters*, XVII(5):27–32, 1997. Proceedings of the 8th International Ada Real-Time Workshop.

[17] 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.

[18] Juan A. de la Puente, Juan Zamorano, José F. Ruiz, Ramón Fernández, and Rodrigo García. The design and implementation of the Open Ravenscar Kernel. *Ada Letters*, XXI(1), 2001.

[19] Brian Dobbing and George Romanski. The Ravenscar profi le: Experience report. *Ada Letters*, XIX(2):28–32, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[20] Briand Dobbing and Alan Burns. The Ravenscar profi le for high-integrity real-time programs. *Ada Letters*, XVIII(6):1–6, 1998. Proceedings of the ACM SIGAda International Conference — SIGAda'98.

[21] ECCS. *ECCS-E-40A Space Engineering — Software*, 1999. Available from ESA.

[22] ESA/ESTEC. *32 Bit Microprocessor and Computer System Development. MEC rev. A Device Specification*, 1997. Report MCD/SPC/0009/SE.

[23] Per Cederqvist et al. *Version Management with CVS*. Free Software Foundation, 1993. For CVS version 1.10.5.

[24] Free Software Foundation. *GNU Emacs Manual*.

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

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

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] H. Shen, A. Charlet, and T.P. Baker. A ’bare-machine’ implementation of Ada multi-tasking beneath the Linux kernel. In Michael González-Harbour and Juan A. de la Puente, editors, *Reliable Software Technologies — Ada-Europe’99*, number 1622 in LNCS, pages 287–297. Springer-Verlag, 1999.

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

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

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

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

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

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

[40] Tullio Vardanega and Gert Caspersen. Using the Ravenscar Profi le for space applications: The OBOSS case. In Michael González-Harbour, editor, *Proceedings of the 10th International Real-Time Ada Workshop*, 2001. To appear in Ada Letters.

[41] W.M. Walker, P.T. Wooley, and A. Burns. An experimental testbed for embedded real time Ada 95. *Ada Letters*, XIX(2):84–89, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[42] Andy Wellings. 10th International Real-Time Ada Workshop — Session summary: Status and future of the Ravenscar profi le. *Ada Letters*, XXI(1), March 2001.

[43] Juan Zamorano, José F. Ruiz, and Juan A. de la Puente. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In Alfred Strohmeier and Dirk Craeynest, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 317–327. Springer-Verlag, 2001.

[44] Andreas Zeller. *Debugging with DDD. User’s Guide and Reference Manual*. Free Software Foundation, 1st edition, 2000. For DDD version3.2.