GNATforLEON / ORK+

User Manual

Version 1.3 — 30 July, 2009

For GNATforLEON 2.1.0/ ORK+

Universidad Politécnica de Madrid
Grupo de Sistemas de Tiempo Real y
Arquitectura de Servicios Telemáticos

Published by

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

This manual has been adapted to GNATforLEON from the *Operation Manual for the Open Ravenscar Real-Time Kernel* (ORK). The original software and its associated documentation were developed at UPM under ESA contract No.13863/99/NL/MV.

Status:	Final
Authors:	Juan A. de la Puente Juan Zamorano Jorge López

The original Operation Manual for ORK was written by Juan A. de la Puente, Juan Zamorano, José F. Ruiz, Ramón Fernández-Marina, and Miguel Ángel Ajo.

History

Version	**Date**	**Comments**
1.0	2007-07-18	First public release.
1.1	2008-11-18	Revised version for GNATforLEON 2.0.1.
1.2	2009-03-30	Revised version with some fixes.
1.3	2009-07-30	New version for GNATforLEON 2.1.0

Contents

Chapter 1

Introduction

1.1 Intended readership

This manual contains instructions for the use of the *Open Ravenscar real-time Kernel* (ORK+) and the GNATforLEON compilation toolchain with the LEON2 computer. It should be read by application programmers and system administrators of software projects using ORK+ and GNATforLEON.

Prior knowledge of the Ada (ISO/IEC, 2007) and C (ISO/IEC, 1999) programming languages is assumed. The reader should also be familiar with the GNAT and GCC programming environments (Ada Core, 2008a), the GDB debugger (Stallman and Pessch, 2007), and the fundamentals of real-time programming and the Ravenscar profile (ISO/IEC, 2005).

1.2 Purpose

The purpose of this document is to describe how to use, install, and configure GNATforLEON for LEON2-based computers.

GNATforLEON is an integrated package combining an instance of the GNAT compiler for the SPARC v8 architecture with ORK+, an open-source real-time kernel of reduced size and complexity, for which users can seek certification for mission-critical space applications. The kernel supports both Ada 2005 and C applications on a LEON2-based computer.

GNATforLEON is fully integrated with other GNAT-based tools and with GDB, the GNU debugger.

1.3 Applicability statement

This manual applies to the following versions of software:

- GNATforLEON 2.1.0/ORK+ 2.1.0
- GNAT GPL 2008
- GCC 4.1.3
- GDB 6.3

1.4 Compliance to standards

ORK+ supports the tasking model defined in the Ada 2005 standard (ISO/IEC 8652:1995/Amd 1:2007), including appendices C and D, except for the following:

1. The tasking model is restricted as specified by the Ravenscar profile (Ada Reference Manual, D13.1). Consequently,
   - Pragma `Priority_Specific_Dispatching` is not supported.
   - Only the `FIFO_Within_Priorities` dispatching policy with the `Ceiling_Locking` policy is supported.
2. The following exceptions to the Ravenscar profile are supported:
   - At most one statically declared execution-time timer per task is allowed.
   - Statically declared group budgets are allowed.

The ORK+ C interface provides a simplified thread interface which implements the Ravenscar computational model (RCM). It is not compliant with the POSIX-1 standard (IEEE Std1003.1, 2004 Edition), but the provided functionality is similar to the POSIX subset needed to implement the RCM.

1.5 Free software

GNATforLEON/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.

GNATforLEON/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 GNATforLEON or ORK+ that they might get from you. The precise conditions are found in the GNU General Public Licence (GPL) that comes with GNATforLEON and ORK+. See appendix D for details.

The easiest way to get a copy of GNATforLEON/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 download the latest version of GNATforLEON/ORK+ from the ORK+ web site:

```
http://www.dit.upm.es/ork/
```

1.6 Related documents

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

1. Ada 2005 Reference Manual (ISO/IEC, 2007).
2. Ada 2005 Rationale (Barnes, 2008).
3. GNAT User's Guide (Ada Core, 2008a).
4. GNAT Reference Manual (Ada Core, 2008b).
5. GNAT Programming Studio User Guide (Ada Core, 2007)
6. Debugging with GDB (Stallman and Pessch, 2007).

1.7 Problem reporting instructions

If you obtained GNATforLEON 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.dit.upm.es/ork/`

In any event, we also recommend that you send bug reports for GNATforLEON to:

`ork@dit.upm.es`

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 GNATforLEON 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, or GDB, rather than GNATforLEON or the ORK+ kernel, please send the bug report to the appropriate address:

- GNAT: `report@gnat.com`
- GCC: `bug-gcc@gnu.org`
- GDB: `bug-gdb@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 the 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.
BCC	Bare-C Cross Compilation System
CCS	Cross Compilation System
CIL	C Interface Library.
DSU	Debug Support Unit.
ESA	European Space Agency.
ESTEC	European Space Research and Technology Center.
FPU	Floating-Point Unit.
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.

GPS	GNAT Programming Studio.
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).
ORK	Open Real-Time Ravenscar Kernel.
OS	Operating System.
PC	IBM Personal Computer architecture.
PROM	Programmable Read-Only Memory.
RAVENSCAR	Reliable Ada Verifiable Executive Needed for Scheduling Critical Applications in Real-Time.
RP	Ravenscar Profile.
RP program	An Ada program that complies with the Ravenscar profile.
RTOS	Real-Time Operating System.
UPM	Universidad Politécnica de Madrid — Technical University of Madrid.
URL	Uniform Resource Locator.

1.9 References

1.9.1 Applicable documents

1. ECCS-E40B. Space Engineering — Software (ECSS, 2003).

2. Ada Reference Manual (ISO/IEC, 2007).

3. Guidance for the Use of the Ada Programming Language in High Integrity Systems (ISO/IEC, 2000).

4. Guide for the use of the Ada Ravenscar Profile in high integrity systems (ISO/IEC, 2005).

5. C Programming Language (ISO/IEC, 1999).

6. POSIX Standards (ISO/IEC, 2003).

1.9.2 Reference documents

1. LEON2 Manuals (Gaisler Research, 2005).
2. GRMON User's Manual (Gaisler Research, 2007).
3. Sparc 8 Architecture Manual (SPARC International, 1992).
4. Ada 2005 Rationale (Barnes, 2008)
5. Ada 95 — Quality and Style (Ausnit-Hood et al., 1995).
6. GNAT Manuals (Ada Core, 2008a,b).
7. Debugging with GDB (Stallman and Pessch, 2007).

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

1.10 History

ORK, the first version of the Open Ravenscar real-time Kernel, was developed under ESA/ESTEC Contract No.13863/99/NL/MV in the period 1999–2000, with subsequent updates up to 2003. It was targeted to ECR32 computers, based on a SPARC 7 architecture, and was integrated with a custom-crafted version of GNAT 3.13p.

In the period 2003–2005, a commercially supported version of the kernel was developed by UPM[1] and AdaCore under ESA/ESTEC Contract No. 17360/03/NL/JA. This version was integrated in the *GNAT Pro for ERC32* product, which is commercialized by AdaCore.

The current version of the kernel, now called ORK+, was developed in the framework of the ASSERT project.[2] It is targeted to LEON computers, and has been extended to support the current Ada 2005 standard (see section 1.4 above). ORK+ is the real-time kernel used in the ASSERT Virtual Machine, a specialized run-time platform that guarantees the specified real-time behaviour of Ravenscar programs.

1.11 Contributors

1.11.1 Contributors to ORK

ORK was developed by a team of the Department of Telematics Engineering, Universidad Politécnica de Madrid(DIT/UPM), led 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.

[1]Universidad Politécnica de Madrid (Technical University of Madrid).

[2]ASSERT (Automated proof-based System and Software Engineering for Real-Time systems) is an Integrated Project partially funded by the European Commission within the Information Society Technologies priority of the 6th Framework Programme in the area *embedded systems*. UPM's work in the project was also supported by the Spanish National R&D Plan.

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

The ORK software was validated by Jesús Borruel and Juan Carlos Morcuende, from Construcciones Aeronáuticas (CASA), Space Division.[3] We also relied very much on Andy Wellings and Alan Burns, of the University of York, UK, for reviewing and discussions about the Ravenscar profile and its implementation.

ORK was developed under contract with ESA. 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.

1.11.2 Contributors to ORK+

ORK+, the real-time kernel used in GNATforLEON, was developed by a team of the Department of Telematics Engineering, Universidad Politécnica de Madrid (DIT/UPM), lead by Juan Antonio de la Puente. The other members of the team were Juan Zamorano, José Pulido, Santiago Urueña, Jorge López, and José Redondo. The same team worked in collaboration with AdaCore to port GNAT to LEON2, and to develop the adapted runtime packages that enable GNAT to work with ORK+.

GNATforLEON was developed in the framework of the ASSERT and THREAD projects. It is based on AdaCore GNAT Pro ERC32 product which in turn was based on ORK.

1.12 Document overview

The rest of this document is organised as follows:

- Chapter 2 describes the general structure of the GNATforLEON software and the Ravenscar profile restrictions.
- Chapter 3 contains instructions for writing, compiling, linking, executing, and debugging programs with the GNATforLEON software.
- Chapter 4 describes in detail the functions of GNATforLEON and the way it is linked with Ada and C programs.
- Appendix A describes the Ravenscar profile in detail.
- Appendix B contains a comprehensive example of a Ravenscar-compliant program and its compilation with GNATforLEON.
- Appendix D contains a copy of the GNU General Public License (GPL).

[3]Currently integrated into EADS-Astrium.

Chapter 2

Software overview

2.1 The ORK+ kernel

ORK+ (Open Ravenscar Real-Time Kernel) is a small, high performance real-time kernel that provides restricted tasking support for Ada 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, that, if required, may even be subject to a certification process, program constructs which are not verifiable 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 verification 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* (ISO/IEC, 2000). Based on these considerations, language subsets for building software with different levels of integrity can be defined. 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 identifiers that are defined in the Ada "Safety and Security" annex (ISO/IEC, 2007, Annex H).

Tasking has often been considered not safe for high integrity systems, mainly due to the difficulty of analysing and verifying tasking programs. However, results in response time analysis for fixed priority preemptive scheduling (Burns, 1994) 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 (IRTAW 8), which was held in 1997 in Ravenscar, Yorkshire, England, was to define a safe tasking model for Ada. The outcome of this work is known as the "Ravenscar profile" (Baker and Vardanega, 1997). The profile was slightly modified in the following IRTAW meetings, after some experience was gained on its implementation and use. It was later included in the Ada High-Integrity Systems report (ISO/IEC, 2000), and it finally made its way into the current Ada 2005 standard (ISO/IEC, 2007, Annex D).

The profile defines a subset of 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 defined in the Ravenscar profile 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 GNATforLEON distribution. Real-time programs are written in a subset of Ada which is consistent with the Ravenscar profile 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 GNATforLEON 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 the GDB version that comes with the GNATforLEON distribution. GNATforLEON can be used from the GNAT Programming Studio (GPS).

The kernel has been designed for efficient support of Ada tasking constructs, but can also be used with C programs. A C interface package is provided for this purpose (see section 3.7 on page 25).

2.2 Architecture of ORK+

The kernel consists of the following Ada packages (figure 2.1:

- System.BB:[1] Root package (empty interface).
- System.BB.Threads: Thread management, including synchronization and scheduling control functions.
- System.BB.Threads.Queues: Different kernel queues such as alarm queue and ready queue.
- System.BB.Time: Clock and delay services including support for timing events.
- System.BB.Time.Execution_Time_Support: Execution time clocks and timers as well as group budgets support.
- System.BB.Interrupts: Interrupt handling.
- System.BB.Parameters: Configuration parameters.
- System.BB.CPU_Primitives: Processor-dependent definitions and operations.
- System.BB.Peripherals: Support for peripherals in the target board.
- System.BB.Peripherals. Registers : Definitions related to input-output registers of the peripheral devices.
- System.BB.Serial_Output: Support for serial output to a console.

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 tasking constructs can be directly used by the real-time application programmer. This interface is described in the next section.

[1] "BB" stands for "bare-board kernel".

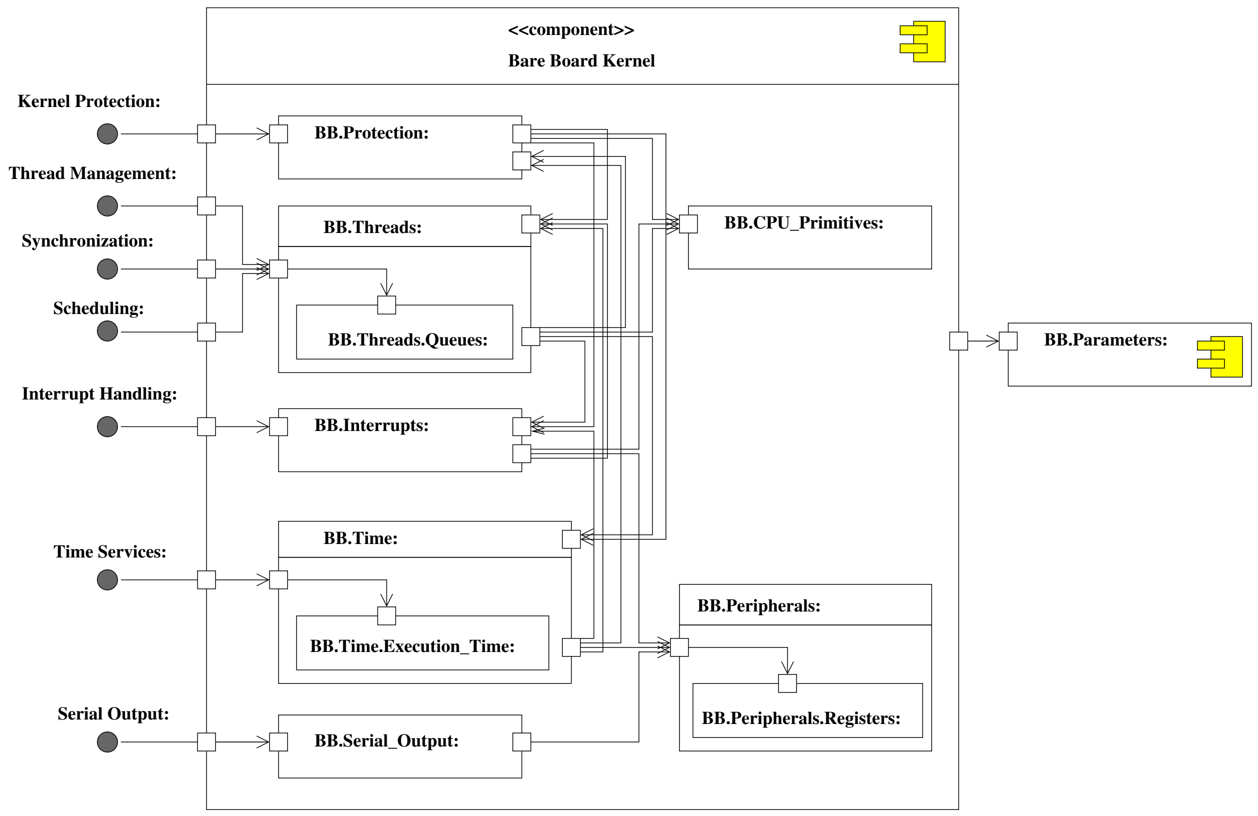

Figure 2.1: Architecture of ORK+

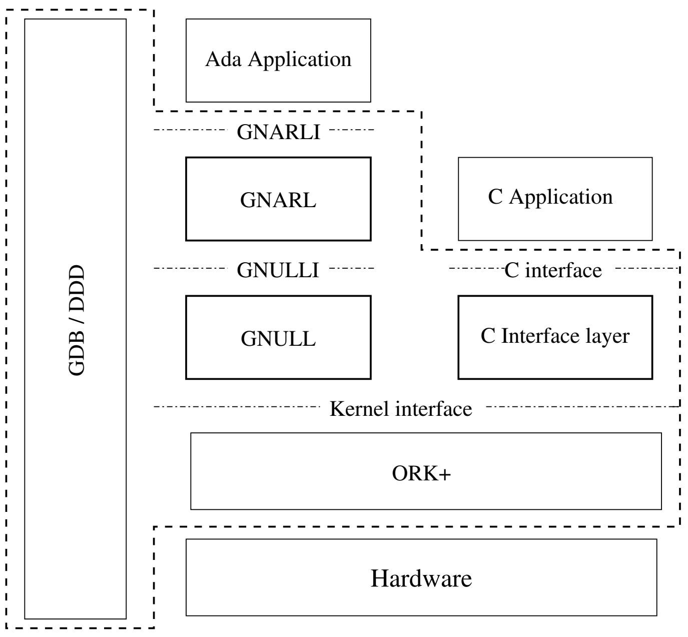

Figure 2.2: Architecture of the GNAT/GCC run-time system based on ORK+

2.3 Interface for Ada programs

Ada tasking is implemented in GNAT by means of the run-time library, called GNARL (Giering and Baker, 1994). 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. The GNATforLEON instance of GNULL is built on top of the bare-board kernel (i.e. ORK+), which provides all the low-level tasking support functionality required by the Ravenscar profile subset of Ada tasking (figure 2.2).

Ravenscar-compliance of Ada programs is enforced by means of the configuration pragma Profile (Ravenscar). GNATforLEON uses a restricted version of GNARL with reduced size and complexity which was developed by AdaCore with certification in mind and just to support the Ravenscar profile. A special version of GNULL is also used, which interfaces directly with the kernel. All this complexity is hidden to the programmer, who only needs to insert the pragma Profile (Ravenscar) in the GNAT configuration file (usually named `gnat.adc`).

2.4 Interface for 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 figure 2.2) is provided for this purpose. The C interface is implemented in a C interface layer (CIL), which is integrated with the GCC compilation system.

2.5 Hardware and software environment

The GNATforLEON kernel is intended to be used with a GNAT or GCC compilation system targeted to a LEON2 computer. In order to use it effectively, the following components are required.

2.5.1 Development platform

Real-time software based on ORK+ can be developed on a PC-compatible system with a GNU/Linux operating system. The software has been tested with the Ubuntu 6.0.6 distribution of GNU/Linux.

The cross development system consists of the following packages targeted to ELF-32 LEON2:

- GNAT GPL 2008: GNU Ada 2005 compilation system.
- GCC 4.1.3: GNU C compiler.
- GDB 6.3: GNU debugger.
- newlib 1.14: Cygnus C-library.
- binutils 2.16.1: GNU binary utilities.

- ORK+ 2.1.0: ORK+ kernel, with C Interface Library and GNAT patches.

You can download all the above mentioned packages from `http://www.dit.upm.es/ork/`. Section 4.1 shows how to install a GNATforLEON compilation system from both the binary and source distributions.

You also may find it useful to have the GNAT Programming Studio (GPS) (`https://libre.adacore.com/gps/`) for developing applications with GNATforLEON.

2.5.2 Execution platform

The execution platform for GNATforLEON based programs is a LEON2 computer with at least 500 KB memory. Programs can be loaded into an LEON2-based computer memory by means of appropriate tools, e.g. GRMON or `mkprom`. 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 using the TSIM simulator, which can be connected to GDB using a socket connection.[2]

[2]TSIM is not free software and it is not part of GNATforLEON. See `http://www.gaisler.com/` for further details.

Chapter 3

How to use GNATforLEON

3.1 Software development

In order to develop programs based on GNATforLEON/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.

Two kinds of programs are supported by GNATforLEON and ORK+:

- Ada programs, restricted as defined by the Ravenscar profile.
- C programs directly using kernel threads.

Figure 3.1 describes the data flow for the GNATforLEON compilation system. Notice that purely sequential Ada or C programs do not require ORK+ support, and can be compiled using a bare-board version of GNAT[1] or GCC.

3.2 Writing Ada programs

The first step in compiling an Ada application is to write the source code for the program units that make up the application. You can use your favourite text editor for this purpose or use an IDE such as GPS, the GNAT Programming Studio.

3.2.1 Ravenscar profile restrictions

When writing your Ada application code, you should bear in mind that only the Ada subset defined by the Ravenscar profile 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 profile):

[1]For details about the different version of GNAT contact Ada Core at `http://www.adacore.com`.

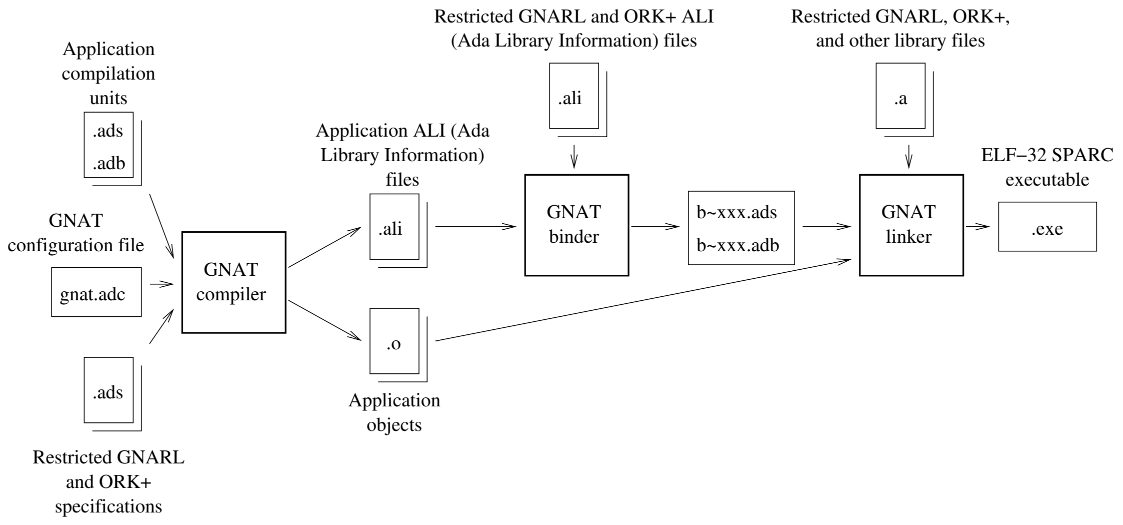
Application compilation units
.ads
.adb
GNAT configuration file
gnat.adc
.ads
Restricted GNARL and ORK+ specifications
GNAT compiler
Application ALI (Ada Library Information) files
.ali
.o
Application objects
Restricted GNARL and ORK+ ALI (Ada Library Information) files
.ali
GNAT binder
b~xxx.ads
b~xxx.adb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

Figure 3.1: Compilation flow for GNATforLEON/ORK+ applications

- 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.
- Execution-time timers and group budgets.[2]
- Non-local timing events.
- 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.

[2]As an exception to the Ravenscar profile, ORK+ supports one execution-time timer per task and group budgets.

- User-defined 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 ALRM, ISO/IEC, 2000 for guidelines on the Ada features you may wish to limit).

GNATforLEON 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 GNATforLEON 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 *GNATforLEON/ORK+ users are recommended to assign distinct priorities to all tasks and protected objects.*[3]

3.2.2 The GNAT configuration file

Configuration pragmas are put in a special source file called `gnat.adc` (see Ada Core, 2008a). You can have GNAT check all the above restrictions for you by compiling the program with a Ravenscar configuration pragma, as shown in the following template:

Listing 3.1: Sample gnat.adc file

```
-- gnat.adc - minimum configuration file template for the Ravenscar profile
pragma Profile (Ravenscar);

-- Any other configuration pragma can be included here
```

A copy of this file is included in the `examples` directory for your convenience.[4] 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 4.5 for the details).

[3]GNATforLEON/ORK+ allows priorities to be shared —as long as it is compatible with the *ceiling locking* policy—, but this is not a commendable practice unless the task and protected object population exceeds the allowable range of priorities. See section 4.5 on how to configure the range of priorities and the maximum number of tasks.

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

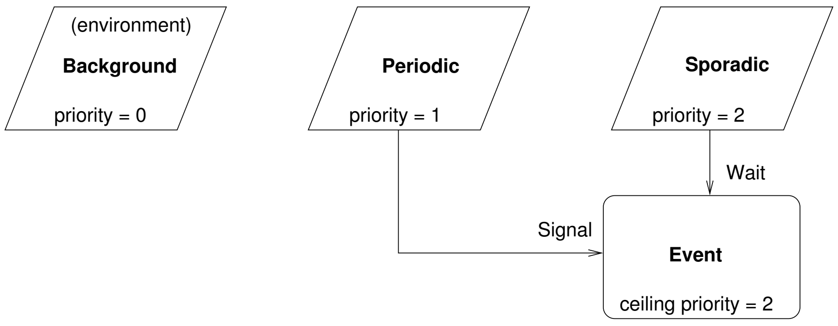

Figure 3.2: Task structure of the `hello` program (see 3.2.3). Parallelograms represent tasks, and rounded rectangles represent protected objects. The arrows denote procedure or entry calls.

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*, Ada Core, 2008a for the details). Figure 3.2 shows the task structure of the program.

The main procedure code is provided in listing 3.2.

Listing 3.2: Main procedure of the `hello` 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 when the pragma Profile (Ravenscar) is in place. 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`, see listing 3.3).

Listing 3.3: Specification of the `Tasking` package.

```
-- 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 (listing 3.4).

Listing 3.4: Body of the `Tasking` package.

```
-- tasking.adb - application tasks for the 'hello' example
with Ada.Real_Time;
-- used for Clock, Time_Span, Milliseconds
with System.BB.Serial_Output; use System.BB.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 (Burns and Wellings, 2001). 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 programs

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

```
$ sparc-elf-gnatmake hello
```

The command line switches are described in the *GNAT User's Guide* (Ada Core, 2008a).

You can also compile, bind, and link separately:

```
$ sparc-elf-gcc -c hello.adb
$ sparc-elf-gcc -c tasking.adb
$ sparc-elf-gnatbind -x hello.ali
$ sparc-elf-gnatlink hello.ali
```

See Ada Core (2008a) 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-elf-gnatmake -g hello -largs -Wl,-Map=hello.map
```

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 programs on the development platform

The simplest way you can test your program is by using a LEON2 simulator on your development platform. If you have the TSIM simulator[5] you can do:

```
$ tsim-leon 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
...
```

Running the program on TSIM by itself 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 LEON2 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-elf-gdb hello
(gdb) target extended-remote 127.0.0.1:1234
     ...
(gdb) load
      ...
(gdb) cont
       ...
(gdb) detach
      ...
```

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

For a complete description of GDB commands, see the document *Debugging with GDB* (Stallman and Pessch, 2007). You can have a better view with the graphical source-level front-end to GDB of the GNAT Programming Studio (GPS).

3.5 Executing and debugging Ada programs on the target platform

The simplest way you can test your program on the target platform is by using the GRMON debug monitor for LEON processors.[6]

GRMON comunicates with the LEON debug support unit (DSU) and enables non-intrusive debugging of the complete target system. It is started by entering the `grmon` command in a terminal window. By default, GRMON communicates with the target using the first UART port of the host. This can be overridden by specifying an alternative device. Device names depend on the host operating system. On Unix systems, serial devices are named as `/dev/ttyXX`.

Use the `-baud` option if you need to use a different baud rate for the DSU serial link than the default 115200 baud.

The example below shows how to execute programs with GRMON using an usual list of start-up switches:

```
$ grmon -dsu -uart /dev/ttyS0 -baud 115200
```

Once you get the GRMON console use the `load` command to download the application and then `go` to start it.

The output from the application appears on the normal LEON UARTs and thus cannot be seen on the GRMON console unless the program is started with the `-u` switch. You can use terminal emulators such as `tip`, `minicom`, or `kermit` to display the output.

Warning 3.2 *When an application terminates, it must be reloaded on the target before it can be re-executed, in order to re-initialise the data segment* (`.data`).

For a complete description of the GRMON debug monitor, see the *GRMON User's Manual* (Gaisler Research, 2007).

3.5.1 Floating-point considerations

If the targeted LEON processor has no floating-point hardware, then all applications must be compiled (and linked) with the `-msoft-float` option in order to enable floating-point emulation. When running the program on the TSIM simulator, the simulator should be started with the `-nfp` option (no floating-point) in order to disable the FPU.

[6]GRMON is not free software, and it is not part of GNATforLEON. See `http://www.gaisler.com/`for futher details.

3.5.2 Making boot PROMS

GNATforLEON applications are linked to run from the beginning of the RAM area at address 0x40000000. To make a boot-PROM that can 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 LEON registers, 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 re-linked 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 41 MHz system clock. For more details see the `mkprom` manual.

```
$ mkprom -ramsz 1024 -ramwws 1 -romws 3 hello -freq 41 hello
```

To create a Motorola S-record file for a PROM programmer, use `objcopy`:

```
$ sparc-elf-objcopy -O srec hello hello.srec
```

A file called `hello.srec` is created. It is possible to use the TSIM simulator to load and test the resulting file.

3.5.3 Using GDB with GRMON and the remote target monitor

It is possible to use GRMON as the remote target, and connect it with `sparc-elf-gdb`, enabling symbolic debugging of target applications. To initiate GDB communications, start the monitor with the `-gdb` switch.

```
$ grmon -dsu -uart /dev/ttyS0 -baud 115200 -gdb
```

Now, the debugging session can be started using the extended-remote protocol. By default GRMON listens on port 2222 for the GDB connection:

```
$ sparc-elf-gdb hello
(gdb) target extended-remote localhost:2222
(gdb) load
(gdb) continue
      ...
```

While attached, normal GRMON commands can be executed using the gdb monitor command. Output from the GRMON commands, such as the trace buffer history is then displayed on the GDB console.

3.6 Interrupt handlers

3.6.1 Protected procedure handlers

The Ravenscar profile 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 identified in the Ada.Interrupts.Names package. This package contains the identifiers of all the LEON2 interrupts.

A general template is shown in listing 3.5.

Listing 3.5: Template for interrupt handlers.

```
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 ceiling priority to the protected object with a pragma Interrupt_Priority . You should use a priority level equal or greater than the hardware priority of the interrupt source. Notice that the corresponding hardware priority levels have also been declared in the package Ada.Interrupt.Names.

Warning 3.3 *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 B describes an example application with interrupt handlers.

3.7 Working with C programs

3.7.1 The C interface

ORK+ can be used from C programs through the CIL (C Interface Layer, see figure 2.2 on page 11). 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:

- `gnatforleon.h`
- `system-bb-interrupts.h`
- `system-bb-serial-output.h`
- `system-bb-threads.h`

- `system-bb-time.h`
- `types.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 assurance of conformance of the application with the Ravenscar profile is left as a responsibility 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 has the following general structure:

  Listing 3.6: Structure of a main C program.

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

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

  Tasks must not terminate either, as required by the Ravenscar profile. Therefore, the structure of task bodies will be as follows:

  Listing 3.7: Structure of C thread code.

  ```
  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.

- ORK+ needs to be initialized before functions exported to C can be used. This means that the function `system_bb_initialize()` must be called after `adainit()`.
- No dynamic memory can be used, which means that 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. If a POSIX interface were used, the start event of the sporadic task would be implemented by means of a condition variable, corresponding to the protected entry in the Ada version.However, ORK+ does not support POSIX-like condition variables, but a lower level synchronization mechanism (unconditional sleep and wakeup) is provided instead. The example shows how to use this mechanism to implement sporadic activation according to the Ravenscar Computational Model. The expected output is the same as in section 3.4

Listing 3.8: "Hello" program in C.

```c
#include "gnatforleon.h"

system_bb_thread_t periodic_task;
system_bb_thread_t sporadic_task;
volatile system_bb_mutex_t mutex;

int periodic (void) {

        int c = 3;
        int previous_priority = 0;
        system_bb_time_t ticks_ps = system_bb_ticks_per_second();
        system_bb_time_t period = 0.5*ticks_ps; //500ms
        system_bb_time_t next = system_bb_clock();

        while (1) {
                system_bb_delay_until(next);
                c--;

                printf("Hello␣periodic\n");

                if (c == 0) {

                        previous_priority = system_bb_get_priority(&periodic_task);

                        // pthread_mutex_lock
                        system_bb_set_priority(mutex.priority);

                        // It is needed to know what is the awaiting task
                        // pthread_condition_signal
                        mutex.barrier = true;
                        system_bb_wakeup(&sporadic_task);

                        // pthread_mutex_unlock
                        system_bb_set_priority(previous_priority);
                        c = 10;

                }
                next = next + period;
        }
}

int sporadic (void) {

        int previous_priority = 0;

        while (1) {

                // pthread_mutex_lock
                previous_priority = system_bb_get_priority(&sporadic_task);
                system_bb_set_priority(mutex.priority);

                // pthread_condition_wait
                if (mutex.barrier == false) system_bb_sleep();
                mutex.barrier = false;

                printf("Hello␣sporadic\n");

                // pthread_mutex_unlock
                system_bb_set_priority(previous_priority);

        }
}

int main (void) {
        int count = 0;
        system_bb_prio_t p_periodic;
        system_bb_prio_t p_sporadic;
        int stacksize = 20480;
        system_bb_addr_t stackaddress = 0x400296C0;
        system_bb_addr_t stackaddress2 = stackaddress - (stacksize+600);
        //...
        //system_bb_addr_t stackaddress3 = stackaddress2 - (stacksize+600);
        //system_bb_addr_t stackaddress4 = stackaddress3 - (stacksize+600);

        adainit();
        system_bb_initialize();

        p_periodic = 199;
        p_sporadic = 200;
        mutex.priority = 200;
        mutex.barrier = false;
        system_bb_thread_create(&periodic_task, periodic, 0, p_periodic,
                                stackaddress, stacksize);
        system_bb_thread_create(&sporadic_task, sporadic, 0, p_sporadic,
                                stackaddress2, stacksize);
        while (1) {
                if (count++ > 1000) count = 0;
                system_bb_delay_until (system_bb_clock()+1000000000);
        }
        adafinal();
}
```

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

3.7.4 Compiling a C program

The compiler has to 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 GNATforLEON packages has to be added to the program.

The code of this procedure is shown on listing 3.9.

Listing 3.9: Specification of the `stub` procedure for C programs.

```
with System.BB.Threads;
with System.BB.Time;
with System.BB.Interrupts;
with System.BB.Serial_Output;

procedure Stub is
begin
   null;
end Stub;
```

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

1. Compile your C code:

   ```
   $ sparc-elf-gcc -I/usr/local/gnatforleon/include -g -c c-program.c
   ```

2. Compile the Ada code:

   ```
   $ sparc-elf-cc -c stub.adb
   ```

3. Bind the Ada program:

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

4. Link everything together:

   ```
   $ sparc-elf-gnatlink stub.ali c-program.o -lgnarl -o c-program
   ```

Chapter 4

GNATforLEON/ORK+ reference

4.1 Installation and directory structure

4.1.1 Getting GNATforLEON and ORK+

GNATforLEON/ORK+ is distributed via:

`http://www.dit.upm.es/ork/`

The sources used to build the GNATforLEON cross-compilation system can be also found at the same location. The GNATforLEON distribution includes:

`gnatforleon-2.1.0-i686-pc-linux-gnu-bin.tar.gz` : gzipped tarfile which contains the binary distribution for GNU/Linux. The current distribution has been built on Ubuntu 6.0.6 using `glibc2` libraries. However, it runs on most modern Linux distributions. In order to avoid problems with different versions of `libc`, all binaries are statically linked.

`gnatforleon-2.1.0-src.tar.gz` : gzipped tarfile which contains the sources as well as the procedures for building the GNATforLEON.

`examples-2.1.0.tgz` : gzipped tarfile which contains some examples of the GNATforLEON functionality. The Ada and C version of the example described in section 3.2.3 as well as the demo application described in appendix B are also included.

4.1.2 Installing GNATforLEON

The GNAT directory tree has been compiled to reside in any directory. After obtaining the gzipped tarfile `gnatforleon-2.1.0-i686-pc-linux-gnu-bin.tar.gz`, which includes the binary distribution, uncompress and untar it in the desired location.

The GNATforLEON distribution can be installed with the following commands (assuming the gzipped tar file is in directory `/tmp`):

```
$ tar -zxvf gnatforleon-2.1.0-i686-pc-linux-gnu-bin.tar.gz -C \\
      desired_location
```

After the cross-compilation system is installed, the directory `desired_location/gnatforleon-2.1-elf-bin/bin/` must be added to the search path (usually, environment variable `PATH` in your shell).

4.1.3 Installing the GNATforLEON sources

In the following, the installation directory for source files is assumed to be `/usr/local/gnatforleon/src`, although they can be installed at any other location as well. After obtaining the gzipped tarfile `gnatforleon-2.1.0-src.tar.gz`, which contains the sources of the GNATforLEON cross-compilation system, uncompress and untar it to `/usr/local/gnatforleon/src`.

The GNATforLEON distribution can be installed with the following commands (assuming the gzipped tar file is in `/tmp/gnatforleon-2.1.0-src.tar.gz`):

```
$ cd /usr/local
$ tar -zxvf /tmp/gnatforleon-2.1.0-src.tar.gz
```

The sources have been adapted using AdaCore patches, LEON Bare-C Cross Compilation System (BCC) patches, and specific GNATforLEON patches. These sources are ready to build GNATforLEON CCS. The source distribution contains procedures (`Makefile`) for building the whole GNATforLEON CCS and the GNATforLEON `adalib` (see sections 4.5 and 4.5.3).

4.1.4 Directory structure

Contents of `/usr/local/gnatforleon`

- `bin`: executables.
- `include`: include files for the C interface.
- `info`: gcc documentation in info format.
- `lib`: gcc libraries which include GNATforLEON adalib for LEON2 target.
- `libexec`: gcc libraries.
- `man`: man pages.
- `sparc-elf`: newlib (libc) library for SPARC family.

Contents of `/usr/local/gnatforleon/src`

- `binutils-2.16.1`: Adapted sources of binutils for GNATforLEON.
- `newlib-1.14.0`: Adapted sources of newlib for GNATforLEON.
- `gcc-4.1.3`: Adapted sources of gcc for GNATforLEON.
- `gcc-4.1.3/gcc/ada`: Adapted sources of GNAT GPL 2007 for GNATforLEON including GNATforLEON itself.
- `gdb-6.3`: Adapted sources of gdb for GNATforLEON.
- `c_interface`: include files for the C interface.

4.1.5 Tools

GNATforLEON includes the following tools in the `/usr/local/gnatforleon/bin` directory:

- `sparc-elf-addr2line`: utility to translate program addresses into file names and line numbers.
- `sparc-elf-ar`: library archiver.
- `sparc-elf-as`: cross-assembler.
- `sparc-elf-c++filt`: utility to demangle C++ symbols.
- `sparc-elf-gcc`: C cross-compiler.
- `sparc-elf-gccbug`: a tool for reporting GCC Bugs.
- `sparc-elf-gcov`: coverage testing tool.
- `sparc-elf-gdb`: the GNU Debugger.
- `sparc-elf-gnat`: utility to list GNAT commands, qualifiers and options.
- `sparc-elf-gnatbind`: Ada binder.
- `sparc-elf-gnatchop`: Ada source code splitter.
- `sparc-elf-gnatfind`: Ada utility for locating definitions and/or references to a specified entity or entities.
- `sparc-elf-gnatkr`: Ada file name kruncher.
- `sparc-elf-gnatlink`: Ada linker.
- `sparc-elf-gnatls`: Ada library lister.
- `sparc-elf-gnatmake`: Ada make utility.
- `sparc-elf-gnatprep`: Ada pre-processor.
- `sparc-elf-gnatxref`: Ada utility to generating a full report of all cross-references.
- `sparc-elf-ld`: linker.
- `sparc-elf-nm`: utility to print symbol table.
- `sparc-elf-objcopy`: utility to convert between binary formats.
- `sparc-elf-objdump`: utility to dump various parts of executables.
- `sparc-elf-ranlib`: library sorter.
- `sparc-elf-size`: utility to display segment sizes.
- `sparc-elf-strings`: utility to dump strings from executables.
- `sparc-elf-strip`: utility to remove symbol table.

4.1.6 Documentation

Extensive documentation for all the tools can be found in the the `/usr/local/gnatforleon/info` and `/usr/local/gnatforleon/man` directories.

Documentation for the LEON2 processor can be found at the Aeroflex Gaisler site located at `http://www.gaisler.com`.

4.2 Kernel interface

4.2.1 Introduction

The ORK+ kernel provides all the required functionality to support real-time programming on top of the LEON2 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. 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 System.BB.Threads :

Listing 4.1: Specification of System.BB.Threads.

```
-- Package that implements basic tasking functionalities

pragma Restrictions (No_Elaboration_Code);

with System;
-- Used for Address
--          Null_Address
--          Any_Priority

with System.Parameters;
-- Used for Size_Type

with System.BB.CPU_Primitives;
-- Used for Context_Buffer
```

```
with System.BB.Time;
-- Used for Time

with System.BB.Interrupts;
-- Used for Interrupt_Set
--          Empty_Interrupt_Set

package System.BB.Threads is
   pragma Preelaborate;

   ---------------------------
   -- Basic thread support --
   ---------------------------

   type Thread_Descriptor;
   -- This type contains the information about a thread

   type Thread_Id is access all Thread_Descriptor;
   -- Type used as thread identifier

   Null_Thread_Id : constant Thread_Id := null;
   pragma Export (C, Null_Thread_Id, "system_bb_null_thread_id");
   -- Identifier used to define an invalid value for a thread identifier

   type Thread_States is (Runnable, Suspended, Delayed);
   -- These are the three possible states for a thread under the Ravenscar
   -- profile restrictions: Runnable (not blocked, and it may also be
   -- executing), Suspended (waiting on an entry call), and Delayed (waiting
   -- on a delay until statement).

   type Exec_Handler is access procedure (I : Integer);

   type Thread_Descriptor is record
      Context : aliased System.BB.CPU_Primitives.Context_Buffer;
      -- Location where the hardware registers (stack pointer, program
      -- counter, ...) are stored. This field supports context switches among
      -- threads.

      ATCB : System.Address;
      -- Address of the Ada Task Control Block corresponding to the Ada task
      -- that executes on this thread.

      Base_Priority : System.Any_Priority;
      -- Base priority of the thread

      Active_Priority : System.Any_Priority;
      pragma Volatile (Active_Priority);
      -- Active priority that differs from the base priority due to dynamic
      -- priority changes required by the Ceiling Priority Protocol. This
      -- field is marked as Volatile for a fast implementation of
      -- Get_Priority .

      Top_Of_Stack : System.Address;
      -- Address of the top of the stack that is used by the thread

      Bottom_Of_Stack : System.Address;
      -- Address of the bottom of the stack that is used by the thread

      Next : Thread_Id;
      -- Points to the ready thread that is in the next position for
      -- execution.

      Alarm_Time : System.BB.Time.Time;
      -- Time (absolute) when the alarm for this thread expires

      Next_Alarm : Thread_Id;
      -- Next thread in the alarm queue. The queue is ordered by expiration
      -- times. The first place is occupied by the thread which must be
      -- first awaken.

      State : Thread_States;
      -- Encodes some basic information about the state of a thread

      Wakeup_Signaled : Boolean;
      -- Variable which reflects whether another thread has performed a
      -- Wakeup operation on the thread.

      Time_Init_Execution : System.BB.Time.Time;
      -- Time when task has received the CPU
      Execution_Time      : System.BB.Time.Time;
      -- Execution Time of the task
      Time_Remaining      : System.BB.Time.Time;
      -- Time remaining of timer associated to the task
      Is_Timer_Alarm      : Boolean;
      -- Flag that indicates if it has been put an alarm associated to the
      -- timer associated to the task
      Is_GB_Alarm         : Boolean;
      -- Flag that indicates if it has been put an alarm associated to the
      -- timer associated to the group of the task
      TM_Integer          : Integer ;
      -- Index to the array with the handlers associated to the timers
      Time_Diff           : System.BB.Time.Time;
      -- Difference of time between time of the init of the execution of the
      -- task and the asignation of the timer to the task
      Time_Diff_GB        : System.BB.Time.Time;
      -- Difference of time between time of the init of the execution of the
```

```
   -- task and the asignation of the timer to the group of the task
   Handler              : Exec_Handler;
   -- Handler associated to the Execution_Time Timer of the task
   GB_Id                : Integer;
   -- Identifier associated to the Group_Budget of the task
   GB_Index             : Integer;
   -- Place of the array associated to the Group_Budget where the task
   -- data is stored
   Handler_GB           : Exec_Handler;
   -- Handler associated to the Group_Budget of the task
end record;

for Thread_Descriptor use
   record
      Context at 0 range 0 ..
        (System.BB.CPU_Primitives.Context_Buffer_Size - 1);
   end record;
-- It is important that the Context field is placed at the beginning of
-- the record, because this assumption is using for implementing context
-- switching.

procedure Initialize
  (Environment_Thread : Thread_Id;
   Main_Priority      : System.Any_Priority);
-- Procedure to initialize the board and the data structures related to
-- the low level tasking system. This procedure must be called before any
-- other tasking operation.

procedure Thread_Create
  (Id            : Thread_Id;
   Code          : System.Address;
   Arg           : System.Address;
   Priority      : System.Any_Priority;
   Stack_Address : System.Address;
   Stack_Size    : System.Parameters.Size_Type);
pragma Export (C, Thread_Create, "system_bb_thread_create");
-- Create a new thread
--
-- The new thread executes the code at address Code and using Args
-- as argument. Priority is the base priority of the new
-- thread. The new thread is provided with a stack of size
-- Stack_Size that has been preallocated at Stack_Address.
--
-- A procedure to destroy threads is not available because that is not
-- allowed by the Ravenscar profile.

function Thread_Self return Thread_Id;
pragma Inline (Thread_Self);
pragma Export (C, Thread_Self, "system_bb_thread_self");
-- Return the thread identifier of the calling thread

----------------
-- Scheduling --
----------------

procedure Set_Priority (Priority : System.Any_Priority);
pragma Inline (Set_Priority);
pragma Export (C, Set_Priority, "system_bb_set_priority");
-- Set the active priority of the executing thread to the given value

function Get_Priority (Id : Thread_Id) return System.Any_Priority;
pragma Inline (Get_Priority);
pragma Export (C, Get_Priority, "system_bb_get_priority");
-- Get the current active priority of any thread

procedure Sleep;
pragma Export (C, Sleep, "system_bb_sleep");
-- The calling thread is unconditionally suspended

procedure Wakeup (Id : Thread_Id);
pragma Export (C, Wakeup, "system_bb_wakeup");
-- Thread Id becomes ready (the thread must be previously suspended)

----------
-- ATCB --
----------

procedure Set_ATCB (ATCB : System.Address);
pragma Inline (Set_ATCB);
-- This procedure sets the ATCB passed as argument for the
-- currently running thread.

function Get_ATCB return System.Address;
pragma Inline (Get_ATCB);
-- Returns the ATCB of the currently executing thread

---------------------------------
-- Execution_Time functions ---
---------------------------------
type Handler is access protected procedure (I : Integer);

Global_TM_Pointer : Integer := 1;

Global_GB_Pointer : Integer := 1;
Budget_Array : array (1 .. 255) of System.BB.Time.Time_Span;
```

```
   function Get_Timer_Id  return Integer;
   function Get_GB_Id return Integer;

   ---------------------------
   -- Timing_Events types  --
   ---------------------------

   Global_TE_Pointer : Integer := 1;
   function Get_TE_Id return Integer;

   type TE_Alarm_Queue;
   -- This type contains the time of alarms relative of Timing Events and
   -- the next alarm.

   type TE_Alarm_Queue_Id is access all TE_Alarm_Queue;
   -- Type used as TE_Alarm_Queue identifier

   Null_TE_Alarm_Queue_Id : constant TE_Alarm_Queue_Id := null;

   type TE_Alarm_Queue is record
      Previous_Alarm : TE_Alarm_Queue_Id  := Null_TE_Alarm_Queue_Id;
      Alarm_Time     : System.BB.Time.Time := System.BB.Time.Time'Last;
      TE_Id          : Integer             := 0;
      Next_Alarm     : TE_Alarm_Queue_Id  := Null_TE_Alarm_Queue_Id;
   end record;

end System.BB.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 thread 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 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 synchronization of threads is usually achieved using condition variables. However, the Ravenscar Profile disallows complex synchronization patterns and two simple procedures are enough for supporting that simpler synchronization pattern. The procedure Sleep unconditionally suspends the current thread. As a result, the currently executing thread will leave the CPU. The procedure Wakeup makes a previously suspended thread become ready. The thread will be inserted at the tail of its active priority so that the thread will resume execution.

Scheduling of threads is performed according to the FIFO_Within_Priorities and Ceiling_Locking policies (see ALRM D.2-3).

4.2.3 Time management

The operations related with time are implemented in package System.BB.Time.

Listing 4.2: Specification of System.BB.Time.

```
-- Package in charge of implementing clock and timer functionalities

pragma Restrictions (No_Elaboration_Code);

with System.BB.Peripherals;
-- Used for Clock_Freq_Hz

package System.BB.Time is
   pragma Preelaborate;

   type Time is mod 2 ** 64;
   for Time'Size use 64;
   -- Time is represented at this level as a 64-bit natural number. The
   -- upper 40 bits representing the number of clock periods (MSP), and the
   -- lower 24 bits containing the number of hardware clock ticks (LSP).

   type Time_Span is range -2 ** 63 .. 2 ** 63 - 1;
   for Time_Span'Size use 64;
   -- Time_Span represents the length of time intervals , and it is
   -- defined as a 64-bit signed integer .

   ---------------
   -- Constants --
   ---------------

   Tick : constant := 1;
   -- A clock tick is a real time interval during which the clock value (as
   -- observed by calling the Clock function) remains constant. Tick is the
   -- average length of such intervals .

   Ticks_Per_Second : constant := System.BB.Peripherals.Timer_Freq_Hz;
   -- Number of ticks (or clock interrupts ) per second

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

   procedure Initialize_Timers ;
   --  Initialize  this package (clock and alarm handlers ).  Must be called
   -- before any other functions .

   ----------------
   -- Operations --
   ----------------

   function Number_Of_Ticks_Per_Second return Time;
   pragma Export (C, Number_Of_Ticks_Per_Second, "system_bb_ticks_per_second");
   -- Get the number of ticks (or clock interrupts ) per second

   function Clock return Time;
   pragma Export (C, Clock, "system_bb_clock");
   -- Get the number of ticks elapsed since startup

   procedure Delay_Until (T : Time);
   pragma Export (C, Delay_Until, " system_bb_delay_until " );
   -- Suspend the calling thread until the absolute time specified by T

   function "+" (Left  : Time; Right : Time_Span) return Time;

   function Get_Pending_Alarm return Boolean;
   -- Returns Pending_Alarm variable

   procedure Turn_True_Pending_Alarm;
   -- Turns Pending_Alarm variable to true

   procedure Cancel_Alarm;
   -- Cancel any alarm set in hardware until next clock timer expires

   procedure Inmediate_Alarm (Now : in out System.BB.Time.Time);

end System.BB.Time;
```

Time is represented in the ORK+ kernel as a 64-bit integer number of ticks. A tick is a real time interval during which the clock value (as observed by calling the System.BB.Clock function) remains constant.

The kernel provides a high resolution clock with a low overhead for 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 LEON2 hardware provides two timers (apart from the special *Watchdog* timer) which can be programmed to be either of single-shot type or periodic type (ATMEL, 2005). 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 one 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 register. 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 current value of the real-time clock can be obtained calling function Clock. This function returns the number of ticks elapsed since system startup, providing a time zone independent, monotonically increasing, absolute time value.

The number of ticks (or clock interrupts) per second can be read from the constant called Ticks_Per_Second.

When a thread needs to be suspended 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.

The reader should notice that this arrangement also provides support for execution-time clocks and timers, group budgets, and timing events (Urueña et al., 2007).

4.2.4 Interrupt handling

Interrupt names and operations are declared in the package System.BB.Interrupts.

Listing 4.3: Specification of System.BB.Interrupts.

```
-- Package in charge of implementing the basic routines for interrupt
-- management.

pragma Restrictions (No_Elaboration_Code);

with System;
-- Used for Any_Priority

with System.BB.Parameters;
-- Used for Interrupt_Levels

with System.BB.Peripherals;
-- Used for Priority_Of_Interrupt

package System.BB.Interrupts is
   pragma Preelaborate;

   Max_Interrupt : constant := System.BB.Parameters.Interrupt_Levels;
   -- The interrupts are distinguished by its interrupt level

   subtype Interrupt_ID is Natural range 0 .. Max_Interrupt;
   -- Interrupt identifier

   No_Interrupt : constant Interrupt_ID := 0;
   -- Special value indicating no interrupt

   type Interrupt_Handler is access procedure (Id : Interrupt_ID );
   -- Prototype of procedures used as low level handlers

   procedure Initialize_Interrupts ;
   -- Initialize table containing the pointers to the different interrupt
   -- stacks. Should be called before any other subprograms in this package.

   procedure Attach_Handler
     (Handler : Interrupt_Handler ;
      Id      : Interrupt_ID );
   pragma Inline (Attach_Handler);
   pragma Export (C, Attach_Handler, " system_bb_attach_interrupt_handler " );
   -- Attach the procedure Handler as handler of the interrupt Id

   function Priority_Of_Interrupt
     ( Id : Interrupt_ID ) return System. Any_Priority
   renames
     System.BB.Peripherals. Priority_Of_Interrupt ;
   -- This function returns the software priority associated to the interrupt
   -- given as argument.

   function Current_Interrupt return Interrupt_ID ;
   pragma Inline ( Current_Interrupt );
   -- Function that returns the hardware interrupt currently being
   -- handled ( if any). In case no hardware interrupt is being handled
   -- the returned value is No_Interrupt .

   function Within_Interrupt_Stack
     (Stack_Address : System.Address) return Boolean;
   pragma Inline ( Within_Interrupt_Stack );
   -- Function that tells whether the Address passed as argument belongs to
   -- the interrupt stack that is currently being used ( if any). It returns
   -- True if Stack_Address is within the range of the interrupt stack being
   -- used. In case Stack_Address is not within the interrupt stack (or no
   -- interrupt is being handled)

end System.BB.Interrupts;
```

Interrupt handlers are always executed using an interrupt stack. The size of the interrupt stack can be modified by the user changing the value of System.BB .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:

- Handler. The address of the procedure used as interrupt handler.

- Id. The interrupt identifier.

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. However, 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.

4.3 Errors

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

4.4 Run-time considerations

Storage allocation

Dynamic storage should only be allocated (from a preallocated pool) during 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.

Interrupt priorities

When attaching protected procedures to interrupts, the ceiling priority of the protected object should be carefully chosen. The compiler checks that the ceiling priority of the protected object is in the range of System. Interrupt_Priority . This range of priorities is mapped to the 15 distinct interrupt levels provided by the SPARC architecture. Therefore, when assigning priorities to protected objects which contains protected procedure handlers, 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 therefore care must be taken by the user not to incur this kind of error.

Potentially blocking operations

Pragma Profile (Ravenscar) includes the pragma Detect_Blocking. Therefore, the exception Program_Error will be raised whenever this kind of bounded error is detected.

Potentially blocking operations are (ALRM 9.5.1):

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

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).

4.5 Tailoring the kernel

ORK+ can be tailored to different applications by means of configuration parameters. Paarmeters are declared in System.BB.Parameters (file `s-bb-para.ads`). This file, as well as other ORK+ files, can be found in the `gcc-4.1.3/gcc/ada` directory.

You can modify this file and rebuild the whole cross-compilation system in order to build a GNATforLEON kernel that satisfies your requirements.

It is recommended that the file `s-bbx-para.ads` be previously compiled with the same flags which will be later used to compile the whole run-time library:

```
$ sparc-elf-gcc -c -gnatpg s-bb-para.ads
```

After updating `s-bbpara.ads`, the GNATforLEON cross-compilation system can be rebuilt from the sources (see section 4.5.3).

4.5.1 Configurable parameters

The configurable parameters included in the System.BB.Parameters package are:

- Interrupt_Stack_Size : Size of the interrupt stack.
- Clock_Frequency: Frequency of the LEON2 processor.

It is also possible to configure the priority ranges by modifying the file `system-xi-sparc-full.ads` and then rebuilding the cross-compilation system.

The configurable parameter which defines the memory space available in the board is included in the linker script file `sparcleon.sc`, which can be found in the`/usr/local/gnatforleon/src/binutils-2.16.1/ld/scripttempl/` directory. You can edit that file and change the `RAM_SIZE` value. There is no need to rebuild the GNATforLEON cross-compilation system when this parameter is changed.

The maximum number of tasks is not configurable. The number of tasks is limited by the task stack size and the amount of storage available. As the storage space needed for tasks is allocated at compilation time, if there is not enough memory a linker error message will be output.

4.5.2 Interrupt names

The interrupt names have been defined in ORK+ as close as possible to the names given in the *Rad-Hard 32 bit SPARC V8 Proccesor AT697E DataSheet* (ATMEL, 2005). For example, the ORK+ name Timer_2 denotes the `Timer 2 time-out` interrupt.

The ORK+ interrupt names are defined in System.BB.Peripherals. These 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:

- Timer_1: This interrupt is issued by the real time clock timer tick.
- Timer_2: This interrupt is issued by the general purpose timer.

- UART_1_RX_TX: This interrupt is generated by the UART channel 1 each time a data word has been correctly received and each time a data word has been sent.
- UART_2_RX_TX: This interrupt is generated by the UART channel 2 each time a data word has been correctly received and each time a data word has been sent.
- External_Interrupt_ {3,2,1,0}: The sources of these interrupts are located outside the LEON2 processor. Consequently these interrupts are input to the processor through pins *ExtINT{3,2,1,0}*.
- DSU: DSU trace buffer interrupt.
- PCI: PCI interrupt.
- Correctable_Error_In_Memory: EDAC interrupt.

4.5.3 Compiling the kernel

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

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

1. Edit the file `Makefile` in order to change GNATforLEON default installation directory (`/usr/local/gnatforleon`) and set the GNATBOOT path to a valid native GNAT GPL 2008 distribution.
2. Type:

   ```
   $ make
   ```

The procedure will take about 10-15 minutes on a modern computer. On successful execution, the GNATforLEON cross-compilation system will be installed.

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

If the GNATforLEON cross-compilation system has been built previously, the subdirectories called `tmp-gcc-4.1.3-build` must already exist in `/usr/local/gnatforleon/src`. It is then possible to rebuild only the GNATforLEON `adalib` by typing

```
$ make adalib
```

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

Warning 4.1 *You will need a native GNAT GPL 2008 distribution installed on your computer in order to rebuild or adapt GNATforLEON 2.1.0.*

Appendix A

The Ravenscar profile

A.1 Introduction

The Ravenscar Profile is the best known result of the *8th International Real-Time Ada Workshop (IRTAW'8)*, which was held in April 1997 in Ravenscar, Yorkshire (Baker and Vardanega, 1997; Burns and Wellings, 1997; Burns et al., 1998). 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 subsequent meetings, including IRTAW'9 (Asplund et al., 1999), IRTAW'10 (Wellings, 2001), IRTAW'11 (Burns and Brosgol, 2002), and IRTAW'12 (Dobbing and de la Puente, 2003). It is included in the ISO report *Guide for the use of the Ada Programming Language in High Integrity Systems* (HIS) (ISO/IEC, 2000), and in the current Ada standard (ISO/IEC, 2007). The summary presented here is based on the *Guide for the use of the Ada Ravenscar Profile in High Integrity Systems* (ISO/IEC, 2005), with a few changes to adapt it to the Ada 2005 standard.

The definition of the Ravenscar profile is based on Ada, including the Systems Programming and Real-Time annexes (ISO/IEC, 2007, annexes C & D). It only addresses tasking constructs, as the reliability aspects of the sequential part of Ada are covered in other sections of the HIS report (ISO/IEC, 2000).

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 ISO/IEC (2005):

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 Ada.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-defined task attributes.

RP17 Dynamic interrupt handler attachments.

RP18 Task termination.

RP19 Specific termination handlers.

RP20 Local timing events.

RP21 Execution-time timers.

RP22 Group budgets.

A.2.2 Supported features

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

RP23 Task objects, restricted as above.

RP24 Protected objects, restricted as above.

RP25 Atomic and Volatile pragmas.

RP26 Delay until statements.

RP27 Ceiling_Locking policy and FIFO_within_priorities dispatching.

RP28 Count attribute (but not within entry barriers).

RP29 Task identifiers, e.g. T'Identity, E'Caller.

RP30 Synchronous task control.

RP31 Task discriminants.

RP32 Ada.Real_Time package.

RP33 Protected procedures as statically bound interrupt handlers.

RP34 Execution-time clocks declared at library level.

RP35 A global fall-back handler.

A.2.3 Dynamic semantics

Some aspects of the profile require their dynamic semantics to be defined:

RP36 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.

RP37 It is implementation-defined what happens if a task attempts to terminate. A global fall-back handler ISO/IEC see 2007, C.7.3 can be set for the environment task. The handler is called whenever a task attempts to terminate.

RP38 If a task executes a potentially blocking operation from within a protected object then Program_Error must be raised.

A.3 Denoting the restrictions

The run-time profile Ravenscar can be enforced with the following pragma:

```
pragma Profile(Ravenscar);
```

which is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy ( FIFO_Within_Priorities );

pragma Locking_Policy ( Ceiling_Locking );

pragma Detect_Blocking;

pragma Restrictions (

      No_Abort_Statements,
      No_Dynamic_Attachment,
      No_Dynamic_Priorities,
      No_Implicit_Heap_Allocations ,
      No_Local_Protected_Objects,
      No_Local_Timing_Events,
      No_Protected_Type_Allocators,
      No_Relative_Delay,
      No_Requeue_Statements,
      No_Select_Statements,
      No_Specific_Termination_Handlers ,
      No_Task_Allocators,
      No_Task_Hierarchy,
      No_Task_Termination,
      Simple_Barriers ,
      Max_Entry_Queue_Length => 1,
      Max_Protected_Entries => 1,
      Max_Task_Entries => 0,
      No_Dependence => Ada.Asynchronous_Task_Control,
      No_Dependence => Ada.Calendar,
      No_Dependence => Ada.Execution_Time.Group_Budget,
      No_Dependence => Ada.Execution_Time.Timers,
      No_Dependence => Ada.Task_Attributes);
```

A.4 Extended profile

ORK+ supports the following features which are not allowed in the Ravenscar profile:

XP01 One execution-time timer per task, declared at the library level.

XP02 Group budgets. Task groups must be static and declared at the library level.

Notice that using `pragma Profile (Ravenscar)` in your program will make the compiler report the use of execution-time timers and group budgets as incorrect. If you need to use the extended features you should provide the full set of pragmas and restrictions listed in A.3 above, except for the two restrictions on `Ada.Execution_Time.Group_Budget` and `Ada.Execution_Time.Timers`.

Appendix B

Example program

B.1 Description

The goal of this example is to show the functionality of GNATforLEON.

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 task is sporadic and the other 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 GNATforLEON 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. LEON2 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 `System.IO` 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

B.2 Temporal requirements of the tasks

Figure B.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 B.1. The period for task A is interpreted as a minimum inter-arrival time.

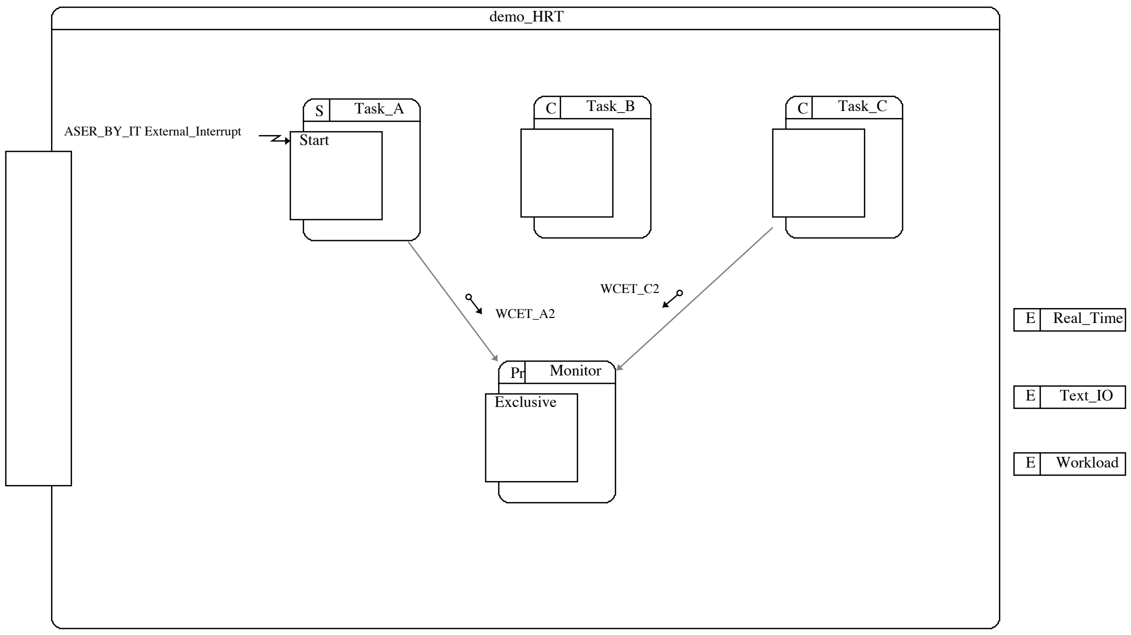

Figure B.1: Example task set

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

Table B.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.

Table B.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.

B.3 Schedulability analysis

The Ravenscar profile includes `pragma Task_Dispatching_Policy (FIFO_Within_-Priorities)` and `pragma Locking_Policy (Ceiling_Locking)` (see appendix A).

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

Therefore, the maximum response time of every task can be evaluated using equation B.1.

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

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 B.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.

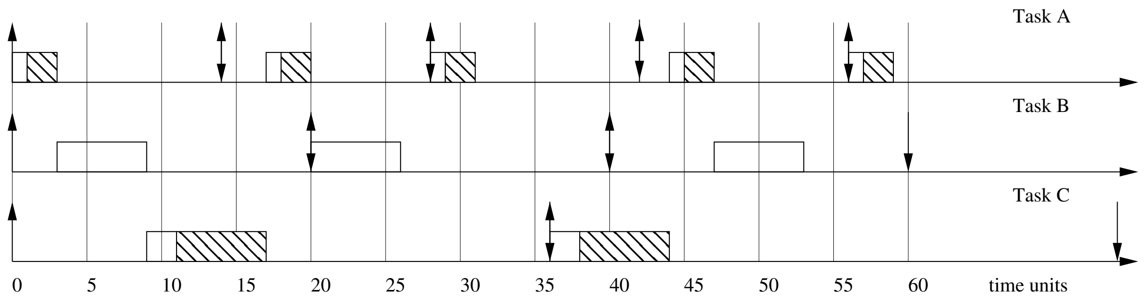

Figure B.2: Schedule of tasks

B.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 41 -fast_uart` can be used to set the clock frequency defined in System.BB.Parameters and to set “infinite” speed in the UART channel.

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

```
$ tsim-leon -freq 41 -fast_uart demo

 TSIM/LEON SPARC simulator, version 2.0.6 (professional version)

  Copyright (C) 2001, Gaisler Research - all rights reserved.
   For latest updates, go to http://www.gaisler.com/
    Comments or bug-reports to tsim@gaisler.com

    serial port A on stdin/stdout
    allocated 4096 K RAM memory, in 1 bank(s)
    allocated 2048 K ROM memory
    icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
    dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
    section: .text, addr: 0x40000000, size 105152 bytes
    section: .data, addr: 0x40019ac0, size 8060 bytes
    read 476 symbols
    tsim> g
    resuming at 0x40000000
    Task A running RT.Clock = 0.00
    Task A finishing RT.Clock = 0.30
    Task B running RT.Clock = 0.30
    Task B finishing RT.Clock = 0.92
    Task C running RT.Clock = 0.92
    Task C finishing RT.Clock = 1.74
    Task A running RT.Clock = 1.74
    Task A finishing RT.Clock = 2.06
    Task B running RT.Clock = 2.06
    Task B finishing RT.Clock = 2.68
```

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

```
Task A running RT.Clock = 2.80
Task A finishing RT.Clock = 3.10
Task C running RT.Clock = 3.60
Task C finishing RT.Clock = 4.42
Task A running RT.Clock = 4.42
Task A finishing RT.Clock = 4.72
Task B running RT.Clock = 4.72
Task B finishing RT.Clock = 5.34
Task A running RT.Clock = 5.60
Task A finishing RT.Clock = 5.90
...
```

There are small variations with respect to the timetable of figure B.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 B.2. As a result, the execution time of protected operations, delay settings, clock readings, and other operations increases the WCET defined in table B.2.

B.5 Example code

Listing B.1: Demo main procedure

```ada
with Tasks;
with System;

procedure Demo is

   pragma Priority (System. PriorityFirst );

begin

   Tasks.Background;

end Demo;
```

Listing B.2: Tasks spec ification

```ada
package Tasks is

   procedure Background;

end Tasks;
```

Listing B.3: Tasks body

```
-- with Kernel. Peripherals ;
with System.IO;

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 tsim -freq 41
   -- 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 (170_540); -- tsim-leon -O2

   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);
   System.IO.Put ("␣RT.Clock␣=␣");
   System.IO.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_Interrupt_2 : constant System. Interrupt_Priority :=
                                    System. Interrupt_Priority ' First + 5;

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 (System.Priority'Last);
end Interrupt;

protected Interrupt_Semaphore is
   pragma Priority ( Priority_Of_External_Interrupt_2 );

   entry Wait;

   procedure 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);
      System.IO.Put ("Task␣");
      System.IO.Put (Running_Task);
      System.IO.Put ("␣finishing" );
      Print_RTClok;
      System.IO.New_Line;
   end Exclusive;

end Monitor;

task body A is
begin
   loop
      Interrupt_Semaphore.Wait;
      System.IO.Put ("Task␣A␣running␣␣");
      Print_RTClok;
      System.IO.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;
      System.IO.Put ("Task␣B␣running␣␣");
      Print_RTClok;
      System.IO.New_Line;
      Execution_Time (WCET_B);
```

```
         Next_Time := Next_Time + Period_B;
         System.IO.Put ("Task␣B␣finishing");
         Print_RTClok;
         System.IO.New_Line;
      end loop;
   end B;

   task body C is
      Next_Time : Ada.Real_Time.Time := Time_Zero;
   begin
      loop
         delay until Next_Time;
         System.IO.Put ("Task␣C␣running␣␣");
         Print_RTClok;
         System.IO.New_Line;
         Execution_Time (WCET_C1);
         Monitor. Exclusive (WCET_C2, 'C');
         Next_Time := Next_Time + Period_C;
      end loop;
   end C;
-- begin
  -- Kernel. Peripherals . Init_UART (Channel => 1, BaudRate => 115200,
    --                               Parity => Kernel.Peripherals. None,
--                                FlowControl => Kernel.Peripherals. Off);
end Tasks;
```

Listing B.4: Force_External_Interrupt_2

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

procedure Force_External_Interrupt_2 is

   package SPR renames System.BB.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

    Interrupt_Mask_Auxiliary  : SPR.Interrupt_Mask_Register :=
                                SPR.Interrupt_Mask;
     Interrupt_Force_Auxiliary : SPR. Interrupt_Register :=
                                 SPR. Interrupt_Force ;
   Interrupt_Clear_Auxiliary   : SPR. Interrupt_Register  :=
                                 SPR. Interrupt_Clear ;

begin

   Interrupt_Clear_Auxiliary  . External_Interrupt_2  := True;
   Interrupt_Mask_Auxiliary . External_Interrupt_2  := True;
   Interrupt_Force_Auxiliary  . External_Interrupt_2  := True;

   SPR. Interrupt_Clear  :=  Interrupt_Clear_Auxiliary  ;
   SPR.Interrupt_Mask := Interrupt_Mask_Auxiliary ;
   SPR. Interrupt_Force  :=  Interrupt_Force_Auxiliary  ;

end Force_External_Interrupt_2 ;
```

Listing B.5: Configuration file

```
-- gnat.adc - minimum configuration file template for the Ravenscar profile
pragma Profile (Ravenscar);

-- Any other configuration pragma can be included here
```

Appendix C

Metrics

C.1 Ada LRM Annex C and D Metrics

This appendix lists the additional characteristics of the ORK+ 2.1.0which are specified in Annex C and D of the Ada Language Reference Manual and are allowed by the Ravenscar profile.

The metrics are expressed in processor cycles. To relate them to time figures in seconds, they should be divided by the processor clock frequency.

C.1.1 Protected Procedure Handlers

ALRM C3.1 15. Overhead of interrupt handlers.

The following metric shall be documented by the implementation:

- *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 long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.*

  The measured values are:

```
=== interrupt_overhead_test ===============================
 Cycles> 1969334
 Cycles> 1970270
 cycles A+B: 4930583
 cycles C: 4931993
 cycles C-(A+B) = 1410
==========================================================
```

  The average overhead of a protected procedure as interrupt handler is 1410 processor cycles.

This metric has been measured with the TSIM 2.0.6 simulator. Otherwise, special hardware such as signal generators and logic analyzers should be used to measure this overhead on real target. Other metrics have been measured on the same simulator as well as on real hardware, and the differences were insignificant.

C.1.2 Monotonic Time

ALRM D.8 33. Definition of type Time.

The implementation shall document the values of Time_First, Time_Last, Time_Span_Last, Time_Span_Unit, and Tick.

These metrics are under development.

ALRM D.8 34. Time base and underlying hardware.

The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities.

Time is represented internally as a 64-bit integer number of ticks. The tick is currently 4 times the period of the processor clock. Therefore, the interval of time values that can be represented in this way is approximately -23360..+23360 years for a 50 Mhz processor. Time is a count of ticks since the clock was started.

The LEON2 hardware provides two timers (apart from the special 'Watchdog timer') which can be programmed to be either of single-shot type or of periodic 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 significant 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 significant part of the clock.

As a result, the clock tick is equal to the period of the input signal of the downcounter divided by the prescaler. That is the processor clock period divided by 4 in the current implementation.

The underlying implementation is described in depth in Zamorano et al. (2001).

ALRM D.8 35. Synchronization with external sources.

The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied.

The system does not synchronize with any external source.

ALRM D.8 36. Interference from the external environment.

The implementation shall document any aspects of the external environment that could interfere with the clock behavior as defined in this clause.

The clock value is a count of ticks since the clock was started, therefore its behavior only is affected by the manufacturing drift of the hardware clock.

ALRM D.8 39–45. Real-time clock metrics

For the purpose of the metrics defined in this clause, real time is defined to be the International Atomic Time (TAI).

The implementation shall document the following metrics:

- *An upper bound on the real-time duration of a clock tick. This is a value D such that if $t1$ and $t2$ are any real times such that $t1 < t2$ and $\mathrm{Clock}_{t1} = \mathrm{Clock}_{t2}$ then $t2 - t1 \leq D$.*

  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.

- *An upper bound on the size of a clock jump.*

  The clock is a count of ticks and is not synchronized with external sources. As a result, the clock does not jump.

- *An upper bound on the drift rate of Clock with respect to real time. This is a real number D such that*

  $$E \cdot (1–D) <= (\mathrm{Clock}_t + E–\mathrm{Clock}_t) <= E \cdot (1 + D)$$

  provided that: $\mathrm{Clock}_t + E \cdot (1 + D) <= \mathrm{Time\_Last}$.

  This test is yet to be performed on the GR-XC3S-1500 or GR-CPCI-AT697 boards.

- *where* Clock_t *is the value of* Clock *at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported.*

  To be completed when the previous test is performed.

- *An upper bound on the execution time of a call to the* Clock *function, in processor clock cycles.*

  The following source code line was tested:

  ```
  T := Clock;
  ```

  The measured value is:

  ```
  === D8-44-clock\_call ====================================

  total processor cycles used by a 'Clock' call: 522

  ==========================================================
  ```

- *Upper bounds on the execution times of the operators of the types* Time *and* Time_Span*, in processor clock cycles.*

  The results of the test give the following WCET in cycles for each operation:

Operator	**Cycles**
Time + Time_Span	58
Time_Span + Time	58
Time − Time_Span	58
Time − Time	58
Time < Time	44
Time ≤ Time	72
Time > Time	62
Time ≥ Time	34
Time_Span + Time_Span	58
Time_Span − Time_Span	58
− Time_Span	47
Time_Span × Integer	92
Integer × Time_Span	99
Time_Span / Time_Span	1098
Time_Span / Integer	1092
abs(*Time_Span*)	44
Time_Span < Time_Span	98
Time_Span ≤ Time_Span	90
Time_Span > Time_Span	100
Time_Span ≥ Time_Span	100
To_Duration(*Time_Span*)	1139
To_Time_Span(*Duration*)	129
Split(*Time, Seconds_Count, Time_Span*)	1142
Time_Of(*Seconds_Count, Time_Span*)	96
Nanoseconds(*Integer*)	1096
Microseconds(*Integer*)	1096
Milliseconds(*Integer*)	55

C.1.3 Delay Accuracy

ALRM D.9 9–13. Metrics.

The implementation shall document the following metrics (only those metrics that are significant in the context of the Ravenscar profile are cited)*:*

- *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.*

  The measured value is equal to 740 processor clock cycles.

- *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.*

The following measurements have been performed:

– *One task + background task*

  The delay until lateness upper bound for a call to a delay until statement is 8051 clock cycles (161 μs), using a 50 MHz system clock. This lateness occurs when the time of the delay until coincides with a second boundary. 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 coincide with a clock interrupt, the lateness upper bound for the execution of a delay until statement is 7061 clock cycles (141.2 μs).

– *N tasks + background task*

  The lateness of delay until for N tasks is $294825.84 + 7 \times N$ μs when the time of the delay until coincides with a clock interrupt. Otherwise, it is $201.8 + 7 \times N$ μs.

C.1.4 Other Optimizations and determinism rules

ALRM D.12 7–11. Metrics.
The implementation shall document the following metric:

- *The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. This shall be measured in the following way:*

  For a protected object of the form:

```
protected Lock is
   procedure Set;
   function Read return Boolean;
private
   Flag : Boolean := False;
end Lock;

protected body Lock is
   procedure Set is
   begin
      Flag := True;
   end Set;
   function Read return Boolean
   Begin
      return Flag;
   end Read;
end Lock;
```

  The execution time, in processor clock cycles, of a call to Set*. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function* Read *shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period.*

The protected object shall have sufficiently high ceiling priority to allow the task to call Set.

The number of processor clock cycles used by a call to the protected procedure Set is 1214.

C.1.5 Execution Time Clocks and Timers

D.14 23–27. Metrics

The implementation shall document the following metrics:

- *An upper bound on the execution-time duration of a clock tick. This is a value D such that if $t1$ and $t2$ are any execution times of a given task such that $t1 < t2$ and $\text{Clock}_{t1} = \text{Clock}_{t2}$ then $t2–t1 \leq D$.*

  The maximum value of a clock tick is $1 \times P$ processor cycles, where P is the setting of the prescaler register (= 4 for the current default ORK+ configuration).

- *An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution-time clock (as observed by calling the Clock function with the same* Task_Id*).*

  The maximum value of a clock jump is $110 \times P$ processor cycles, where P is the setting of the prescaler register (= 4 for the current default ORK+ configuration).

- *An upper bound on the execution time of a call to the* Clock *function, in processor clock cycles.*

  The execution time of a call to the Clock function is 439 clock cycles.

- *Upper bounds on the execution times of the operators of the type* CPU_Time*, in processor clock cycles.*

  The results of the test give the following values:

Operator	Cycles
CPU_Time + Time_Span	58
Time_Span + CPU_Time	58
CPU_Time − Time_Span	58
CPU_Time − CPU_Time	58
CPU_Time < CPU_Time	65
CPU_Time ≤ CPU_Time	77
CPU_Time > CPU_Time	73
CPU_Time ≥ CPU_Time	61
Split(*CPU_Time, Seconds_Count, Time_Span*)	1142
Time_Of(*Seconds_Count, Time_Span*)	80

C.1.6 Timing Events

D.15 23–24. Metrics.

The implementation shall document the following metric:

- *An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.*

  The measured value for this metric is 4756 processor clock cycles.

C.2 Other useful metrics

C.2.1 Context Switch

The context switch time is measured between two tasks with the same priority.

The worst case measured value is 1027 processor clock cycles.

C.2.2 Interrupt Latency

This tests measures the interrupt latency. This test uses a special register of LEON2 which permits interrupts to be enforced.

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 measured value of interrupt latency is 1747 processor clock cycles.

This elapsed time has been measured with the TSIM simulator. Otherwise special test instrument should be used. The example program detailed in appendix B has been used for the test.

Appendix D

GNU General Public License

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 licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an

announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable 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 modifications to it. For an executable work, complete source code means

End of terms and conditions

How to Apply These Terms to Your New Programs

Bibliography

Ada Core (2007). *Using the GNAT Programming Studio.* Version 4.2.1.

Ada Core (2008a). *GNAT GPL User's Guide.*

Ada Core (2008b). *GNAT Reference Manual.*

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

ATMEL (2005). *32 bit SPARC V8 Proccesor AT697E DataSheet.*

Ausnit-Hood, C., Johnson, K. A., IV, R. G. P., and Opdahl, S. B., editors (1995). *Ada 95 Quality and Style.* Number 1344 in Lecture Notes in Computer Science. Springer-Verlag.

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

Barnes, J. (2008). *Ada 2005 Rationale.* Number 5020 in Lecture Notes in Computer Science. Springer-Verlag.

Burns, A. (1994). Preemptive priority based scheduling: An appropriate engineering approach. In Son, S. H., editor, *Advances in Real-Time Systems.* Prentice-Hall.

Burns, A. and Brosgol, B. (2002). Session summary: Future of the Ada language and language changes such as the Ravenscar profile. *Ada Letters*, XXII(4). Proceedings of the 11th International Real-Time Ada Workshop.

Burns, A., Dobbing, B., and Romanski, G. (1998). The Ravenscar tasking profile for high integrity real-time programs. In Asplund, L., editor, *Reliable Software Technologies — Ada-Europe'98*, number 1411 in LNCS, pages 263–275. Springer-Verlag.

Burns, A. and Wellings, A. J. (1997). Restricted tasking models. In *Proceedings of the 8th International Ada Real-Time Workshop*, pages 27–32. Ada Letters.

Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages.* Addison-Wesley, 3 edition.

Dobbing, B. and de la Puente, J. A. (2003). Session report: Status and future of the Ravenscar profile. *Ada Letters*, XXIII(4):55–57. Proceedings of the 12th International Real-Time Ada Workshop (IRTAW 12).

ECSS (2003). *ECSS-E-40 Part 1B: Space engineering — Software — Part 1: Principles and requirements.* Available from ESA.

Gaisler Research (2005). *LEON2 Processor User's Manual.*

Gaisler Research (2007). *GRMON User's Manual.* Available at http://www.gaisler.com/doc/grmon.pdf.

Giering, E. W. and Baker, T. P. (1994). The GNU Ada Runtime Library (GNARL): Design and implementation. In *WADAS '94: Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada*, pages 97–107, New York, NY, USA. ACM Press.

ISO/IEC (1999). *Std. 9899:1999 — Programming Languages — C.*

ISO/IEC (2000). *TR 15942:2000 — Guide for the use of the Ada programming language in high integrity systems.*

ISO/IEC (2003). *Std. 9945-1:2003 Portable Operating System Interface (POSIX).*

ISO/IEC (2005). *TR 24718:2005 — Guide for the use of the Ada Ravenscar Profile in high integrity systems.* Based on the University of York Technical Report YCS-2003-348 (2003).

ISO/IEC (2007). *Std. 8652:1995/Amd 1:2007 — Ada 2005 Reference Manual. Language and Standard Libraries.* Published by Springer-Verlag, ISBN 978-3-540-69335-2.

SPARC International (1992). *The SPARC architecture manual: Version 8.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Stallman, R. M. and Pessch, R. H. (2007). *Debugging with GDB.* Free Software Foundation, 9th edition. For GDB version 6.6 for GNAT GPL 2007.

Urueña, S., Pulido, J. A., Redondo, J., and Zamorano, J. (2007). Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Letters*, XXVII(2):61–66. Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007).

Wellings, A. (2001). 10th International Real-Time Ada Workshop — Session summary: Status and future of the Ravenscar profile. *Ada Letters*, XXI(1).

Zamorano, J., Ruiz, J. F., and de la Puente, J. A. (2001). Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In Strohmeier, A. and Craeynest, D., editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 317–327. Springer-Verlag.