A Detailed Description of the GNU Ada Run Time

(Version 1.0)

Integrated with the
GNAT 3.15p sources
and the
Annotated Ada Reference Manual (Technical Corrigendum 1)

jmiranda@iuma.ulpgc.es

Applied Microelectronics Reseach Institute
University of Las Palmas de Gran Canaria
Canary Islands
Spain

14th December 2002

To my wife Pino,
my children Fayna and Tahiche,
and my parents Candelaria and Antonio.

Preface

It is well known that free software needs free manuals: reference and user manuals[1]. This allows the general community to *use* the technology. However, free software has a tremendous potential for research and teaching: the sources permit us to *understand* the technology. Nowadays, due to the lack of free books which describe the behaviour of free technology, most researchers and educators interested in it must repeat the same unpleasant task: to read the sources to try to understand the architecture and details of the free software. Obviously this is time consuming and error prone. This is also the case of the free GNU Ada Compiler: GNAT.

GNAT is currently a mature technology used for many industrial projects, research, and education. However, the lack of free books which describe real implementation hinders the modification of the sources to many researchers. The main benefits of a free book on GNAT are:

- **The book can be used and enhanced by many researchers.** If the book is free it can be used as a shared document which provides an easy way to learn about the Run-Time (thus reducing the time required to experiment with Ada), but the book can also be upgraded by any researcher of the Ada community.
- **The book facilitates the use of Ada to teach students about compilers and real-time systems.** The sources of the GNAT run-time are very well commented and written in Ada, thus providing a high level of abstraction which is quite useful for teaching.

[1] R. Stallman. *Free Software and Free Manuals*. Essay. Avaiable at http://www.gnu.org/philosophy/free-doc.html

The main goals of this project are:

- **To provide documentation on the GNAT Run-Time.** The contents of the book is my knowledge on the GNAT run-time.
- **To keep the book fully integrated with the GNAT sources.** Although the book is distributed in several formats for printing, the goal is to write a digital book with many links to the GNAT sources which facilitate the understanding and verification of the main concepts of the Ada run-time.
- **To keep the document free.** The book is distributed under the GNU Free Documentation License (cf. Appendix A). This allows members of the Ada community to enhance the quality of the book for research and teaching.

Technical Contents

In order to facilitate the reading of the book for teaching, each chapter is structured in two parts. The first part has a brief summary of the Ada concepts described in the chapter (I used the excellent book *Concurrency in Ada* [BW98] as a reference to write these summaries). The second part of the chapter describes GNAT implementation of the concepts presented in the first part. Currently the book has the following chapters:

1. **The GNAT project.** Brief introduction to the GNAT project, and the overall architecture of the compiler and run-time.
2. **Task types and objects.** This chapter describes the main aspects of the life-cycle of Ada tasks: task creation, task activation, task termination, and task identification.
3. **The Rendezvous.** This chapter deals with the handling of the entry call parameters, the rendez-vous queues and the basic rendezvous modes (simple, conditional and selective).
4. **Protected Objects.** The Ada95 eggshell model for protected objects and its implementation is presented here. Several alternative implemen-tations are also described and discussed.
5. **Time and clocks.** This chapter covers the Ada timed sentences: timed entry call and timed selective accept.

6. **Interrupts.** The Ada model of interrupts and its implementation is presented here.

7. **Exceptions.** Data types used to identify the exceptions, and hash table used by GNAT to handle the exceptions are presented here.

8. **Abort and Asynchronous Transfer of Control.** Ada tasks abortion and the implementation of the Ada95 asynchronous transfer of control are discussed in this chapter.

9. **Bibliography.** This chapter provides the bibliography used to write the book.

Distribution

The book is available at:

- http://gnat.webhop.info
- http://www.iuma.ulpgc.es/users/jmiranda/

The first address provides an "easy to remember" access to the second address, which is the real address of my web page. Nowadays the distribution of the book includes:

- The **html** version of the book. The book is always available by internet as an on-line book (accessible to your preferred web navigator).
- A compressed **html.tgz** file with the html sources. This distribution includes one script which patches the html files and installs them locally (in any local directory) or in your personal web page.
- The **PostScript**, **PDF**, and **DVI** files of the whole book. This distribution facilitates the printing of the book. In order to not lose information, most of the html distribution hyperlinks are included in this distribution by means of footnotes.
- The **LaTeX sources** of the book. This distribution allows the members of the Ada community to cooperate to improve the quality of the document.

Although it is well known that the GNU format for free books is Texinfo, the book is written in LaTeX. The main reasons are:

- I do not know Texinfo and I feel quite familiar with LaTeX.
- LaTeX is well known to the scientific community. This facilitates any member of the Ada community the modification of the sources of the book to improve its contents.
- LaTeX is also free software. It is obvious that a free-book must be written with free software to avoid any future problem with the distribution of the sources (due to the copyright of the file formats).
- LaTeX2html is a good tool to make the automatic translation of the book to HTML. Therefore it is not necessary to maintain two versions of the book (one for the printed version, and another for the HTML files).

Background

A previous version of this book was written in 1999 under the title *"How to modify the GNAT Run-Time to experiment with Ada extension"* [MGGM99]. That book was the result of a project to integrate Drago[5] into the GNAT sources. (Drago was the result of my PhD research. It is an Ada extension which facilitates the programming of fault-tolerant and cooperative distributed applications by means of the addition of the groups paradigm into the Ada language).

Although the book was available on the web, no publicity was sent out. Ingenuously we thought that the internet search engines would provide the book to any interested people. Obviously very few people found it, but we thought that the book was not good enough and we stopped our efforts.

During the AdaEurope2001 conference (Belgium) I presented the book to some colleagues and felt that the book could still be of interest for the Ada community. Therefore, in July 2001 I decide to personally restart the project with the following goals:

- **Write the manuscript in English.** The manuscript of the previous version was written in Spanish and translated to English. Because I was now alone to do the whole work, I decided to concentrate any effort in the most widely distributable version of the book. Once completed many people can help to translate it to other languages.

- **Recover the (still) useful documentation of the previous book.** Although most of the chapters of the previous book were now obsolete, some parts were still reusable (the previous book described the GNAT-3.10p sources).
- **Structure each chapter of the book in two parts.** The first part summarizes the Ada concepts whose implementation is described in the second part. This will facilitate the use of the book for teaching.
- **Reduce the maintenance cost as much as possible.** A single document should be used for all the distributions of the document. I decided to write the sources of the book with LaTeX because it is free and has many free tools to translate the document to HTML, PostScript, PDF, and DVI files.
- **Keep the book integrated with the GNAT sources.** The HTML version of the book would be the "star" of the project with hyperlinks which facilitate to the reader the direct access to the Annotated Ada Reference Manual [AAR95] and GNAT sources.

I have been working on this book for one year and half. In July 2002 the first Beta version of the book was available in the WEB. In September 2002 I added the GNU Free Documentation License to the book, and I put the full sources on my web site. Now (December 2002) I give you the upgraded version of the book which is now integrated with the GNAT 3.15p sources and has 288 hyper-links to these sources.

Acknowledgements

I would like to thank professor Edmond Schonberg (New York University) for replying to me many messages with questions on the GNAT compiler. I also thank professor Ted Baker (Florida State University) for offering his help to enhance future versions of this book.

I acknowledge Alexis González and José Jerónimo Martín for the tremendous effort done during the previous project (which was the base of this book). I also thank my colleagues of the *Distributed Systems Research Group* (University of Las Palmas de Gran Canaria) Francisco Guerra, Ernestina Martel, José Miguel Santos, and Luis Hernández for the discussions which have helped me to understand many details of the GNAT sources. I also thank David Shea (Faculty of Translation and Interpreting, University of Las Palmas de Gran Canaria) for all the corrections done to improve the quality of this English text.

Finally, I wish to thank Angel Alvarez (Technical University of Madrid) and Sergio Arvalo (Rey Juan Carlos University at Madrid) not only for their guidance during my PhD studies, but also for their generosity and unfailing support.

Short Biography

Javier Miranda was born in 1965 in Canary Islands (Spain). He studied Computer Science Engineering at the University of Las Palmas de Gran Canaria. He finished his studies by implementing a Modula-2 compiler under the direction of José Fortes Gálvez.

In 1990 he went to the Technical University of Madrid (Department of Telematic Systems Enginnering —DIT) to do his PhD research under the direction of Angel Álvarez and Sergio Arévalo (in the *Distributed Systems Research Group*). In 1994 he finished his PhD thesis entitled *"Drago: A Language for Programming Fault-Tolerant and Cooperative Distributed Applications*.

In 1997, he started a project to integrate Drago into the GNAT sources. This project was done in collaboration with his colleague Francisco Guerra (who implemented the required protocols by means of the Group_IO Ada Library), and the students Alexis Rodríguez and José Jerónimo Martín.

After the project he personally continued upgrading Drago to the next GNAT distributions. This book summarizes the experience achieved during these years.

Las Palmas de Gran Canaria, December 2002

Contents

List of Figures

Chapter 1

The GNAT Project

GNAT (an acronym for **G**NU **N**YU **A**da **T**ranslator) is a front-end and runtime system for *Ada 95* that uses the successful GCC back-end as a retargettable code generator. GNAT is thus part of the GNU[1] software, and is distributed according to the guidelines of the *Free Software Foundation*[2] [SGC94, Section 1]. GNAT has been developed by two cooperating teams:

- GNAT Development Team (New York University). Guided by professors Edmond Schonberg and Robert B.K. Dewar. This group developed the front-end of the Ada compiler.
- Project PART Team[3] (Florida State University). Guided by professor Theodore P. Baker This group developed the Ada Run-Time Library.

The project was initially sponsored by the U.S. government (from 1991 to 1994). In August, 1994 the main authors created the *Ada Core Technologies, Inc.*[4] company which gives technical support under contract to the entities which use GNAT with industrial or commercial products. Nowadays *Ada Core* continues extending the platforms for which the compiler is available and also provides tools for the development and debugging of the Ada programs. The company provides free upgrades of the compiler to the Ada community.

In this chapter, the external and internal structure of GNAT is described.

[1] http://www.gnu.org/
[2] http://www.gnu.org/fsf/fsf.html
[3] PART - POSIX Ada Real-Time.
[4] E-mail: info@gnat.com; http://www.gnat.com/

1.1 GCC

GCC is the compiler system of the GNU environment. GNU (a self-referential acronym for 'GNU is Not Unix') is a Unix-compatible operating system, being developed by the Free Software Foundation, and distributed under the GNU Public License (GPL)[5].

GNU software is always distributed with its sources, and the GPL enjoins anyone who modifies GNU software and then redistributes the modified product to supply the sources for the modifications as well. Thus, enhancements to the original software benefit the software community at large [Sta92].

GCC is today the centerpiece of the GNU software. GCC is a retargetable and rehostable compiler system, with multiple front-ends and a large number of hardware targets. Originally designed as a compiler for C, it now includes front-ends for C++, Modula-3, Fortran, Objective-C, and most recently Ada. Technically, the crucial asset of the GCC is its mostly language-independent, target-independent code generator, which produces excellent quality-code both for CISC machines such as the Intel and Motorola families, as well as RISC machines including the IBM RS/6000, the DEC Alpha and the MIPS R4000. Remarkably, the machine dependences of the code generator represent less than 10 new target to GCC, an algebraic description of each machine instruction must be given using a register-transfer language. Most of the code generation and optimization then uses the RTL, which GCC maps when necessary into the target machine language. The leverage for constructing a front-end for GCC is thus enormous: GNAT potentially has over 30 targets.

Furthermore, GCC produces high-quality code, comparable to that of the best commercial compilers [SGC94, Section 2].

1.2 GNAT Organization

The first decision involved choosing the language in which GNAT should be written. GCC is fully written in C, but for technical reasons as well as non-technical ones, it was inconceivable to use anything but Ada for GNAT itself. The GNAT team started using a relatively small subset of Ada83, and in typical fashion, extended the subset whenever new features became implemented. Six months after the coding started in earnest, we were able to bootstrap the compiler, and abandon

[5] http://www.gnu.org/copyleft/gpl.html

the commercial compiler we had been using up to that point. As Ada95 features are implemented, we are now able to write GNAT in Ada95. In fact, the definition of the language depends heavily on hierarchical libraries, and cannot be given except in Ada95, so that it is natural for the compiler and the environment to use child units throughout [SGC94, Section 3.1].

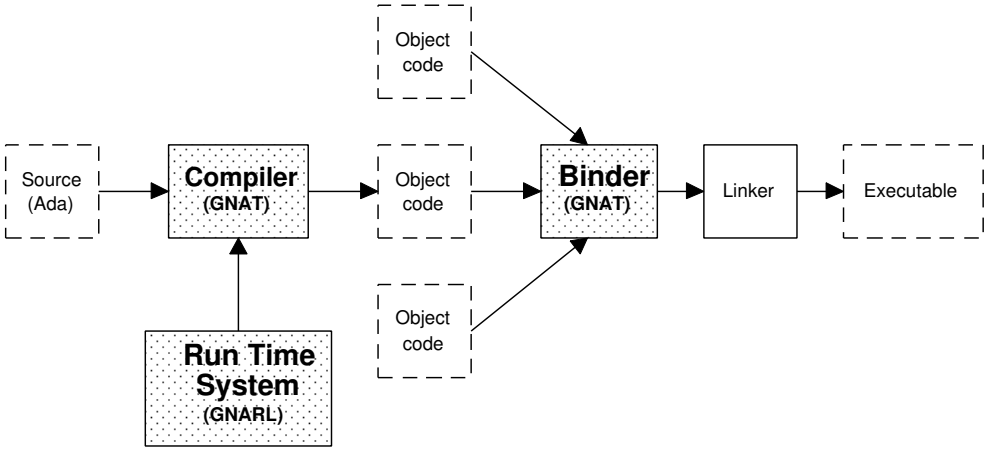

Figure 1.1: GNAT Overall Structure.

Figure 1.1 presents the overall structure of the GNAT system. It has three main parts: the *Compiler*, the *Run Time System* and the *Binder* (the GNU linker for the target operating system is always reused). From the figure, we can also deduce the steps followed to compile an Ada program.

The GNAT binder verifies the consistency of the objects and determines a valid order of elaboration (initialization) for the objects (from the same or different languages) that are to be assembled into an executable file. Following this sketch, the next sections of this chapter describe each part of the compiler.

1.3 The Compiler

The compiler is composed of two main parts: the *front-end* and the *back-end* (cf. Figure 1.2). The front-end of the GNAT compiler is thus written in Ada95. The back-end of the compiler is the back-end of GCC proper, extended to meet the needs of Ada semantics [SGC94, Section 3.1].

The front-end comprises five phases (cf. Figure 1.3): lexical analysis, syntactic analysis (parsing), semantic analysis, AST expansion, and finally AST transformation into an equivalent C tree (this stage is labeled GiGi (GNAT to GNU

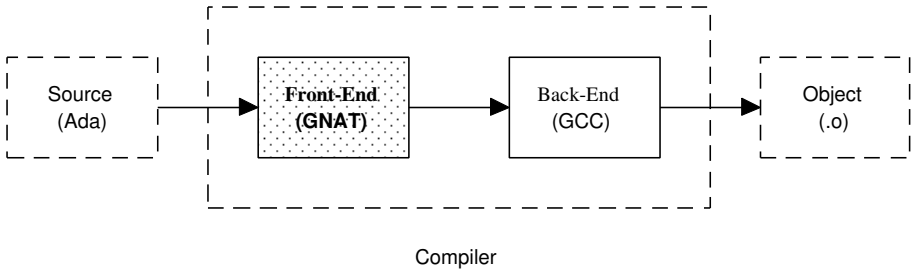

Figure 1.2: GNAT Compiler.

transformation). These phases communicate by means of a rather compact Abstract Syntax Tree (AST). The implementation details of the AST are hidden by several procedural interfaces that provide access to syntactic and semantic attributes. The layering of the system, and the various levels of abstraction, are the obvious benefits of writing in Ada, in what one might call "proper" Ada style. It is worth mentioning that strictly speaking GNAT does not use a symbol table. Rather, all semantic information concerning program entities is stored in defining occurrences of these entities directly in the AST [SGC94, Section 3.1].

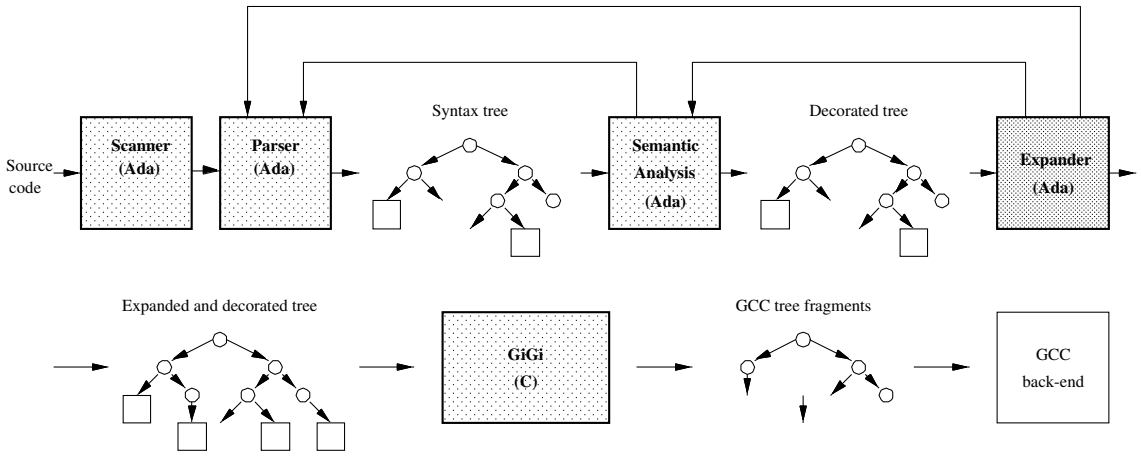

Figure 1.3: GNAT Front-end Stages.

As the figure 1.3 shows, the *Scanner* starts analyzes the input file and generates the associated *Tokens*. The *Parser* analyzes the syntax of the tokens and creates the Abstract Syntax Tree (syntactic analysis). The *Semantic Analyzer* performs name and type resolution (that is, it resolves all the possible ambiguities of the source code), decorates the AST with various semantic attributes, and as by-product performs all static legality checks on the program. After that, the *Expander* transforms high level AST nodes (nodes representing tasks, protected ob-

jects, etc.) into nodes which call to Ada Run-Time library routines. (Multi-tasking constructs are generally implemented by a combination of high-level source code transformations and calls to Ada Run-Time Library [DIB94, Section 4.2.1]).

Most of the expander activity results in the construction of additional AST fragments. Given that code generation requires that such fragments carry all semantic attributes, every expansion activity must be followed by additional semantic processing on the generated tree. This recursive structure is carried further: some predefined operations (i.e. exponentiation) are defined by means of a generic procedure. The expansion of the operation results in the generic instantiation (and corresponding analysis) of this generic procedure [SGC94, Section 3.3]. At the end of this process the *GIGI* phase transforms the AST into a tree understable by the GCC backend. This phase is an interface between the GNAT front-end and the GCC back-end. In order to bridge the semantic gap between Ada and C, several code generation routines in GCC have been extended, and others added, so that the burden of translation is also assumed by Gigi and GCC whenever it would be awkward or inefficient to perform the expansion in the front-end. For example, there are code generation actions for exceptions, variant parts and accesses to unconstrained types. As a matter of GCC policy, the code generator is extended only when the extension is likely to be of benefit to more than one language [SGC94, Section 3.4].

There is a further unusual recursive aspect to the structure of GNAT. The program library (described in greater detail below) does not hold any intermediate representation of compiled units. As a result, package declarations are analyzed whenever they appear in a context clause. Furthermore, if a generic unit, or an inlined unit `G`, is defined in a package `P`, then the instantiation or inlining of `G` in the current compilation requires that the body of `P` be analyzed as well. Thus the library manager, the parser and the semantic analyzer can be activated from within semantic analysis (note the backward arrows in figure 1.3) [SGC94, Section 3.3].

1.4 The Run Time System

In order to make GNAT portable, the Ada Run-Time System (RTS) is written in Ada. The compiler communicates with the RTS through procedure and function calls, without direct reference to RTS data structures aside from the parameters of the RTS subprograms. The RTS data structures may be kept in a separate address space, protected from access by the application. The direction of call is always from application code to the RTS [GB94a, Section 2]. The exceptions to this rule are:

- Task creation, in which the compiler passes to the RTS the address of a procedure corresponding to the task body.

- Protected entries, in which the compiler passes to the RTS the address of an array with the reference to the subprograms generated by the compiler.

Thus, the opportunities for optimization involve alternate source-code transformations, and alternate algorithms and data structures in the runtime library routines [DIB94, Section 4.2.1].

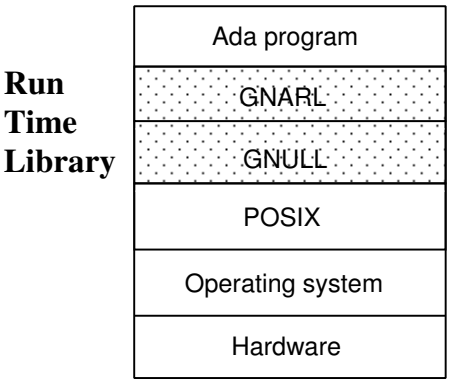

Figure 1.4: The GNAT Run Time Library.

Figure 1.4 presents Run-Time hierarchy. The Run-Time library is made up of three levels: `GNARL, GNULL` and `Pthreads`[6]. An Ada program requests the services of the Run Time through the `GNARL` subprograms calls. This level uses the services provided by the `GNULL` level. This intermediate level is an interface between the upper level and the POSIX standard library. POSIX Pthreads provides support to languages for concurrent programming.

1.4.1 GNARL

GNARL is the **GN**U **A**da **R**un-time **L**ibrary. High level language constructs are translated by the expander into calls to this library. Packages that constitute the run-time are treated as any other unit of the context of the compilation, and analyzed when necessary. This obviates the need to place run-time information in the compiler itself, and allows a knowledgeable user to modify the run-time if he/she

[6]Pthreads - POSIX Threads.

so chooses. The design of GNARL is based on the CARTS (Common Ada Run-Time System) specification [SGC94, Section 3.6]. The original design objectives of GNARL, in order of priority, were [GB94a, Section 1]:

1. *Semantic correctness*. GNARL must support the full core tasking semantics, and as much of the Real-Time Systems and Systems Programming annexes as permitted by other constraints.

2. *Timeliness*. Development should be incremental, so that working partial implementations can be delivered early. The RTS and compiler development should proceed in parallel. This means the RTS responsibilities should be clearly separated from those of the compiler. Ada features that early versions of the compiler are not likely to be able to translate should be avoided. Ideally, it should be possible to compile and test it using an existing Ada 83 compiler. It should use the existing PART RTS code for leverage, to speed development.

3. *Modularity*. GNARL should be partitioned to hide information that the compiler does not need to know, including information about tasking features that are not used by a particular compilation. The GNAT strategy for implementing tasking is based on tree-to-tree translation, converting tasking constructs in the intermediate syntax tree representation into equivalent Ada language constructs, with interspersed RTS service calls. Thus, the GNAT interface should be expressible in terms of ordinary Ada packages whenever possible.

4. *Portability*. GNARL should be written in Ada, with target-specific code clearly isolated in the GNULL level and kept as small as possible. It should be possible to produce configurations that run both over commercial off-the-shelf operating systems and on bare machines with minimal modifications. Initially, it should be supported on several commonly used operating systems.

5. *Research and technology transfer*. GNARL should serve as a test-bed for implementation ideas, providing experience that will be useful in designing other implementations. Lessons learned should be reported promptly, and code made publicly available. Among other experimental goals, GNARL should provide a basis for measuring the overhead imposed by implementing Ada over Pthreads.

6. *Efficiency*. GNARL should be as efficient as possible, consistent with the other objectives. The design should allow for future optimizations, includ-

ing inline expansion and optimization of RTS calls. This means using ordinary procedure calls in the interface, as opposed to traps or calls via procedure pointers".

GNARL is designed to facilitate the in-line optimization of Ada tasking constructs. The use of task constructs results in the implicit **with** of one or more of the packages that make up the GNARL by the GNAT compiler. Other than this implicit import, GNARL packages are indistinguishable from other application packages. There are no special restrictions on GNARL code. In particular, GNARL subprograms can be named in Inline pragmas, resulting in the replacement of implicit calls to these subprograms with the subprogram body. This should result in somewhat faster code due to the elimination of the subprogram call. However, once the code has been inserted inline, it can be further optimized by the compiler using information about the local environment including current register contents. This process is further augmented by the inline nature of the GNARL interface. Tasking is implemented with calls to the GNARL interleaved with user code. The only exception to this is task startup, where GNARL executes the task body code from a new thread of control via call-back. This inline nature of the GNARL interface is intended to allow local optimizations across the boundaries between the application and the GNARL, in particular when the GNARL calls are expanded inline. This kind of optimization is much less applicable with an interface involving call-backs to user code within the RTS. Each call-back point can call one of an arbitrary number of user code sequences, so they cannot be inlined, and it is less likely that local optimizations (i.e. register allocation) will be applicable to all of them [GMB94, Section 3].

Implementing GNARL semantics is relatively complex, and will probably be of interest only to users requiring unusual tasking semantics, or to take advantage of unusual hardware architecture (i.e. multiprocessing or distributed environments).

1.4.2 GNULL

GNULL exists only for portability; it provides a standard interface to services that are typically provided by an operating system or real-time kernel, isolating dependences on a particular host from the rest of GNARL [DIB94, Section 4.2.1].

The GNULL interface is an abstraction of a subset of the POSIX interfaces, including Pthreads. Therefore, it is trivially implementable over an operating system that supports the POSIX standards. In order to permit a simpler and more

efficient implementation over other operating systems, or a bare machine, many features of Pthreads have been left out or restricted. The deleted features are ones that the Ada RTS does not need, or cannot use. For example, the POSIX semantics of thread cancellation do not fit the Ada semantics of abortion, so the Pthread cancellation services are not included in GNULL. The features retained include thread creation and operations on mutexes, condition variables, and signals. In cases where the Ada RTS does not need the full strength or generality of the Pthread interface, the semantics are relaxed. For example, GNULL mutexes have only one form of priority inheritance (the priority ceiling emulation locking protocol) and are required to be unlocked in FIFO order. Condition variables are only allowed to be used by one task at a time. Further simplifications are contemplated, including the hiding of condition variables behind a suspend/resume interface [GB94a, Section 4.2.1].

1.4.3 POSIX

The POSIX Portable Operating System Interface provides an application program interface to services supporting the creation and executio¡n of multiple threads of control sharing the address space and file descriptors of a single POSIX process. POSIX has its roots in an effort to promote application program portability by establishing a non-proprietary standard interface to the many variants of the UNIX operating system [GB92, Section 2].

The IEEE identifies POSIX standards by designations of the form 1003.x. For instance, 1003.1 designates the C language application program interface for core operating systems services (i.e. file and process creation, input/output and inter-process communication). It is the base POSIX standard, and has been approved by the ISO as ISO/IEC 9945-1: 1990. Two other POSIX standards on which this project depends are 1003.4 and 1003.4a. The 1003.4 interface (Real-time Extension) is an extension to 1003.1 that provides services commonly needed in real-time applications. Examples of these services include binary semaphores, process memory locking and timers. The 1003.4a interface (Threads Extension, or Pthreads for short) is an extension to 1003.4 that supports multiple threads of control within a single POSIX process. Examples of services provided by 1003.4a include thread creation, mutual exclusion, and thread suspension. Both 1003.4 and 1003.4a are expressed as C language interfaces [GB92, Section 2]. There is also a standard Ada binding for 1003.1, namely 1003.5. This interface is defined as a set of packages, which provide access to the facilities of POSIX.1 via Ada data types, subprograms and generics [GB92, Section 3].

GNARL uses the POSIX services to build services with correct Ada semantics. The scheduling of the threads is directly under the control of the *Pthreads* scheduler, as is the state of each thread. Runtime stack allocation is also under the control of the Pthread implementation. Pthread priorities are fully dynamic, allowing the Ada RTS to make this priority adjustment in the code implementing the accept statement. Other Ada features, including the distinction between task creation and activation and the rules for task termination, are very different from their Pthread counterparts, and must be implemented almost entirely by the Ada RTS [GB92, Section 4.1]. Pthreads can be supported by the OS kernel or by a separate library. If Pthreads is supported by the OS kernel. System calls need only block the calling task, rather than the whole process. If global thread scheduling is provided, there may be better response to asynchronous events [GB92, Section 2].

1.4.4 Low-Level Locks

The GNAT run-time uses *Lock/Unlock* operations in order to maintain data consistency under concurrent read/update operations by multiple threads of control. It does quite a few more *Lock/Unlock* operations than is typical of older Ada run-time systems. The difference is that this run-time was designed to be multithreaded, whereas most earlier Ada runtime systems were designed as a monolithic monitor. That is, the older style of Ada runtime system only allowed one task to be executing in the RTS at a time (we call this single-lock mode), but with the GNAT run-time several tasks may be executing in the RTS concurrently. Rather than just one lock that protects the entire RTS, there are individual locks for several RTS global data structures, and a lock for each task control block (we call this multiple-lock mode). Multiple-lock mode allows more concurrency between tasks. According to conventional wisdom, more concurrency is generally better. It permits more parallel execution if there are multiple processors, and even if there is only one processor it may permit quicker response to high-priority real-time events [DIB94, Section 4.2.1].

Mutual exclusion is provided through POSIX mutexes. When a thread wants exclusive access to some shared resource, it locks the associated mutex, via *pthread_mutex_lock()*; if some other thread has already locked that mutex, the requesting thread is suspended until the thread holding the mutex unlocks it, via *pthread_mutex_unlock()*. Any number of tasks can be suspended on the same mutex; one of them is granted the mutex and permitted to continue execution when the holder unlocks the mutex. Mutexes are similar to binary semaphores; the principal difference is that the thread which holds the mutex must be the one to unlock it. This makes mutexes difficult to use for general communication between threads;

an arbitrary thread cannot signal to other threads that something has occurred by unlocking a locked mutex. For this kind of synchronization, condition variables are used [GB92, Section 5.1].

A thread waits for a condition to become true by calling *pthread_cond_wait()* on a condition variable. Another thread can signal that the condition has become true by signaling the condition variable, via *pthread_cond_signal()* (this is not to be confused with operations on POSIX signals). A mutex is associated with the condition variable by the *pthread_cond_wait()* call. This mutex must be locked before the call; it is unlocked (atomically) by the call and locked again before the call returns. This is to protect the condition for which the thread is waiting. A *pthread_cond_signal()* call is guaranteed to wake up at least one waiting thread, but it turns out to be more efficient (particularly on multiprocessors) to allow more than one waiting thread to return. Since the first thread to reacquire the associated mutex might make the condition false again, each thread needs to check that the condition is true when *pthread_cond_wait()* returns. This is usually done in a **while** loop [GB92, Section 5.1].

1.5 Summary

In this introductory chapter, we have seen the overall structure of the GNAT project and focussed our attention on the two main components: the Compiler and the Run-Time Library. The main concepts presented in this chapter are:

- The compiler is composed of two parts: the *front-end* and the *back-end*. The *front-end* comprises five phases which communicate by means of an *Abstract Syntax Tree*. The *back-end* is the GCC target independent code generator. This ensures two main advantages: portability and excellent-quality code generation.
- The Run-Time is implemented as an Ada Library and is structured in two levels: GNARL, and GNULL.
- GNARL is the *GNU Ada Run-Time Library*; it is written in Ada. Rather than just one lock to protect GNARL, there are individual locks for several global data structures, and a lock for each task control block (multiple-lock mode).
- GNULL exists only for portability; it provides the minimum interface to the POSIX services.

Chapter 2

Task Types and Objects

The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks [AAR95, section 9(1)].

This chapter is organized in two parts. In this first part the main concepts of the Ada tasks are presented. The second part describes the GNAT implementation.

2.1 Ada Tasks

In Ada, tasks are objects. Each task has a unique type, which is specified in an object declaration or allocator (an expression of the form "**new** ...") that causes the creation of the task. Each task type is declared in two separate parts: a task specification and a task body. The specification has a sequence of entry declarations, which define the communications interface of tasks of that type. The body has the rest of the description of the task type [BR85, Section 2]. The Ada Reference Manual defines the full syntax for a task type and body as follows:

```
task_type_declaration ::=
   task type Defining_Identifier
             [known_discriminant_part] [is task_definition];

single_task_declaration ::=
   task defining_identifier [is task_definition];

task_definition ::=
     task_item
```

```
    [ private
        task_item]
    end [task_identifier]

task_body ::=
    task body defining_identifier is
        declarative_part
    begin
        handled_sequence_of_statements
    end [task_identifier];
```

Over time, tasks proceed through various states. A task is initially inactive; upon activation, and prior to its termination it is either blocked (as part of some task interaction) or ready to run. While ready, a task competes for the available execution resources that it requires to run [AAR95, section 9].

2.1.1 Task Creation

A task type can be regarded as a template from which actual tasks are created. Task objects and types can be declared in any declarative part, including task bodies themselves. For any task type, the specification and body must be declared together in the same unit, with the body usually being placed at the end of the declarative part [BW98, chapter 4.1].

A task object can be created either as part of the elaboration of an object declaration occurring immediately within some declarative region, or as part of the evaluation of an allocator (an expression in the form "**new**..."). All tasks created by the elaboration of object declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together [AAR95, section 9].

The execution of a task object has three main active phases [BW98, chapter 4.2]:

1. *Activation* — the elaboration of the declarative part, if any, of the task body (local variables in the body of the task are created and initialized during activation). The *Activator* identifies the task which created and activated the task.

2. *Normal execution* — the execution of the statements visible within the body of the task.

3. *Finalization* — the execution of any finalization code associated with any objects in its declarative part.

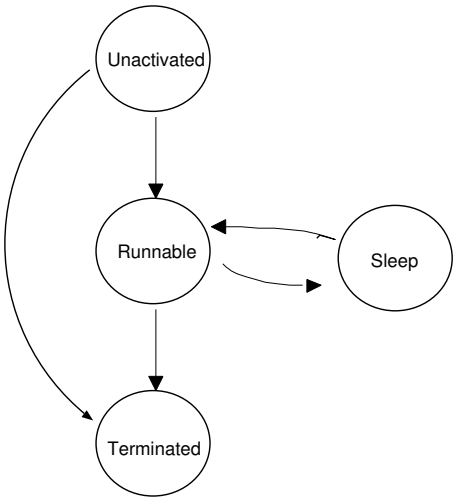

Figure 2.1: Task States.

Figure 2.1 shows the transitions among these states during task life. The created task is said to be in the *Unactivated* state. Then the run-time associates a thread of control to this task. If the elaboration of the task fails then the task goes directly to the *Terminated* state; otherwise the task reaches the *Runnable* state, and executes the task user code. If this code executes some operation that blocks the task (according to the Ada semantics— rendezvous, protected operation, or delay statement—, it reaches the *Sleep* state and later returns to the *Runnable* state. When the task executes an Ada **terminate** alternative or finalizes the execution of the Ada user code, it goes to the *Terminated* state.

A task indicates its willingness to begin finalization by executing its **end** statement. A task may also begin its finalization as a result of an unhanded exception, or by executing a **select** statement with a **terminate** alternative or by being aborted. A finished task is *Completed* or *Terminated* depending on whether it has any active dependents [BW98, chapter 4.2].

The *Parent* is the task on which a task depends. The following rules apply:

- If the task has been declared by means of an object declaration, its *Parent* is the task which declared the task object.

- If the task has been declared by means of an allocator (an Ada expression in the form '**new** ...'), its *Parent* is the task which has the corresponding access declaration.

When a parent creates a new task, the parent's execution is suspended while it waits for the child to finish activating (either immediately, if the child is created by an allocator, or after the elaboration of the associated declarative part). Once the child has finished its activation, parent and child proceed concurrently. If a task creates another task during its activation, then it must also wait for its child to activate before it can begin execution [BW98, chapter 4.3.1].

There is a conceptual task (called the *Environment Task*) which is responsible for the program elaboration. (The environment task is generally the operating system thread which initializes the run-time and executes the main Ada subprogram.) Before calling the main procedure of the Ada program, the environment task elaborates all library units referenced to in the main Ada procedure. This elaboration will cause library-level tasks to be created and activated before the main procedure is called.

2.1.2 Task Activation

The following rules apply to task activation [BW98, chapter 4.2.1]:

1. For static tasks, activation starts immediately after the complete elaboration of the declarative part in which they are defined.
2. The first statement following the declarative region is not executed until all tasks have finished their activation.
3. A task need not wait for the activation of other concurrently created tasks before executing its body.
4. A task may attempt to communicate with another task which, though created, has not yet been activated. The calling task will be delayed until the communication can take place.
5. If a task object is declared in a package specification, then it commences its execution after the elaboration of the declarative part of the package body.
6. Dynamic tasks are activated immediately after the evaluation of the allocator (the **new** operator) which created them.

7. The task which executed the Ada statement responsible for new tasks creation is blocked until these tasks have finished their activation.

8. If an exception is raised in the elaboration of a declarative part, then any task created during that elaboration becomes terminated and is never activated. As the task itself cannot handle the exception, the language model requires the parent (creator) task or scope to deal with the situation: the predefined exception *Tasking_Error* is raised.

   - In the case of dynamic task creation, the exception is raised after the statement which issued the allocator call. However, if the calls is in a declarative part (as part of the initialization of an object), the declarative part fails and the exception is raised in the surrounding block (or calling subprogram).
   - In the case of static task creation, the exception is raised prior to the execution of the first executable statement of the declarative block. This exception is only raised after all created tasks have been activated (whether successfully or not).

9. The task attribute *Callable* returns *True* if the designated task is neither *Completed*, *Terminated* nor *Callable*. (An abnormal task is one that has been aborted). The task attribute *Terminated* returns *True* if the named task has terminated.

2.1.3 Task Termination

The *Master* is the execution of a construct that includes finalization of local objects after it is complete (and after waiting for any local task), but before leaving [AAR95, section 7.6.1(1)]. Each task depends on one or more masters, as follows [AAR95, section 9.3]:

- If the task is created by the evaluation of an allocator for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.
- If the task is created by the elaboration of an object declaration, it depends on each master that includes its elaboration.

Furthermore, if a task depends on a given master, it is defined as depending on the task that executes the master, and (recursively) on any master of that task [AAR95, section 9.3].

For the *Finalization* of a master, dependent tasks are first awaited. Then each object whose accessibility level is the same as that of the master is finalized if the object was successfully initialized and still exists. Note that any object whose accessibility level is deeper than that of the master would no longer exist; those objects would have finalized by some inner master. Thus, after leaving a master, the only objects yet to be finalized are those whose accessibility level is not as deep as that of the master [AAR95, section 7.6.1(4)].

2.1.4 Task Abortion

Ada allows tasks abortion by means of the following syntax:

```
Abort_Statement ::= abort Task_Name Task_Name;
```

Tasks which are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task. After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred* operation. The execution of an abort-deferred operation is allowed to complete before it is aborted [BW98, chapter 10.2]. Task abortion will be analyzed in detail in chapter 8.

2.1.5 Task Identification

Ada tasks have a unique identifier. The Systems Programming Annex [AAR95, Annex C]. provides a mechanism by which a task can obtain its own unique identification which can be passed to other tasks [a-taside.ads]:

```
package Ada.Task_Identification is

   type Task_Id is private;
   Null_Task_Id : constant Task_Id;

   function  "=" (Left, Right : Task_Id) return Boolean;

   function Image (T : Task_Id) return String;

   function Current_Task return Task_Id;

   procedure Abort_Task (T : Task_Id);
```

```
   function Is_Terminated (T : Task_Id) return Boolean;

   function Is_Callable (T : Task_Id) return Boolean;

private
   --  Implementation defined
   . . .
end Ada.Task_Identification;
```

As well as this package, the Annex supports two attributes:

- For any prefix `T` of a task type, `T'Identity` returns a copy of the task identifier.
- For any prefix `E` that denotes an entry declaration, `E'Caller` returns the task identifier of the task whose entry call is being serviced. The attribute is only allowed inside an entry body or an accept statement.

Care must be taken when using task identifiers since there is no guarantee that, at some later time, the task will still be active or even in scope [BW98, chapter 4.4].

2.2 GNAT Implementation

Although the GNAT run-time reuses most of the support provided by the low level *Pthreads* library, it needs to handle some additional information to provide the full Ada semantics: state of the Ada task (according to the Ada semantics), parent, activator, etc. This information is stored by the run-time in a per-task register called *Ada Task Control Block (ATCB)*[1]. In addition, some task specific information is also required to store the task discriminants (the task parameters). The compiler generates code which creates another register for such information. The ATCB is linked with this register and with the corresponding *Threads Control Block (TCB)* in the POSIX level (cf. Figure 2.2).

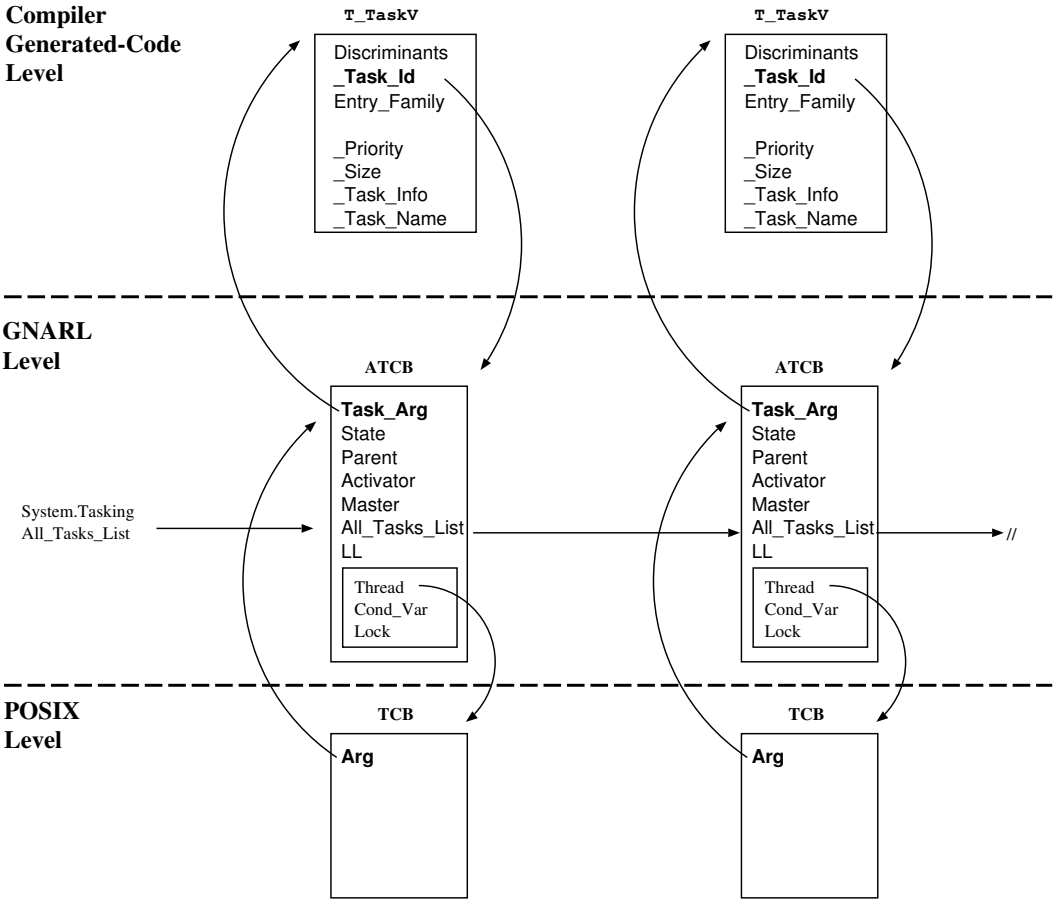

Figure 2.2: Run-Time Information Associated with Each Task.

When a task is created, the run-time dynamically generates a new ATCB and

[1] *System.Tasking.Ada_Task_Control_Block*

inserts it in a list (*All Tasks List*[2]). ATCBs are always inserted in LIFO order (as a stack). Therefore, the first ATCB in this list corresponds to the most recently created task.

2.2.1 GNAT Task States

GNAT considers four basic states during task life (indicated by the *State* ATCB field):

- *Unactivated*. The ATCB has been created and inserted in the *All Tasks List*, but no thread of control has been assigned to execute its body.
- *Runnable*. The task is executing (although it may be waiting for a mutex).
- *Sleep*. The task is blocked.
- *Terminated*: The task is terminated, in the sense of ARM 9.3 (5). Any dependents that were waiting on Ada **terminate** alternatives have been awakened and have terminated themselves.

The sleep state is composed of the following sub-states:

- *Activator_Sleep*: Waiting for created tasks to complete activation.
- *Acceptor_Sleep*: Waiting on an accept or selective wait statement.
- *Entry_Caller_Sleep*: Waiting on an entry call.
- *Async_Select_Sleep*: Waiting to start the abortable part of an asynchronous select statement.
- *Delay_Sleep*: Waiting on a select statement with only a delay alternative open.
- *Master_Completion_Sleep*: Master completion has two phases. In Phase 1 the task is sleeping in Complete_Master having completed a master within itself, and is waiting for the tasks dependent on that master to become terminated or waiting on a terminate phase.
- *Master_Phase_2_Sleep*: In phase 2 the task is sleeping in Complete_Master waiting for tasks on terminate alternatives to finish terminating.

[2] *System.Tasking.All_Tasks_List*

2.2.2 GNAT Masters Implementation

The master is a construct used to finalize local objects, including tasks (see section 2.1.3). Each master handles the termination of an Ada scope to ensure the Ada tasks termination semantics (all dependent tasks must terminate before its master performs finalization on other objects that it created). It is associated with the scope being executed by the Parent when the task was created. The run-time is only concerned with masters for purposes of task termination.

GNAT associates one identifier to each master, and two values are associated with each task: the master of its Parent (*Master_Of_Task*) and its internal master nesting level (*Master_Within*).

- *Master_Of_Task* is set to 1 for the environment task. The level 2 is reserved for server tasks of the run-time (the so called *Independent Tasks*), and the level 3 is for the library level tasks. When a task is created it inherits the internal master nesting level of its Parent (the initial value of its *Master_Of_Task* is initialized with the current value of its Parent *Master_Within*). This value remains unmodified during the new task life and is used to ensure the Ada semantics for tasks finalization.

```
New_Task.Master_Of_Task = Activator.Master_Within
```

- *Master_Within* is set to the initial *Master_Of_Task* value plus one. When the tasks enters a scope with dependent tasks, its internal nesting level is incremented to one.

Tasks created by an allocator do not necessarily depend on its activator; in such case the activator's termination may precede the termination of the newly created task [AAR95, section 9.2(5a)] Therefore, the master of a task created by the evaluation of an allocator is the declarative region which contains the access type definition. Tasks declared in library-level packages have the main program as their master. That is, the main program can not terminate until all library-level tasks have terminated [BW98, chapter 4.3.2]. Given a task T, table 2.3 presents a summary of the basic concepts used by the run-time for handling Ada task termination.

Example

In order to understand these concepts better, let's apply them to the following example.

Parent	The task executing the master on which T depends.
Activator	The task that created T's ATCB and activated it.
Master of Task	Parent's scope on which T depends.
Master Within	Nesting level of T dependent tasks.

Figure 2.3: Definition of Parent, Activator, Master of Task and Master Within.

```
procedure P is      --  P:  Parent = Environment Task;
                    --      Activator = Environment
                    --      Master_Of_Task = 1; Master_Within = 2;

   task T1;                    --  T1: Parent = P; Activator = P
                               --      Master_Of_Task = 2; Master_Within = 3;
   task body T1 is

      task type TT;
      task body TT is
      begin
         null;
      end TT;

      type TTA is access TT;
      T2 : TT;                 --  T2: Parent = T1; Activator = T1
                               --      Master_Of_Task = 3; Master_Within = 4;

      task T3;                 --  T3: Parent = T1; Activator = T1
                               --      Master_Of_Task = 3; Master_Within = 4;
      task body T3 is
         task T4;              --  T4: Parent = T3; Activator = T3
                               --      Master_Of_Task = 4; Master_Within = 5;
         task body T4 is
         begin
            null;
         end T4;
         T5 : TT;              --  T5: Parent = T3; Activator = T3
                               --      Master_Of_Task = 4; Master_Within = 5;
         T6 : TTA := new TT;   --  T6: Parent = T1; Activator = T3
                               --      Master_Of_Task = 2; Master_Within = 3;
      begin
         null;
      end T3;

   begin
      null;
   end T1;

begin
   null;
end P;
```

Parent and activator do not coincide in T6 because the task is created by means

of an Ada allocator (an Ada expression in the form '**new** ...'). In this case the parent of the new task is the task where the type is declared and the activator is the task which executes the allocator. In the other cases, parent and activator coincide.

2.2.3 Compiler Task Translation

In order to understand the run-time behavior we first present the task translation done by the compiler.

Task Specification

The Ada task type is translated by the compiler into a limited record with the same discriminants. For example, the following task specification:

```
task type T_Task (Discriminant : DType) is
   ...
end T_Task;
```

... is translated by the compiler into the following code:

```
T_TaskE : aliased Boolean := False;
T_TaskZ : Size_Type       := [ Unspecified_Size |
                               Size_Type (Size_Expression) ];
type T_TaskV [ (Discriminant : DType) ] is
      limited record
         _Task_Id : System.Tasking.Task_Id;
         [ Entry_Family : array (Bounds) of Void; ]
         [ _Priority    : Integer        := Priority_Expression;    ]

         [ _Size        : Size_Type      := Size_Expression;        ]
         [ _Task_Info   : Task_Info_Type := Task_Info_Expression;   ]
         [ _Task_Name   : Task_Image_Type : new String'(Task_Name); ]
      end record;
```

The optional code (the code that it is not always generated by the compiler) has been put between square brackets (`[...]`). First, a boolean flag `E` is declared and initialized to false. It is set to *True* when the body of the task is elaborated. The `Z` variable holds the task stack size (either the default value, *unspecified_size*, or the value set by a pragma *Storage_Size*). Next the task type is translated by the compiler into a limited record `V` with *Discriminants* present only if the corresponding task type has discriminants. The first field contains the *Task_ID*[3] value

[3] *System.Tasking.Task_ID*

(an access to the corresponding ATCB). One *Entry_Family* component is present for each entry family in the task definition. The bounds correspond to the bounds of the entry family (which may depend on discriminants). Since the run-time only needs such information for determining the entry index the element type is void. The next three fields are present only if the corresponding pragma is present in the task definition: the *Size* field corresponds to *Storage_Size* pragma; *Task_Info* corresponds to *Task_Info* pragma, and *Task_Name* corresponds to *Task_Name* pragma. A reference to this record is stored in the *Task_Arg*[4] ATCB field (cf. Figure 2.4). This reference is used by the thread associated with the task to find the task discriminants.

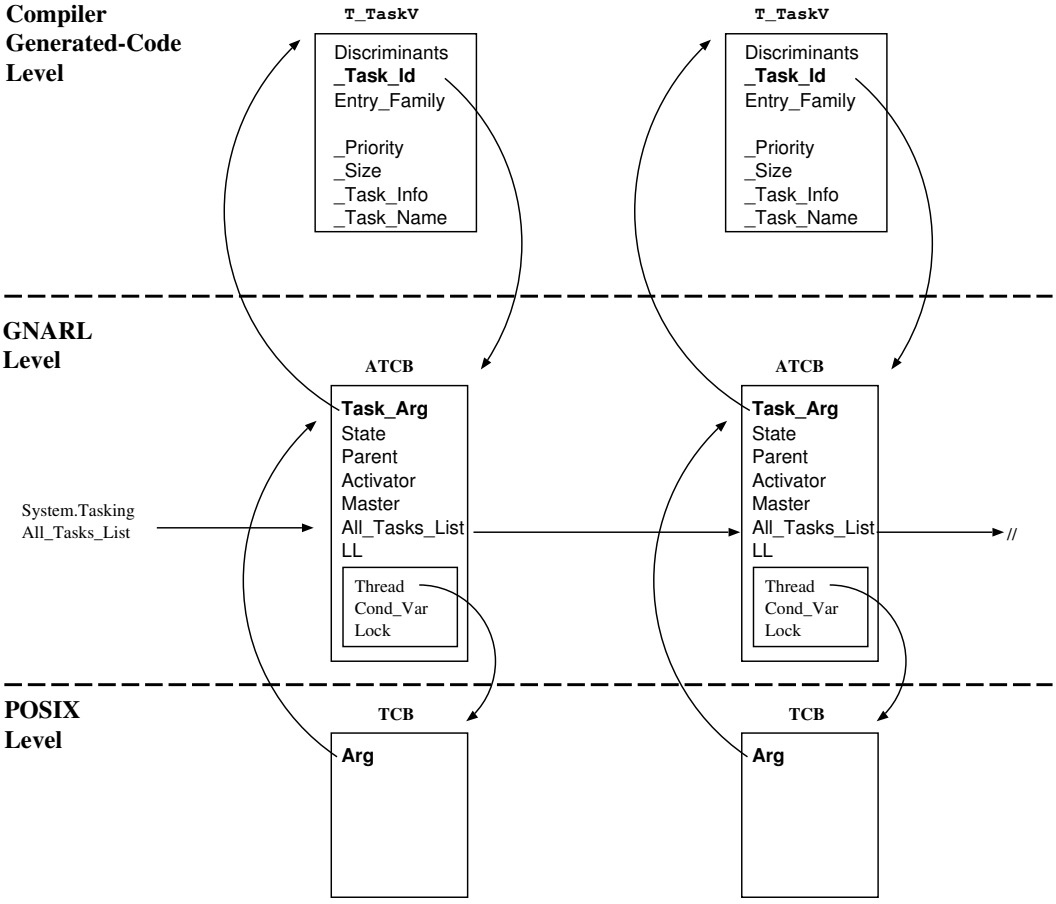

Figure 2.4: Compiler-Generated Information Associated with Each Task.

[4] *System.Tasking.Task_Arg*

Task body

The run-time needs an access to subprogram to call the task user code. Therefore the compiler translates the task body into a procedure. For example, the following task body:

```
task body T_Task is
   <Declarations>
begin
   <Statements>
end T_Task;
```

...is translated by the compiler into the following code:

```
 1: procedure T_TaskB (_task : access T_TaskV) is
 2:   Discriminant : Dtype renames _task.Discriminant;
 3:
 4:   procedure _Clean is
 5:   begin
 6:      Abort_Defer;
 7:      GNARL.Complete_Task;
 8:      Abort_Undefer;
 9:   end _Clean;
10:
11:  begin
12:     Abort_Undefer;
13:     <Declarations>
14:     [ Activate_Tasks ]
15:     GNARL.Complete_Activation;
16:     <Statements>
17:  at end
18:     _Clean;
19:  end T_TaskB;
```

The call to *Activate_Tasks*[5] (line 14) is only generated if the task body is an activator. The **at end** handler is a single point of task finalization that is called even in the presence of exceptions or task abortion [BG94, section 6.9.5].

2.2.4 Run-Time Subprograms for Task Creation and Termination

Figure 2.5 presents the sequence of calls to the run-time issued by the compiler generated code during the creation and finalization of a task. Each rectangle represents a subprogram.

[5] *System.Tasking.Stages.Activate_Tasks*

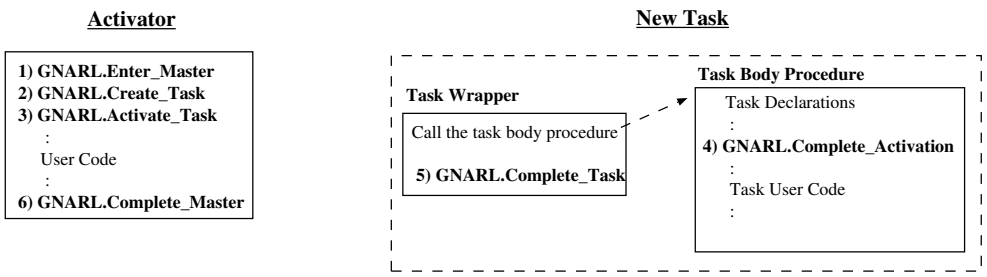

Figure 2.5: GNARL Subprograms Called During the Task Life-Cycle

The whole sequence is as follows:

1. *Enter_Master* is called in the Ada scope where the task or access type designating objects containing tasks is declared.

2. *Create_Task* is called to create the ATCBs of the new tasks and to insert it in the all tasks list and in the activation chain (see section 2.2.4).

3. *Activate_Tasks* is called to create new threads and to associate them to the new ATCB (the ATCBs in the activation chain). When all the threads have been created the activator becomes blocked until they complete the elaboration of their declarative part.

The thread associated with the new task executes a *Task_Wrapper* procedure. This procedure has some locally declared objects that serve as per-task run-time local data. The *Task Wrapper* calls the *Task Body Procedure* (the procedure generated by the compiler which has the task user code) which elaborates the declarations within the task declarative part, setting up the local environment in which it will later execute its sequence of statements. (In general the compiler must generate code for the elaboration of Ada declarations.) Note that if these declarations also have task objects, then there is a chained activation: this task becomes the activator of dependent task objects and can not start the execution of its user code until all dependent tasks complete their activation.

4. *Complete_Activation* is called when the new thread completes the elaboration of all the task declarations, but before executing the first task body sentence. This call is used to signal to the activator that it need no longer wait for this task to finish activation. If this is the last task which completes its activation, the activator becomes unblocked.

From here the activator and the new tasks proceed concurrently and their execution is controlled by the POSIX scheduler. Afterward, any of them can terminate their execution and therefore the following two steps can be interchanged.

5. *Complete_Task* is called when the task terminates its execution. Even though a completed task cannot execute any more, it is not yet safe to deallocate its working storage at this point because some reference may still be made to the task. In particular, it is possible for other tasks to still attempt entry calls to a terminated task, to abort it, and to interrogate its status via the *'Terminated* and *'Callable* attributes. Nevertheless, completion of a task requires action by the run-time. The task must be removed from any queues on which it may happen to be, and must be marked as completed. A check must be made for pending calls on entries of the completed task, and the exception *Tasking_Error* must be raised in any such calling tasks [BR85, Section 4].

6. *Complete_Master* is called by the activator when it finishes the execution of this scope. At this point the activator waits until all its dependent tasks either complete their execution (and call *Complete_Task*) or are blocked in a *Terminate* alternative. Alive dependent tasks in a **terminate** alternative are forced to terminate.

   In general this is the earliest point at which it is completely safe to discard all storage associated with its dependent tasks (because it is at this point that execution leaves the scope of the task's type declaration). This is so because reference to a task may be passed far from its point of creation, as via task access variables and functions returning task values [BR85, Section 4].

In the following sections we make a detailed description of the work done by the following run-time subprograms: Enter Master, Create Task, Activate Tasks, Complete Activation, Complete Task, and Complete Master.

1. Enter Master

Enter_Master[6] just increments the current value of *Master_Within* in the activator.

[6] *System.Tasking.Stages.Enter_Master*

2. Create Task

Create_Task[7] carries out the following actions:

1. If no priority was specified for the new task then assign to it the base priority of the activator.

   The priority of a task where no priority is specified is the priority at which it was created, that is, the activator priority at the time that it calls *Create_Task* [BG94, section 6.7.5]

2. Traverse the parents list of the activator to look for the parent of the new task via the master level (the Parent Master is lower than the master of the new task).

3. Defer the abortion.

4. Request dynamic memory for the new ATCB[8].

5. Lock *All_Tasks_List* because this lock is used by Abort_Dependents and Abort_Tasks and, up to this point, it is possible that the new task is part of a family of tasks that is being aborted.

6. Lock the Activator's ATCB.

7. If the Activator has been aborted then unlock the previous locks (*All_Tasks_Lists* and its ATCB), undefer the abortion and raise the *Abort_Signal* internal exception.

8. Initialize all the fields of the new ATCB[9] (*Callable* set to True; *Wait_Count*, *Alive_Count* and *Awake_Count* set to 0).

9. Unlock the Activator's ATCB.

10. Unlock *All_Tasks_List*.

11. Add some data to the new ATCB to manage exceptions[10].

12. Insert the new ATCB in the activation chain.

13. Initialize the structures associated with the task attributes.

14. Undefer the abortion.

[7] *System.Tasking.Stages.Create_Task*
[8] *System.Task_Primitives.Operations.New_ATCB*
[9] *System.Tasking.Initialize_ATCB*
[10] *System.Soft_Links.Create_TSD*

From this point the new task becomes callable. When the call to this run-time subprogram returns the code generated by the compiler executes a sentence which sets to *True* the variable which reminds that the task has been elaborated (described in section 2.2.3).

3. Activate Tasks

With respect to the tasks activation the Ada Reference Manual says that *"all tasks created by the elaboration of object_declarations of a single declarative region (including subcomponents of the declared objects) are activated together. Similarly, all tasks created by the evaluation of a single allocator are activated together."* [AAR95, section 9.2(2)]

GNAT uses an auxiliary list (the *Activation List*) to achieve this semantics. In a first stage all the ATCBs are created and inserted in the two lists (*All Tasks* and *Activation* lists); in a second stage the Activation List is traversed and new threads of control are created and associated with the new ATCBs. Although ATCBs are inserted in both lists in LIFO order (as a stack) all activated tasks synchronize on the activators lock before they start their activation in priority order. The activation chain is not outstanding when all its tasks have been activated.

Activate_Tasks[11] performs the following actions:

1. Defer abortion.
2. Lock *All_Tasks_List* to prevent activated tasks from racing ahead before we finish activating all tasks in the *Activation Chain*.
3. Check that all task bodies have been elaborated. Raise *Program_Error* otherwise.

   For the activation of a task, the activator checks that the task_body is already elaborated. If two or more tasks are being activated together (see ARM 9.2), as the result of the elaboration of a declarative_part or the initialization for the object created by an allocator, this check is done for all of them before activating any.

   Reason: As specified by AI-00149, the check is done by the activator, rather than by the task itself. If it were done by the task itself, it would be turned into a Tasking_Error in the activator, and the other tasks would still be activated [AAR95, section 3.11(12)].

[11] *System.Tasking.Stages.Activate_Tasks*

4. Reverse the activation chain so that tasks are activated in the order they were declared. This is not needed if priority-based scheduling is supported, since activated tasks synchronize on the activators lock before they start activating and so they should start activating in priority order.

5. For all tasks in the activation chain do the following actions:

   (a) Lock the task's parent.

   (b) Lock the task ATCB.

   (c) If the base priority of the new task is lower than the activator priority, raise the priority to the activator priority, because a task being activated inherits the active priority of its activator [AAR95, section D.1(21)].

   (d) Create a new thread by means of `GNULL` call[12] and associates it to the task wrapper. If the creation of the new thread fails, release the locks and set the caller ATCB field Activation_Failed to *True*.

   (e) Set the state of the new task to *Runnable*.

   (f) Initialize the counters of the new task (*Await_Count* and *Alive_Count* set to 1)

   (g) Increment the parent counters (*Await_Count* and *Alive_Count*).

   (h) If the parent is completing the master associated with this new task, increment the number of tasks that the master must wait for (*Wait_Count*).

   (i) Unlock the task ATCB.

   (j) Unlock the task's parent.

6. Lock the caller ATCB.

7. Set the activator state to *Activator Sleep*

8. Close the entries of the tasks that failed thread creation, and count those that have not finished activation.

9. Poll priority change and wait for the activated tasks to complete activation. While the caller is blocked POSIX releases the caller lock.

   Once all of these activations are complete, if the activation of any of the tasks has failed (due to the propagation of an exception), Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any task aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error [AAR95, section 9.2(5)].

[12] *System.Task_Primitives.Operations.Create_Task*

10. Set the activator state to *Runnable*.
11. Unlock the caller ATCB.
12. Remove the Activation Chain.
13. Undefer the abortion.
14. If some tasks activation failed then raise *Program_Error*. Tasking_Error is raised only once, even if two or more of the tasks being activated fail their activation [AAR95, section 9.2(5b)].

4. Complete Activation

Complete_Activation[13] is called by each task when it completes the elaboration of its declarative part. It carries out the following actions:

1. Defer the abortion.
2. Lock the activator ATCB.
3. Lock self ATCB.
4. Remove dangling reference to the activator (since a task may outline its activator).
5. If the activator is in the *Activator_Sleep* State then decrement *Wait_Count* in the activator. If this is the last task to complete the activation in the Activation Chain, wake up the activator so it can check if all tasks have been activated.
6. Set the priority to the base priority value.
7. Undefer the abortion.

5. Complete Task

The Complete_Task[14] subprogram performs the following single action:

1. Cancel queued entry calls.

[13] *System.Tasking.Stages.Complete_Activation*
[14] *System.Tasking.Stages.Complete_Task.*

From this point the task becomes not callable.

6. Complete Master

The run-time subprogram Complete_Master[15] carries out the following actions:

1. Traverse all ATCBs counting how many active dependent tasks does this master currently have (and terminate all the still unactivated tasks). Store this value in *Wait_Count*.
2. Set the current state of the activator to *Master_Completion_Sleep*.
3. Wait until dependent tasks are all terminated or ready to terminate.
4. Set the current state of the activator to *Runnable*.
5. Force those tasks on terminate alternatives to terminate (by aborting them).
6. Count how many *active* dependent tasks does this master currently have. Store this value in *Wait_Count*.
7. Set the current state of the activator to *Master_Phase_2_Sleep_State*.
8. Wait for all counted tasks to terminate themselves.
9. Set the current state of the activator to *Runnable*.
10. Remove terminated tasks from the list of dependents and free their ATCB.
11. Decrement *Master_Within*

2.3 Summary

In this chapter we have seen the basic data structures used by the GNAT run-time to support Ada tasks, the task states considered by GNARL, the task translation done by the compiler, and the subprograms called by this generated code. In summary, the main aspects covered in this chapter are:

1. Each task has an associated Ada Task Control Block (ATCB).

[15] *System.Tasking.Stages.Complete_Master*

2. There is a list which contains all the ATCBs (*All Tasks List*).

3. One auxiliary list is used to activate task objects in the same Ada scope at the same time.

4. Masters define a task scope which allow the run-time to manage task finalization.

5. The Ada task specification is translated by the compiler into a limited record; the Ada task body is translated into a procedure with intermixed calls to the RTS to manage the task body creation, activation and finalization.

6. The environment task is responsible for the RTS initialization. After this work, it also executes the main Ada subprogram.

Chapter 3

The Rendezvous

The *Rendezvous* is the basic mechanism for synchronization and communication of Ada tasks. The model of Ada is based on a client/server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). It does this by declaring one or more public *entries* in its task specification. A rendezvous is requested by one task making an entry call on an entry of another task. For the rendezvous to take place the called task must accept this entry call. During the rendezvous the calling task waits while the accepting task executes. When the accepting task ends the rendezvous both tasks are freed to continue their execution [BR85, Section 6].

This chapter is organized in two parts. In this first part the Ada rendezvous model and syntax are presented. The second part describes the GNAT implementation.

3.1 The Ada Rendezvous

3.1.1 Entry Declaration

Each entry identifies the name of the service, the parameters that are required with the request and the results that will be returned [BW98, section 5.1]. The Ada Reference Manual defines the syntax as follows [AAR95, section 9.5.2]:

```
entry_declaration ::=
   entry defining_identifier
         [(discrete_subtype_definition)] parameter_profile;
```

The entry declaration must be placed inside the task specification. The parameter profile is the same for Ada procedures (**in**, **out** and **in out** — with **in** being the default). Access parameters are not permitted, though parameters of any access type are, of course, allowed. Default parameters are also allowed. The optional *Discrete_Subtype_Definition* in the entry declaration is used to declare a family of entries (an array of entries), all of which will have the same formal part. Similar to procedures, entries can be overloaded. This means that a task can have more than one entry with the same name if the parameters to the entries are different [BW98, section 5.2].

3.1.2 Simple Mode Entry Call

A client task (also referred to as *calling task*) issues an "entry call" on the server task by identifying both the server and the required entry [BW98, section 5.1].

```
entry_call_statement ::= entry_name [actual_parameter_part];
```

As the reader can see, a simple mode entry call is much like a procedure call. It may have parameters, which permit values to be passed in both directions between the calling and accepting tasks. Semantically the calling task is blocked until completion of the requested rendezvous. If the call is completed normally, it resumes execution with the statement following the call, just as it would after return from a procedure call. Recovery from any exception raised by the call is also treated as it would be for a procedure call. One minor difference detectable by the calling task is that an entry call may result in *Tasking_Error* being raised in the calling task, whereas an ordinary procedure call would not [BR85, Section 6].

3.1.3 Conditional Entry Calls

The conditional entry call allows the task client to withdraw the offer to communicate if the server task is not prepared to accept the call immediately [BW98, section 6.9.2]. The syntax is [AAR95, section 9.7.3]:

```
conditional_entry_call ::=
   select
      entry_call_alternative
   else
      sequence_of_statements
```

```
      end select;

   entry_call_alternative ::=
      entry_call_statement [sequence_of_statements]
```

3.1.4 Accept Statement

The server task indicates a willingness to provide the service at any particular time by executing an "accept" statement [BW98, section 5.1]. The syntax is [AAR95, section 9.5.2]:

```
accept_statement ::=
   accept entry_direct_name [(entry_index)] parameter_profile [do
     handled_sequence_of_statements
   end [entry_identifier]];
```

The accept statement specifies the actions to be performed when the entry is called. It must be placed in the task body; it can not be placed in a procedure which is called by the task body [BW98, section 5.3]. For the communication to occur between the client and the server, both tasks must have issued their respective requests. When they have, the communication takes place; this is called the *rendezvous* because both tasks have to meet at the entry at the same time. When the rendezvous occurs, any **in** (and **in out**) parameters are passed to the server task from the client. The server task then executes the code inside the **accept** statement. When this statement finishes, **out** (and **in out**) parameters are passed back to the client and both tasks proceed independently and concurrently [BW98, section 5.1].

It is quite possible that the client and server will not both be in a position to communicate at exactly the same time. For example, the operator may be willing to accept a service request but there may be no subscribers issuing an entry call. For the simple rendezvous case, the server must wait for a call; whilst it is waiting it frees up any processing resource it is using; a task which is generally waiting for some event to occur is usually termed *suspended* or *blocked*. If a client issues a request and the server is not ready to accept the request (either because it is already servicing another request or it is doing something else), then the client must wait. Clients waiting for service at a particular entry are queued. The order of the queue depends on whether the Ada implementation supports the Real-Time Systems Annex [AAR95, Annex D]. If the annex is not supported, then the queue is first-in-first-out; otherwise other possibilities are allowed including priority queuing [BW98, section 5.1].

If an exception is raised during the rendezvous, then the rendezvous is terminated and the exception is raised again in *both* the server (called) and the client (calling) task. In addition, when a task attempts to call another task that has already terminated (completed or has become abnormal), the caller task gets the exception *Tasking_Error* raised at the point of the call [BW98, section 5.7].

3.1.5 Selective Accept

The "selective accept" allows a server task to [BW98, section 6.1]:

- wait for more than a single rendezvous at any time;
- time-out if no rendezvous is forthcoming within a specified period (see section 5.1.5);
- withdraw its offer to communicate if no rendezvous is immediately available;
- terminate if no clients can possibly call its entries.

The syntax of the selective accept is [AAR95, section 9.7.1]:

```
selective_accept ::=
  select
      [guard]
      select_alternative
 or
      [guard]
      select_alternative
[ else
    sequence_of_statements ]
  end select;

guard ::= when condition =>

select_alternative ::=
   accept_alternative
   | delay_alternative
   | terminate_alternative

accept_alternative ::=
   accept_statement [sequence_of_statements]

delay_alternative ::=
   delay_statement [sequence_of_statements]

terminate_alternative ::= terminate;
```

A selective accept must contain at least one accept alternative. It is possible that several clients may be waiting on one or more of the entries when the server task executes the select statement. In this case, the one chosen is implementation dependent. This means that the language itself does not define the order in which the requests are serviced. If the implementation is supporting the Real-Time Systems Annex [AAR95, section D], then certain orderings can be defined by the programmer. For general-purpose concurrent-programming, the programmer should assume that the order is arbitrary; that way the program cannot make any assumptions about the implementation of the language and thus it will be portable across different implementation approaches. By default, single queues are serviced on a first-come first-served basis [BW98, section 6.1].

Each selective accept alternative can have a *guard* associated with it. This guard is a boolean expression which is evaluated when the select statement is executed. If the expression evaluates to *True*, the alternative is eligible for selection. If it is *False*, then the alternative is not eligible for selection during this execution of the select statement, even if clients are waiting on the associated entry. It is an error if a selective accept statement has a guard on each of its alternatives and all the guards evaluate to *False*. When this happens, the exception *Program_Error* is raised. Alternatives without guards are deemed to have "true" guards [BW98, section 6.2].

The **else** part allows the server to withdraw its offer to communicate if no call is immediately available. The else part can not be guarded and consequently only one else part may appear in a single select statement [BW98, section 6.4].

The **delay** alternative of the selective accept allows a server task to time-out if an entry call is not received within a certain period of time. The time-out can be a relative or an absolute delay. If the relative time expressed is zero or negative, or the absolute time has passed, then the delay alternative is equivalent to having an "else part". More than one delay alternative is allowed, though only the delay with the smallest time interval will act as the time-out. Relative and absolute delay alternatives can not be mixed in a single select statement [BW98, section 6.3]. Timed sentences will be analyzed in chapter 5.

The **terminate** alternative allows a server task to become completed when the following conditions are satisfied [BW98, section 6.6]:

- The task depends on some master whose execution is completed (the concept of the "master" is explained in section 2.1.3)
- Each task which depends on the master considered is either already terminated or similarly blocked at a select statement with an open terminate

alternative.

When both the above conditions are satisfied, not only is the server task completed but so also are all tasks that depend on the master being considered. Once these tasks are completed any associated finalization code is executed [BW98, section 6.6].

3.1.6 The Count Attribute

Each entry queue has an attribute associate with it that allows the current length of the queue to be accessible to the owning task. `E'Count` returns a natural number representing the number of entry calls currently on the entry `E`, where `E` is either a single entry or a single entry of a family [BW98, section 5.3.1]. The *Count* attribute can only be used within the body of the task but not within a dependent subprogram.

3.2 GNAT Implementation

Achieving rendezvous ordinarily requires that one of the two tasks wait until the other is ready. In the case that more than one task is waiting on the same entry of a task, Ada requires the calls be accepted in first-in-first-out order. An implementation must therefore maintain data structures to keep track of which tasks are waiting on entry calls, which entries they are calling, and in what order the calls on each entry of a task arrived [BR85, Section 6].

3.2.1 Entry Call and Parameters

GNAT associates a record to each entry call: the *Entry Call Record*[1]. It is used to group all the run-time information associated with the entry call. It includes the identifier of the called entry, the current state of the entry call, the links to the previous and next queued entry calls, etc.

The compiler generates one record with one field associated with each entry parameter: the *Entry Parameters Record*. The compiler also generates code which fills these fields with the address of the corresponding parameter. (In case of simple Ada types—Integer, Float, enumeration, etc.— the compiler generates code which declares local variables, copies the real parameter in these variables and stores the address of these variables in the corresponding field of the *Entry Parameters Record*). The address of the *Entry Parameters Record* is then passed to the GNAT run-time. The run-time stores the address of the *Entry Parameters Record* in the *Uninterpreted_Data*[2] field of the *Entry Call Record*.

As a summary, Figure 3.1 presents the GNAT run-time data structures used to handle an entry call to the entry `E` of the following task specification:

```
task T is
   entry E (Number : in Integer; Text : in String);
end T;
```

3.2.2 Simple Mode Entry Call

Due to the similarity of the simple mode entry call and the procedure call, the compiler translates a simple mode entry call into a procedure call.

[1] *System.Tasking.Entry_Call_Record*
[2] *System.Tasking.Uninterpreted_Data*

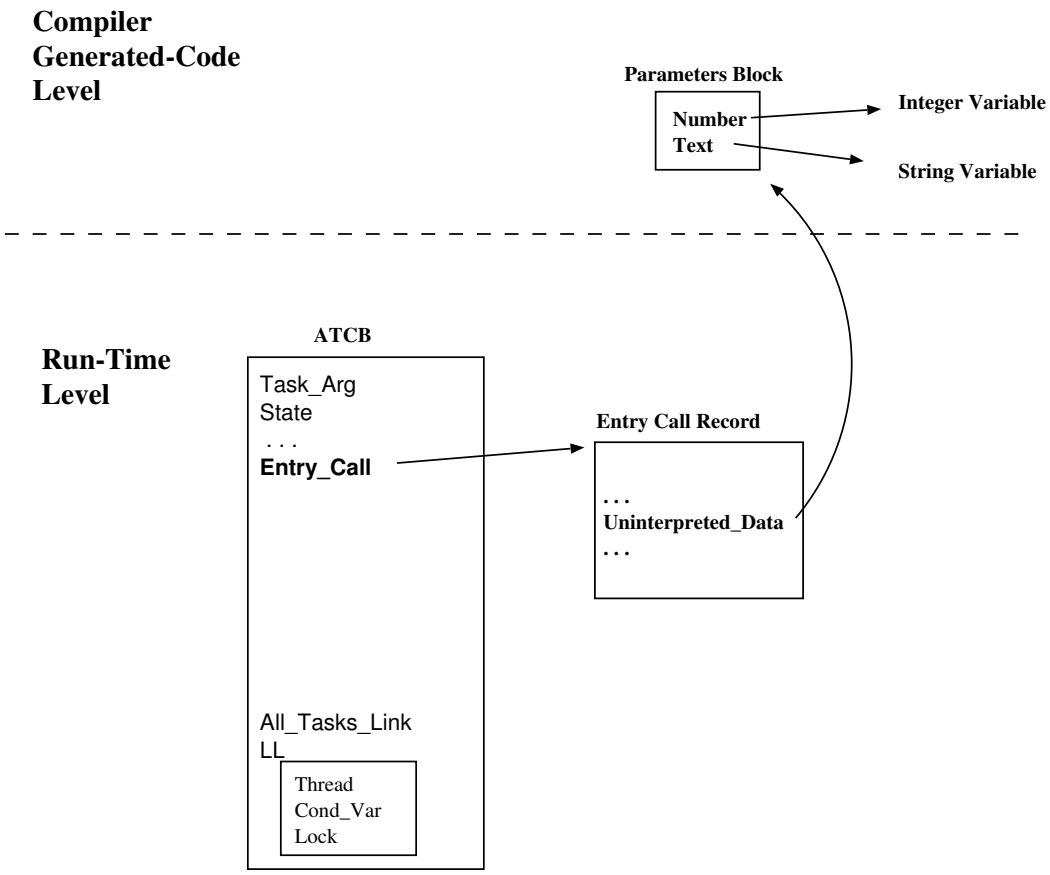

Figure 3.1: Entry Call.

```
declare
   P : Parms_Block := (Parm1, Parm2, ..., ParmN);
begin
   GNARL.Call_Simple (Task_ID, Entry_ID, P'Address);
   [ Parm1 := P.Parm1; ]
   [ Parm2 := P.Parm2; ]
   [ ...               ]
end;
```

`P` is an aggregate which saves the parameters (the *Entry Parameters Record* described in section 3.2.1). The address of this aggregate is passed to the GNAT run-time, along with the identifiers of the called task and entry. The assignments after the call are present only in the case of **in out** or **out** parameters for elementary types, and are used to assign back the resulting values of such parameters. Let's see the actions performed by the run-time.

- **Call_Simple**[3]:

  1. Call the GNARL subprogram *Call_Synchronous*

- **Call_Synchronous**[4]:

  1. Defer the abortion.
  2. Create and elaborate a new *Entry Call Record* and associate to it the entry call parameters (the *Entry Parameters Record*).
  3. Call the GNARL subprogram *Task_Do_Or_Queue*.
  4. Wait for the completion of the rendezvous (*Wait_For_Completion*[5]).
  5. Undefer the abortion.
  6. Raise any pending exception from the entry call (*Check_Exception*[6]).

- **Task_Do_Or_Queue**[7]:

  1. Try to serve the call immediately. If the acceptor is accepting some entry call and the current call can be accepted the following actions are done:
     (a) Commit the acceptor to rendezvous with the caller.
     (b) If the acceptor is in a **terminate** alternative then cancel the terminate alternative. If the acceptor has no dependent tasks notify its parent that the acceptor is again awake.
     (c) If the **accept** statement has a null body (an accept used for tasks synchronization) then wake up the acceptor, wake up the caller and RETURN.
     (d) If the **accept** statement has some body then call a run-time procedure (*Setup For Rendezvous With Body*[8]) to insert the *Entry Call Record* in the *Accepted Entry Calls Stack* of the acceptor task (described in section 3.2.6), and to raise the priority of the acceptor (if the caller priority is higher than the priority of the acceptor). Then wake up the acceptor and RETURN.

[3] *System.Tasking.Rendezvous.Call_Simple*
[4] *System.Tasking.Rendezvous.Call_Synchronous*
[5] *System.Tasking.Entry_Calls.Wait_For_Completion*
[6] *System.Tasking.Entry_Calls.Check_Exception*
[7] *System.Tasking.Rendezvous.Task_Do_Or_Queue*
[8] *System.Tasking.Rendezvous.Setup_For_Rendezvous_With_Body*

3.2.3 Conditional Entry Call

The efficient implementation of the conditional entry call requires to check if the called task is ready to accept the call. If the test fails, the GNAT run-time returns control immediately to the calling task. Otherwise, the actions are similar to those for the simple mode call [BR85]. The compiler translates the conditional entry call into the following code:

```
declare
   P          : Parms_Block := (Parm1, Parm2, ..., ParmN);
   Successful : Boolean;
begin
   GNARL.Task_Entry_Call (Task_ID,
                          Entry_ID,
                          P'Address,
                          Successful);
   if Successful then
      [ Parm1 := P.Parm1; ]
      [ Parm2 := P.Parm2; ]
      [ ...               ]
      Statements; --  Statements after the entry call
   else
      Statements; --  Statements in the "else" part
   end if;
end;
```

In this case the code generated by the compiler calls the GNARL subprogram *Task_Entry_Call*[9] with the same parameters of the simple mode entry call and one additional **out** mode parameter (*Successful*). If the entry call is immediately accepted this parameter is set to *True* by the run-time, and the statements after the entry call are executed. Otherwise, the statements in the else part are executed.

3.2.4 Entries and Queues

The GNAT compiler associates a positive number to each task entry: the *Entry Identifier*. This number corresponds with the position of the entry in the task type specification (starting with 1). Families of entries are handled like individual entries. For example, the following task has five entries: a single entry (*Hello*), a family of entries (*Do_Work*) and another single entry (*Bye*). Identifier 1 is assigned to *Hello*; identifiers 2 to 4 are assigned to the entry family *Do_Work*, and identifier 5 is assigned to *Bye*.

[9] *System.Tasking.Rendezvous.Task_Entry_Call*

```
task T is
   entry Hello (A : in Integer);
   --  Hello       Entry_Id = 1

   entry Do_Work(1..3) (B : in Integer);
   --  Do_Work (1) Entry_Id = 2
   --  Do_Work (2) Entry_Id = 3
   --  Do_Work (3) Entry_Id = 4
   entry Bye;
   --  Bye         Entry_Id = 5
end T;
```

Each entry has one queue which stores all the pending entry calls. If the queue is nonempty, the next caller to be served is at the head of the queue. The GNARL implementation uses circular doubly linked lists so that checking, insertion and deletion are all constant-time operations.

The ATCB field *Entry_Queues* is an array indexed by the entry identifier. Each element of this array has two fields: the *Head* and the *Tail* of the queue.

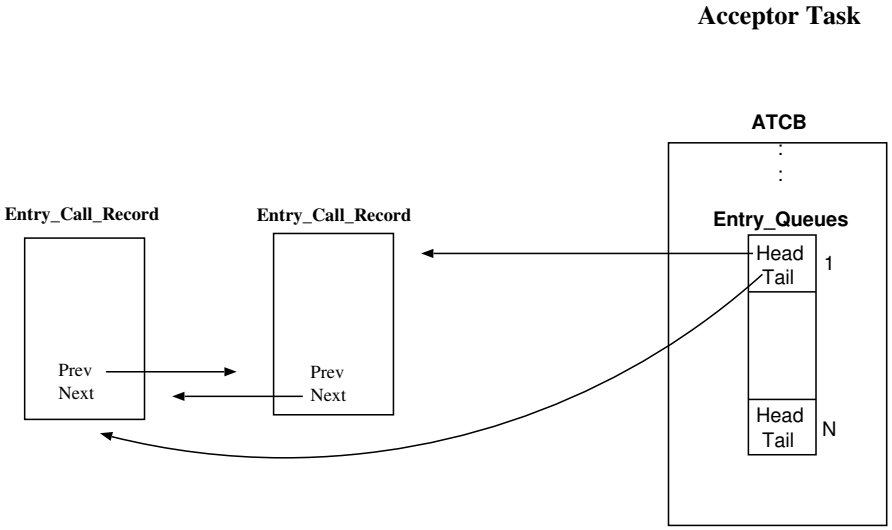

Figure 3.2: Entry Queues.

3.2.5 Trivial Accept

The GNAT run-time classifies the Ada accept sentences into the following modes: *Trivial* (accept without parameters and without code which is used to synchronize Ada tasks), *Simple* (accept with parameters or code) and *Selective* (the Ada se-

lective accept sentence). The trivial accept corresponds with the following Ada code:

```
accept My_Entry;
```

The compiler translates this sentence into the following GNARL call:

```
GNARL.Accept_Trivial (Entry_ID);
```

This GNARL procedure performs the following actions:

- **Accept_Trivial**[10]:
  1. Defer the abortion.
  2. If no entry call is still queued then block the acceptor task to wait for the next entry call (*Wait_For_Call*[11]).
  3. Extract the *Entry Call Record* from the head of the queue (*Dequeue Head*[12]) and wake-up the entry caller (*Wakeup_Entry_Caller*[13]).
  4. Undefer the abortion.

3.2.6 Accept Statement

When the accept has some code the GNAT run-time extracts the *Entry Call Record* from the entry queue and pushes it in an *Accepted Entry Calls Stack*. The top of this stack is referenced by the *Call* field of the acceptor's ATCB (cf. Figure 3.3).The *Entry Call Records* in this stack are linked by means of the *Acceptor_Prev_Call* field. All the entry calls in this stack correspond to nested accept statements executed by the acceptor task.

The simple mode accept corresponds with the following Ada code:

```
accept My_Entry ( . . . ) do
     << Entry_Body >>
end My_Entry;
```

[10] *System.Tasking.Rendezvous.Accept_Trivial*
[11] *System.Tasking.Rendezvous.Wait_For_Call*
[12] *System.Tasking.Queuing.Dequeue_Head*
[13] *System.Tasking.Initialization.Wakeup_Entry_Caller*

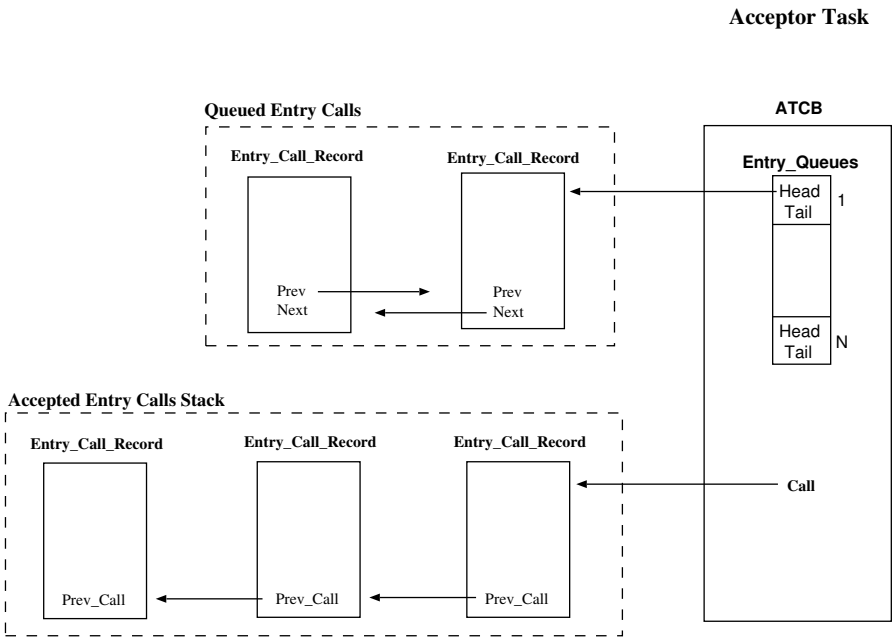

Figure 3.3: Simple Accept.

It is translated by the compiler to the following code:

```
declare
   Params_Block_Address : Address;
begin
   GNARL.Accept_Call (Entry_ID, Params_Block_Address);
   << Entry_Body >>
   GNARL.Complete_Rendezvous;
exception
   when others =>
        GNARL.Exceptional_Complete_Rendezvous;
end;
```

The local variable *Params_Block_Address* is used to store the address of the *Entry Parameters Record*. The user code is put by the compiler in middle of two calls to GNARL. The GNARL procedure *Accept_Call* carries out the following actions.

- **Accept_Call**[14]:

  1. Defer the abortion.

[14] *System.Tasking.Rendezvous.Accept_Call*

2. If the entry has no queued entry calls then block the acceptor tasks to wait for the next entry call (*Wait_For_Call*[15]).
3. Extract the *Entry Call Record* from the head of the queue (*Dequeue Head*) and push it in the *Accepted Entry Calls Stack*.
4. Update the out-mode parameter *Param_Access* with the reference to the *Entry Parameters Record* so that the compiler generated code can access the entry parameters.
5. Undefer the abortion.

If no exception is raised during the execution of the user code the GNARL subprogram *Complete_Rendezvous*[16] is called. This subprogram just calls *Exceptional_Complete_Rendezvous* notifying that no exception has been raised. If some exception is raised *Exceptional_Complete_Rendezvous* is called from the exception handler. This procedure performs the following actions.

- **Exceptional_Complete_Rendezvous**[17]:

1. Defer the abortion.
2. Pop the reference to the *Entry Call Record* from the *Accepted Entry Calls Stack*.
3. If an exception was raised, get its identifier from the entry call field *Exception_To_Raise* and save its occurrence in the ATCB field *Compiler_Data*. This exception will be propagated back to the caller when the rendezvous is completed [AAR95, section 9.5.3].
4. Wake up the caller (*Wakeup_Entry_Caller*).
5. Undefer the abortion.

3.2.7 Selective Accept

The special implementation problem introduced by the selective wait is that a task may at one instant be ready to accept a call on a set of several entries. From the viewpoint of the Ada run-time, this is really two problems, since it comes up in the processing of entry calls, as well as selective waits:

[15] *System.Tasking.Rendezvous.Wait_For_Call*
[16] *System.Tasking.Rendezvous.Complete_Rendezvous*
[17] *System.Tasking.Rendezvous.Exceptional_Complete_Rendezvous*

1. Since a task may be waiting on more than one open accept alternative, processing an entry call requires checking whether the called entry corresponds to one of the open alternatives.

2. Since there may be several open accept alternatives, processing the selective wait requires checking the set of pending entry calls against the set of open accept alternatives.

The need to be able to perform both of these operations efficiently strongly influences an implementation's choice of data structures. There are two obvious ways to perform the first operation, checking whether a called entry has a currently open accept alternative:

1.1. If the set of open accept alternatives is represented as a list, checking requires comparing the called entry against each of the entries in this list. We call this approach the use of an *open entry list*. It may be time consuming if there are many open entries.

1.2. An alternative is to use a vector representation for the set of open entries: the *open accepts vector*. This vector would have one component for each entry of the task. Each component would minimally indicate whether the corresponding entry is open.

Note that the accept vector or open entry list must be created at the time the selective wait statement is executed, once it is known which alternatives are open. The time needed to do this only depends on the number of alternatives in the selective wait statement.

With separate queues for each entry, it is necessary to check the queue corresponding to each open entry. This requires sequencing through the open entries. Alternatively, if the open entries are represented by an open entry list, this check can be performed more quickly, without looking at the non-open entries. This may be a good reason to keep both an open entry list and an accept vector, though this redundancy may cost more in overhead than it saves through faster execution of the check for pending calls.

GNAT uses the *Open Accepts Vector*. Each element of this vector has two fields: the entry identifier and a boolean which indicates if the accept statement has a null body. Each element of the accept vector corresponds to the accept alternatives of the select statement (in the same order; first element of the accept vector

corresponds to the first alternative, second element corresponds to the second alternative, etc.). The entry identifier is set to 0 when the entry guard is closed. The reference to the accept vector is stored in the *Open_Accepts* ATCB field. For example, the following task has three entries (P, Q and R). In the select statement the first two entries are open, but the third entry is closed. Additionally, the first and third entry have a null body.

```
task T is
   entry P;   -- Entry Id = 1
   entry Q;   -- Entry Id = 2
   entry R;   -- Entry Id = 3
end T;

task body T is
begin
   select
      accept Q;
   or
      accept P do
         ...
      end P;
   or
      when False =>
      accept R;
   end select;
end T;
```

Figure 3.4 has the corresponding *Open Accepts Vector*. The first **accept** alternative corresponds to the second entry (therefore, the *Entry_Id* field is set to "2") which is open and has a null body. The second alternative corresponds to the first entry, which is also open and has some user-defined code. Finally, the last alternative has the guarding condition closed and, therefore the *Entry_Id* in the *Open Accepts Vector* is set to "0".

The GNAT compiler translates the selective accept to one scope where it declares three variables: the *Open Accepts Vector*, the *Index* of the selected alternative and the address of the *Entry Parameters Record*. Index value 0 is used by the run-time to indicate that the **else** alternative has been selected.

```
declare
   Open_Accepts_Table   : GNARL.Open_Accepts_Table;
   Index                : Natural;
   Params_Block_Address : System.Address;
begin
   GNARL.Selective_Wait
     (Open_Accepts_Table, GNARL.Select_Mode,
      Params_Block_Address, Index);
```

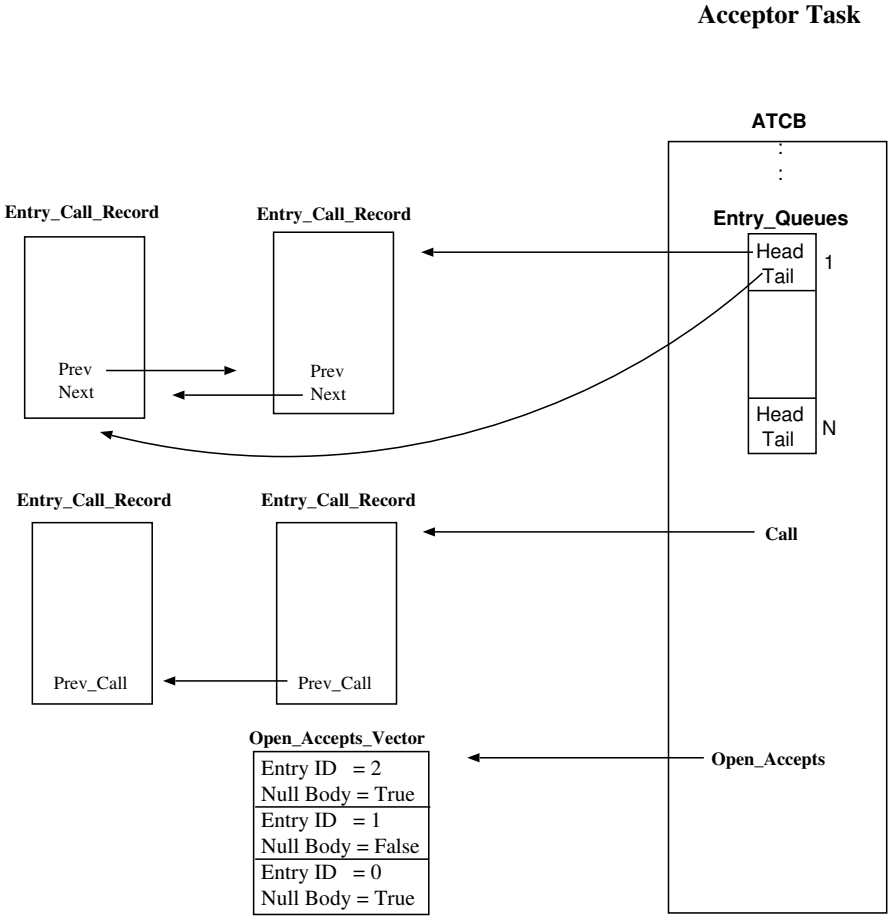

Figure 3.4: Open Accepts Vector.

```
   case Index is
      when 0 =>
         --  else part
         ...
      when 1 =>
         --  user code of the first accept
         ...
      when 2 =>
         --  user code of the second accept
         ...
   end case;
end;
```

The user code associated with each alternative is translated to local procedures. Below we have the general structure of these procedures.

```
procedure Entry_Name is
```

```
begin
   GNARL.Undefer_Abortion;
   << User Code >>
   GNARL.Complete_Rendezvous;
exception
   when others =>
      GNARL.Exceptional_Complete_Rendezvous;
end Entry_Name;
```

The GNARL procedure *Selective_Wait* carries out the following actions:

- **Selective_Wait**[18]:

  1. Defer the abortion.
  2. Try to serve the entry call immediately. GNARL subprogram *Select_Task_Entry_Call*[19] selects one entry call following the queuing policy being used.
     (a) If there is some candidate and the accept has a null body then complete the rendezvous, wake up the caller, undefer the abortion and RETURN.
     (b) If there is some candidate and the accept has some associated code then insert the *Entry Call Record* in the *Accepted Entry Calls Stack* (*Setup_For_Rendezvous_With_Body*[20]), update the reference to the *Parameters Record*, undefer the abortion and RETURN.
     (c) If there is no candidate but there are alternatives opened, wait for a caller. In the future some caller will put an entry call record in the *Accepted Entry Calls Stack* and it will wake up this acceptor. Then this acceptor will update the reference to the entry parameters, it will undefer the abortion, and it will RETURN.
     (d) If there is a terminate alternative, notify its ancestors that this task is on a terminate alternative (*Make_Passive*[21], and wait for normal entry call or termination.
     (e) If no alternative is open and no delay (or terminate) has been specified then raise the predefined exception *Program_Error*.

[18] *System.Tasking.Rendezvous.Selective_Wait*
[19] *System.Tasking.Queuing.Select_Task_Entry_Call*
[20] *System.Tasking.Rendezvous.Setup_For_Rendezvous_With_Body*
[21] *System.Tasking.Utilities.Make_Passive*

3.2.8 The Count Attribute

The GNARL function *Task_Count*[22] implements this attribute and returns the number of queued entry calls in the specified entry queue.

3.3 Summary

The *Rendezvous* is the basic mechanism for synchronization and communication of Ada tasks. In this chapter, the main aspects of the GNAT implementation have been described. In summary:

- The run-time information associated with the entry call is grouped into an *Entry Call Record*.
- The compiler generates one *Entry Parameters Record* with the address of the real-parameters. GNARL registers the address of this record in a field of the Entry Call Record.
- The entry queues are implemented by means of double linked lists of Entry Call Records.
- Nested accepts are handled by means of one *Accepted Entry Calls Stack* (a linked list of accepted Entry Call Records).
- An *Accept Vector* is used to evaluate the open guards of the selective accept.

[22] *System.Tasking.Rendezvous.Task_Count*

Chapter 4

Protected Objects

Ada 95 protected objects provide synchronization based on a data object rather than a thread of control. Protected operations can be procedures, functions and entries. Calls to protected procedures and entries are executed in mutual exclusion; no other operation of the same protected object can proceed in parallel when a protected procedure or entry is being executed. The functions can execute in parallel, but not when a protected procedure or entry of that object is executing. Functions are not permitted to affect the state of a protected object.

4.1 The Ada 95 Protected Object

A protected object in Ada encapsulates data items and provides access to them only via protected subprograms on protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures that the data is updated under mutual exclusion [BW98, chapter 7.1].

A protected unit may be declared as a type or as a single instance. In this latter case it is said that the corresponding type is *anonymous*. A protected unit has a specification and a body. The specification has the following syntax [AAR95, section 9.4].

```
protected_type_declaration ::=
   protected type defining_identifier [known_discriminant_part] is
   protected_definition;

single_protected_declaration ::=
   protected defining_identifier is protected_definition;
```

```
protected_definition ::=
    protected_operation_declaration
[ private
    protected_element_declaration  ]
end [protected_identifier]

protected_operation_declaration ::=
     subprogram_declaration
   | entry_declaration
   | aspect_clause

protected_element_declaration ::=
     protected_operation_declaration
   | component_declaration
```

Thus a protected type has an interface that can contain functions, procedures and entries. As with tasks and records, the discriminants can only be a discrete or access type [BW98, chapter 7.1]. The body is declared using the following syntax:

```
protected_body ::=
  protected body defining_identifier is
      protected_operation_item
  end [protected_identifier];

protected_operation_item ::=
    subprogram_declaration
  | subprogram_body
  | entry_body
  | aspect_clause
```

A protected type is a “limited type”, and therefore there are no predefined assignment or comparison operators (the same is true for task types).

A protected procedure provides mutually exclusive read/write access to the data encapsulated. Protected functions provide concurrent read-only access the encapsulated data. This means that many function calls can be executed simultaneously. However, calls to a protected function are still executed mutually exclusive with calls to a protected procedure. The core language does not define the order in which tasks waiting to execute protected functions and protected procedures are executed. If, however, the Real-Time Systems Annex [AAR95, Annex D] is being supported, certain assumptions can be made about the order of execution [BW98, chapter 7.2].

A protected entry is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has the read/write access the encapsulated data. However, a protected entry is guarded by a boolean expression (called a *Barrier*)

inside the body of the protected object; if this barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier evaluates to true and no other task are currently active inside the protected object. Hence protected entry calls can be used to implement condition synchronization [BW98, chapter 7.3].

```
protected type Signal_Object is
   entry Wait;
   procedure Signal;
   function Is_Open return Boolean;
private
   Open : Boolean := False;
end Signal_Object;

protected body Signal_Object is

   entry Wait when Open is
   begin
      Open := False;
   end Wait;

   procedure Signal is
   begin
      Open := True;
   end Signal;

   function Is_Open return Boolean is
   begin
      return Open;
   end Is_Open;

end Signal_Object;
```

The state of the object must be put in the private part of the specification. The reason is that the protected object interface must provide all the information required by the compiler to allocate the required memory in an efficient manner.

Clearly, it is possible for more than one task to be queued on a particular protected entry. As with task queues, a protected entry is, by default, ordered in a first-in-first-out fashion; however, if the Real-Time Systems Annex is being supported, other queuing disciplines are allowed [BW98, chapter 7.3].

4.1.1 Entry Calls and Barriers

To issue a call to a protected object, a task simply names the object and the required subprogram or entry. As with task entry calls, the caller can use the select

statement to issue a *timed* or *conditional* entry call [BW98, chapter 7.4] (timed conditional entry call will be analyzed in a separate chapter).

```
Object_Name.Entry_Name (Parameters)
```

```
select
   Object_Name.Entry_Name (Parameters);
   <<Statements>>;
else
   <<Statements>>;
end select;
```

When a call on a protected procedure or protected entry is executed, the barrier is evaluated; if the barrier is closed (evaluates to False), the call is queued. When the execution of a protected procedure or entry is completed, all the barriers are re-evaluated and, potentially, entry bodies are executed. The evaluation of the entry barrier and the queuing of the entry call are also protected operations on the associated object. Barrier evaluation, protected entry queuing operations and protected subprogram execution are collectively referred to as *protected actions*.

Any exception raised during the evaluation of a barrier results in *Program_Error* being raised in all tasks currently waiting on the entry queues associated with the protected object containing the barrier [BW98, chapter 7.8]).

4.1.2 The Eggshell Model

A queued entry call has precedence over other operations on the protected object. This is often explained in terms of the *eggshell model*. The lock on a protected object is the eggshell.

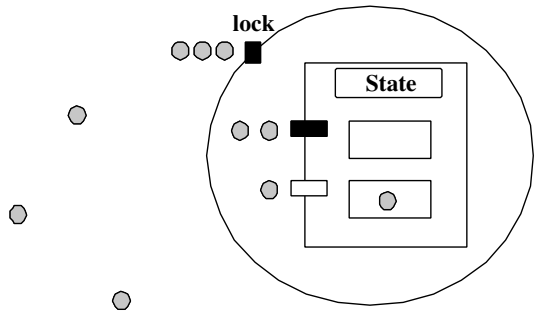

Figure 4.1: Graphical Representation of the Protected Object.

Figure 4.1 presents a graphical representation of the protected objects. Tasks flow of control is represented by means of shadowed circles; the protected object levels are represented by means of a big circle (associated with the object lock) and a big rectangle (associated with the object state and operations—represented by small rectangles). Closed entries are represented by means of black rectangles; opened entries are represented by means of while rectangles. In this example the reader can see one single task executing one protected operation (in mutual exclusion), several tasks queued in the entry queues and several additional tasks queued in the lock.

To perform any protected operation, a task (represented by small shadowed circles) must enter the eggshell. One protected procedure or entry call or several protected function calls can be active in the eggshell at a time. Tasks attempting to enter an eggshell that is occupied by a procedure or entry call will be blocked.

An entry call must enter the eggshell to evaluate the associated barrier (represented by small rectangles). If the barrier is *Open* (white small rectangle), the entry body is executed inside the eggshell; otherwise (black small rectangle), the call is queued within the eggshell. Since queued entry calls are not active, other calls can enter the eggshell in which they are queued.

Queued entry calls become eligible to execute when their barriers become *Open*. This need only be checked when they become true as the result of a protected procedure or entry call on the same object [AAR95], essentially treating the barrier expression as though they depended only on the state of the protected object. Therefore before an operation that may have changed the state of a protected object exits the eggshell, any queued entry calls waiting on barriers that now evaluate to *True* must be executed. Only when the barriers of all entries with queued calls are *False* can the eggshell be exited. This assures that all entry calls made eligible by a state change are executed before any further operations are initiated.

4.1.3 Private Entries and Entry Families

As with tasks, protected objects may have private entries. These are not directly visible to users of the protected object. They may be used during requeue operations [BW98, chapter 7.5].

A protected type can also declare a family of entries by placing a discrete subtype definition in the specification of the entry declaration. Unlike task entry families, however, the programmer need not provide a separate entry body for each member of the family. The barrier associated with the entry can use the index of

the family (usually to index into an array of booleans) [BW98, chapter 7.5] (see examples in [BW98, chapter 7.5])

4.1.4 Restrictions on Protected Objects

In general, code executed inside a protected object should be as brief as possible. This is because whilst the code is being executed other tasks are delayed when they try to gain access to the protected object. The Ada language clearly cannot enforce a maximum length of execution time for a protected action. However, it does try to ensure that a task cannot be blocked indefinitely waiting to gain access to a protected procedure or a protected function. The ARM defines it to be a bounded error to call a *potentially blocking* operation from within a protected action. The following operations are defined as potentially blocking [BW98, chapter 7.6]:

1. A select statement.
2. An accept statement.
3. An entry call statement.
4. A delay statement.
5. Task creation or activation.
6. A call on a subprogram whose body contains a potentially blocking operation.

If a bounded error is detected, the predefined exception *Program_Error* is raised.

4.1.5 Elaboration and Finalization

A protected object is elaborated when it comes into scope in the usual way. However, a protected object cannot simply go out of scope if there are still tasks queued on its entries. Finalization of a protected object requires that any tasks left on entry queues have the exception *Program_Error* raised [BW98, chapter 7.8]:

4.1.6 The Count Attribute

Protected object entries, like task entries, have a *Count* attribute defined that provides the current number of tasks queued on the specified entry. It is important to note that even if a task is destined to end up on an entry queue (due to the barrier being closed) it requires the write-lock to be placed on the queue. Moreover, having been queued, the *Count* attribute will have changed (for that queue) and hence any barriers that have made reference to that attribute will need to be re-evaluated [BW98, chapter 7.4].

4.2 GNAT Implementation

Ada does not specify which task executes the entry barrier and the entry body. Protected entry bodies can be executed by any task, regardless of which task made the corresponding entry call [GB94a, Section 5(3)]). Therefore two obvious candidates are (1) the task that called the entry and (2) the task which opens the barrier of the entry [GMB93, Section 3.4]. They are referred to as the *Self-Service Model* and the *Proxy Model* of protected entry execution, respectively.

4.2.1 Self-Service Versus Proxy

In the self-service mode[1] a task exiting a protected object selects a task waiting on an open entry barrier (if any) based on the entry queuing discipline in effect and wakes it up, making it the active task within the eggshell.

In the proxy model, the task exiting the eggshell executes the entry body itself, wakes the calling task and then repeats the check and entry body execution for waiting callers, finally exiting the eggshell when there are no more entry calls waiting on True barriers.

The principal advantages of the self-service model are that it permits more parallelism and simplifies schedulability analysis. Parallelism is increased because on a multiprocessor the exiting task can proceed with its own execution in parallel with the execution of the next queued call. Schedulability analysis is simplified because a thread is allowed to continue with its own execution immediately after the (presumably bounded) time it takes to complete the body of its own protected operation and pass lock ownership to the next queued caller.

The principal advantage of the proxy model is simplicity. If an entry body cannot be executed immediately, the calling task has only to suspend and wait (some other task will be responsible for executing the entry body). More complex features of protected objects, including timed and asynchronous entry calls, are simplified even more by this model. However, schedulability analysis is compromised by this model since a task attempting to exit an eggshell must first execute all of the waiting entry calls whose barriers are open, and there is no language-imposed restriction on the number of such calls that can be pending.

Using Pthreads to implement the self-service model introduces one problem. The task attempting to exit an eggshell must be able to transfer ownership to a

[1] The contents of this section are a summary of [GMB93].

task waiting on an *Open* entry. However, there is no good way to solve it with Pthreads because, though it is possible to force a thread to be given a mutex by raising its priority over that of the other contenders, this may lead to unnecessary context switches and degrades the implementation of Ada priority over Pthreads. Therefore GNAT implements the proxy model.

4.2.2 Proxy Model: In-line Versus Call-Back Implementation

There are two possible implementations of the proxy model: *In-line* and *Call-back*. In the In-line implementation the code generated for all the barriers and entry bodies is put by the compiler in a single *entry service procedure* (cf. Figure 4.2). This procedure has a main loop which evaluates all the barriers and calls a GNARL procedure to select the next entry code to be executed (the value 0 is returned when there is no candidate). This service entries procedure is called at the end of any protected action that might result in a change of state in the object.

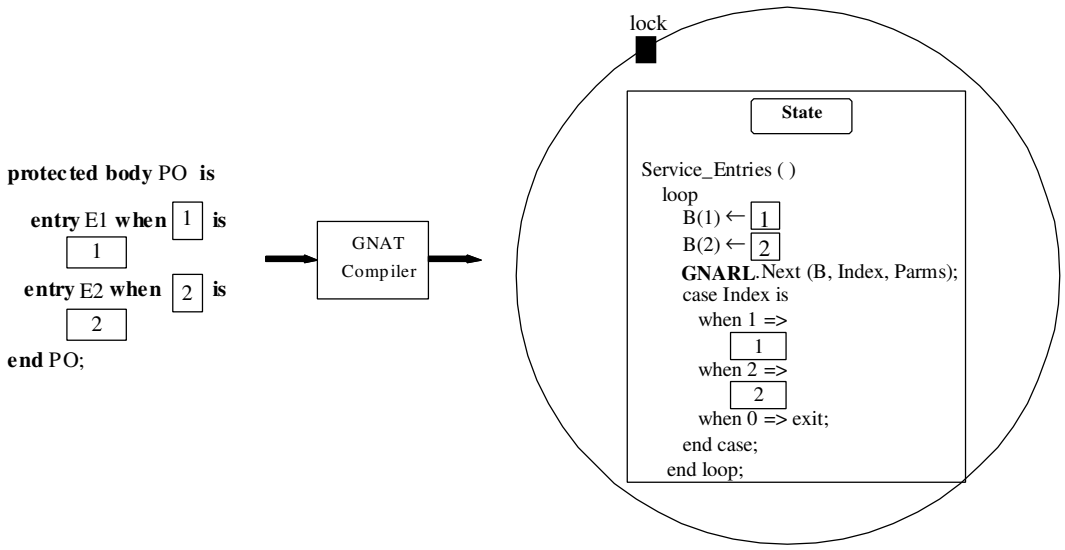

Figure 4.2: Proxy Model: In-Line Implementation.

In the Call-back implementation all the logic required to implement the semantics of the protected objects is moved down to the GNARL level. The compiler translates entry barrier to a function that returns a Boolean datum and translates the entry body to a procedure (cf. Figure 4.3). In addition the compiler generates a table with the access to these subprograms. The reference to this table must be

given to the Ada run-time.

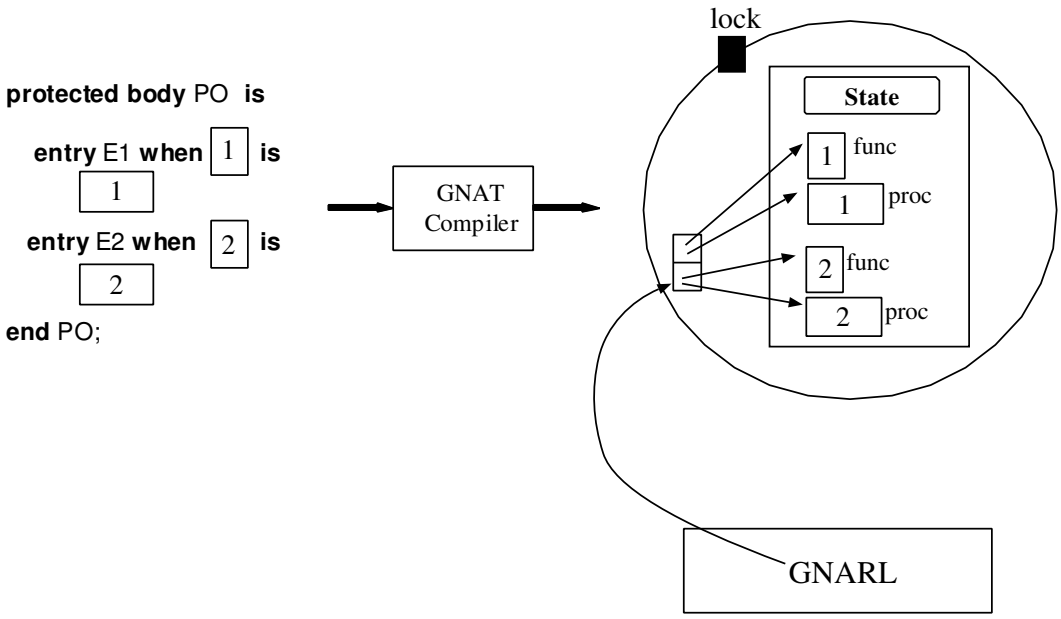

Figure 4.3: Proxy Model: Call-Back Implementation.

Until version 2.04, the in-line implementation was used in GNAT. Then they decided to implement both models and to compare both implementations. Although the in-line implementation allows the compiler to make better optimizations, their results indicated that the call-back interface allows for much simpler translations and eliminates some of the overhead inherent in the in-line interface's frequent alternation between the GNU Ada Run-Time and the application code. The call-back interface also has a big advantage in the simplicity and understandability of both the generated code and the internal logic of the compiler [GB95]. Therefore, the current versions of the GNAT compiler have the call-back implementation.

4.2.3 Protected Type Specification

The GNAT compiler translates the following protected type specification

```
protected type PO ( <Discriminants> )is
   procedure P ( <Params> );
   function  F ( <Params> ) return ...;
private
```

```
    <Private_Data>;
end PO;
```

...to the following code:

```
 1: type poV (Discriminants) is new Limited_Controlled with
 2:    record
 3:       <Private_Data>
 4:       _object : aliased GNARL.Protection_Entries ( <Num_Entries> );
 5:    end record;
 6:
 7: procedure Finalize (_object : poV) is
 8: begin
 9:    --  Raise Program_Error to the queued tasks.
10:    ...
11: end Finalize;
```

The protected type specification is translated to a record type declaration (lines 1 to 5). If the protected type has discriminants, the record type has the same discriminants (in order to provide the same semantics). Private data is translated to components of this record (line 3). The additional field *_object* (line 4) contains all the run-time data required to implement the protected object semantics (the object lock, the entry queues, the object priority, etc. —see the GNARL data type *Protection_Entries*[2]). It is an **aliased** component because its access must be passed to the GNU Ada Run-Time. As the reader can see, the record type is Limited_Controlled[3]. These are the reasons:

1. *Limited.* Protected objects can not be copied. In this way Ada ensures that the object state can only be modified by its protected operations. This semantics is provided by the Ada limited types.

2. *Controlled.* When the object finalizes, the predefined exception exception *Program_Error* must be raised to all the queued tasks. Therefore, the GNAT compiler automatically generates a procedure which does this work (lines 10 to 14).

4.2.4 Protected Subprograms

For each protected subprogram, the GNAT compiler generates two subprograms: N and P. N has the user code. It is only called when the object lock has been

[2] *System.Tasking.Protected_Objects.Entries*
[3] *Ada.Finalization*

acquired. P is responsible to take the lock and to call N. This scheme allows a protected subprogram to call another protected subprogram in the same object (in this case the compiler generates a direct call to the corresponding N subprogram). One additional parameter is added by the compiler to the parameters profile of the protected subprograms: the *_object*. Because protected procedures can modify the object's state, they receive the object as **in out** mode parameter. Protected functions receive the object as an **in** mode parameter.

```
procedure procN (_object : in out poV; ... );
procedure procP (_object : in out poV; ... );
```

The compiler adds some renaming sentences to the declarations of the protected subprograms. These renamings simplify the access to the object discriminant and to the private state.

```
procedure procN (_object : in out poV; ... ) is
   <Discriminant_Renamings>
   <Private_Object_Renamings>
begin
   <Sequence_Of_Statements>
end procN;
```

All the sentences of the protected subprograms are properly modified by the compiler to make use of these renamings when the discriminant or the object private data fields are used. Let's see the P subprogram in detail.

```
 1: procedure procP (_object : in out poV; ... ) is
 2:
 3:    procedure Clean is
 4:    begin
 5:       GNARL.Service_Entries (_object._object'access);
 6:       GNARL.Unlock (_object._object'access);
 7:       GNARL.Abort_Undefer;
 8:    end Clean;
 9:
10: begin
11:     GNARL.Abort_Defer;
12:     GNARL.Lock_Write (_object._object'access);
13:     procN (_object; ... );
14:     Clean;
15: exception
16:     when others =>
17:         declare
18:            E : Exception_Occurrence;
19:         begin
20:            GNARL.Save_Occurrence (E, GNARL.Get_Current_Exception);
21:            Clean;
```

```
22:           GNARL.Reraise (E);
23:        end;
24: end procP;
```

The first actions performed by the P subprogram are to defer the abortion[4] (line 11) and to lock the object[5] (line 12). After calling the N subprogram (line 13) we have two possible scenarios:

1. *No exception was raised*. In this case the local subprogram *Clean* is called (line 14) to make the following actions: serve the opened entries with queued tasks (line 5), unlock the protected object (line 6) and undefer the abortion (line 7).

2. *Some exception was raised*. In this case, before propagating the exception to the calling task, the protected object must first service the entries with queued tasks (according to the Proxy model). However, the execution of these entries may also raise new exceptions (and the current exception would be lost). Therefore, it is necessary to save the exception occurrence originally raised (line 20) and re-raise it after local subprogram *Clean* returns.

The GNARL subprogram *Service_Entries*[6] will be described in section 4.2.8.

4.2.5 Entry Barrier

Each entry barrier expression is translated by the compiler to a function that returns a Boolean type; each entry body is put inside a procedure. A table[7] is also created by the compiler to store the access these subprograms. This table is used by GNARL to evaluate the entry barriers and to call the selected entry body.

```
function EntryBarrier
   (Object       : Address;
    Entry_Index  : Protected_Entry_Index)
   return Boolean
is
   <Discriminant_Renamings>
```

[4] *System.Tasking.Initialization.Defer_Abort*
[5] *System.Tasking.Protected_Objects.Entries.Lock_Entries*
[6] *System.Tasking.Protected_Objects.Entries*
[7] The data type of this table is *System.Tasking.Protected_Objects.Protected_Entry_Body_Array*

```
    <Private_Object_Renamings>
begin
   return <Barrier_Expression>;
end EntryBarrier;
```

Because all the entry barriers must have the same profile (to create the access table used to implement the call-back) the object parameter is passed by means of an *Address* parameter.

4.2.6 Entry Body

The entry body is translated by the compiler to a procedure.

```
 1: procedure EntryName
 2:    (Object      : Address;
 3:     Parameters  : Address;
 4:     Entry_Index : Protected_Entry_Index)
 5: is
 6:    <Discriminant_Renamings>
 7:    <Private_Object_Renamings>
 8:
 9:    type poVP is access poV;
10:    function To_PoVP is new Unchecked_Conversion (Address, PoVP);
11:    _object : PoVP := To_PoVP (Object);
12: begin
13:    <Statement_Sequence>
14:    GNARL.Complete_Entry_Body (_object._object);
15: exception
16:    when others =>
17:       GNARL.Exceptional_Complete_Entry_Body
18:          (_object._object, GNARL.Get_GNAT_Exception);
19: end EntryName;
```

The object is again passed by means of an *Address* parameter (to create the access table used to implement the call-back). Similar to the N subprograms (section 4.2.4), the compiler adds some renamings to facilitate the access to the object discriminant and to the private fields (lines 6 and 7). The compiler also generates the unchecked conversion of the object parameter to the typed object (lines 9 to 11). If no exception is raised by this code, the GNARL subprogram *Complete_Entry_Body*[8] is called to notify to the Ada run-time that this entry body has been successfully serviced. Otherwise (some exception was raised) the GNARL subprogram *Exceptional_Complete_Entry_Body*[9] is called. This subpro-

[8] *System.Tasking.Protected_Objects.Operations.Complete_Entry_Body*
[9] *System.Tasking.Protected_Objects.Operations.Exceptional_Complete_Entry_Body*

gram stores the exception occurrence in the entry call. The exception will be raised by the calling tasks after being woken up.

4.2.7 Entry Family

For each entry family the compiler adds one field to the type declaration which is used to store the bounds of the entry family declaration (the contents of the array are not used).

```
type poV (Discriminants) is new Limited_Controlled with
record
   <Private_Data>
   _object : aliased GNARL.Protection_Entries ( <Num_Entries> );
   Entry_Family_Name : array ( <Bounds> ) of Void;
end record;
```

4.2.8 Service Entries

The basic algorithm of the GNARL *Service_Entries*[10] procedure is as follows:

```
 1 while <There_Is_Some_Open_Barrier_With_Queued_Entry_Calls> loop
 2    Update object reference to the Entry_Call_Record
 3    begin
 4      Call the Entry_Body
 5    exception
 6      when others => Broadcast Program_Error
 7    end
 8    Remove the Reference to the Entry_Call_Record
 9    GNARL.Wake_Up_Entry_Caller
10 end loop;
```

Line 1 is evaluated by the GNARL procedure *Select_Protected_Entry_Call*[11] which traverses all the entry queues and reevaluates the barrier of those entries with queued entry calls. As soon as some barrier is open (it evaluates to true), GNARL selects it to be serviced. In line 2, the *Call_In_Progress* field of the *_object* (see the *Protection_Entries* type definition) is set to the selected entry call record to remember that this is the entry call being attended. Lines 3 to 7 open a new scope to issue the call to the entry body and to handle the exceptions in the user

[10] *System.Tasking.Protected_Objects.Entries.Service_Entries*
[11] *System.Tasking.Queueing.Select_Protected_Entry_Call*

code. In this case the predefined exception *Program_Error* is broadcasted to all tasks currently queued in any entry of the protected object. In line 8 the reference to the entry call is removed (this entry call has been attended) and the task entry caller is woken up (line 9). After this work the loop is executed again and the entry barriers are reevaluated. This process stops when no open barrier is found in an entry with queued tasks.

4.2.9 Simple Mode Entry Call

A simple call to a protected entry is translated by the compiler to a call to the GNARL subprogram *Protected_Entry_Call*. The entry calls to protected procedures are handled in a similar way to task entry calls (this facilitates the implementation of the Ada requeue statement). Therefore, one *Entry_Call_Record* of the *Entry Calls Stack* is used to issue the entry call.

- **Protected_Entry_Call**[12]:

  1. Defer the abortion.
  2. Write lock the object.
  3. Elaborate a new *Entry Call Record*[13].
  4. Call the GNARL procedure *PO_Or_Queue* to issue the call or to enqueue it in the corresponding entry queue.
  5. Call the GNARL procedure *PO_Service_Entries*[14] to service the opened entries.
  6. Unlock the object.
  7. Undefer the abortion.
  8. Check if some exception must be re-raised.

- **PO_Do_Or_Queue**[15]:

  1. Call the barrier function.

[12] *System.Tasking.Protected_Objects.Operations.Protected_Entry_Call*
[13] *System.Tasking.Entry_Call_Record*
[14] *System.Tasking.Protected_Objects.PO_Service_Entries*
[15] *System.Tasking.Protected_Objects.PO_Do_Or_Queue*

2. If the barrier is closed then enqueue the *Entry Call Record*, and RETURN.
3. If the barrier is open then execute the steps 2 to 9 of the GNARL procedure *Service_Entries*.

4.2.10 Conditional Mode Entry Call

In this case the actions carried out by the GNAT run-time are basically the same as in the previous case; however, if the barrier is closed the *Entry Call Record* is not enqueued and the *else part* of the conditional entry call is executed.

4.3 Summary

In this chapter we have briefly presented the GNAT implementation of the protected objects. The main concepts are:

- There are two models to implement the protected objects: the self-service and the proxy model. GNAT uses the *Proxy Model*.
- There are also two possibilities to implement the proxy model: the in-line and the call-back implementation. Although the in-line implementation was used in the initial versions, nowadays GNAT has adopted the *Call-back* implementation.
- Protected subprograms are translated to two subprograms (P and N). P is responsible to take the object lock and N has the user code.
- Barriers are translated to functions that return a Boolean data type.
- Entry bodies are translated to procedures.

Chapter 5

Time and Clocks

Many embedded systems need to coordinate their executions with the natural time of their environment. To do this, they use a hardware clock that approximates the passage of real-time. For long running programs (that is, years of non-stop execution), this clock may need to be resynchronized to some external authority (including the International Atomic Time) but from the program's point of view the clock is the source of *real* time [BW98, section 2.5].

5.1 Ada Time and Clocks

Ada provides access to the clock by providing two packages. The main section of the Ada Reference Manual defines a compulsory library package called *Ada.Calendar* that provides an abstraction for "wall clock" time that recognizes leap years, leap seconds and other adjustments. In Real-Time Systems Annex, a second representation is given that defines a monotonic (that is, non-decreasing) regular clock (package *Ada.Real_Time*). Both these representations should map down to the same hardware clock but cater for different application needs [BW98, section 2.5].

5.1.1 Ada.Calendar

The interface of this package is as follows [AAR95, section 9.6]:

```
package Ada.Calendar is

   type Time is private;

   subtype Year_Number  is Integer range 1901 .. 2099;
   subtype Month_Number is Integer range 1 .. 12;
   subtype Day_Number   is Integer range 1 .. 31;

   subtype Day_Duration is Duration range 0.0 .. 86_400.0;

   function Clock return Time;

   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;

   procedure Split
     (Date    : Time;
      Year    : out Year_Number;
      Month   : out Month_Number;
      Day     : out Day_Number;
      Seconds : out Day_Duration);

   function Time_Of
     (Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Duration := 0.0)
      return    Time;

   function "+" (Left : Time;     Right : Duration) return Time;
   function "+" (Left : Duration; Right : Time)     return Time;
   function "-" (Left : Time;     Right : Duration) return Time;
   function "-" (Left : Time;     Right : Time)     return Duration;

   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   Time_Error : exception;

private
   . . .
end Ada.Calendar;
```

The current time is returned by the function *Clock*. Conversion between *Time* and program accessible types is provided by subprograms *Split* and *Time_Of*. in addition some arithmetic and boolean operations are specified [BW98, section 2.5].

5.1.2 Ada.Real_Time

The interface of this package is as follows [AAR95, section D.8]:

```
package Ada.Real_Time is

   type Time is private;
   Time_First : constant Time;
   Time_Last  : constant Time;
   Time_Unit  : constant := implementation-defined-real-number;

   type Time_Span is private;
   Time_Span_First : constant Time_Span;
   Time_Span_Last  :  constant Time_Span;
   Time_Span_Zero  :  constant Time_Span;
   Time_Span_Unit  :  constant Time_Span;

   Tick : constant Time_Span;
   function Clock return Time;

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

   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   function "+"  (Left, Right : Time_Span)             return Time_Span;
   function "-"  (Left, Right : Time_Span)             return Time_Span;
   function "-"  (Right : Time_Span)                   return Time_Span;
   function "*"  (Left : Time_Span; Right : Integer)   return Time_Span;
   function "*"  (Left : Integer;   Right : Time_Span) return Time_Span;
   function "/"  (Left, Right : Time_Span)             return Integer;
   function "/"  (Left : Time_Span; Right : Integer)   return Time_Span;

   function "abs" (Right : Time_Span) return Time_Span;

   function "<"  (Left, Right : Time_Span) return Boolean;
   function "<=" (Left, Right : Time_Span) return Boolean;
   function ">"  (Left, Right : Time_Span) return Boolean;
   function ">=" (Left, Right : Time_Span) return Boolean;

   function To_Duration  (TS : Time_Span) return Duration;
   function To_Time_Span (D : Duration)   return Time_Span;

   function Nanoseconds  (NS : Integer) return Time_Span;
   function Microseconds (US : Integer) return Time_Span;
   function Milliseconds (MS : Integer) return Time_Span;

   type Seconds_Count is new Integer range -Integer'Last .. Integer'Last;

   procedure Split (T : Time; SC : out Seconds_Count; TS : out Time_Span);
```

```
   function Time_Of (SC : Seconds_Count; TS : Time_Span) return Time;

private
   ...
end Ada.Real_Time;
```

5.1.3 Delay Statement

Ada tasks are able to delay their execution for a period of time. This enables the task to be queued on some future event rather than busy-wait o calls to the clock function. The expression following the **delay** must yield the value of the predefined Ada type *Duration*. It is important to appreciate that this sentence is an approximate time construct which indicates that the task will be delayed by at least the amount specified [BW98, section 2.5.1].

```
delay_relative_statement ::= delay delay_expression
```

If a delay to an absolute time is required, then the **delay until** statement should be used.

```
delay_until_statement ::= delay until delay_expression
```

As with delays, **delay until** is accurate only in its lower bound. The task involved will not be released before the current time has reached that specified in the statement but may be released later [BW98, section 2.5.1].

5.1.4 Timed Entry Call

A timed entry call issues an entry call which is canceled if the call is not accepted within the specified period (relative or absolute) [BW98, section 6.9.1]. The syntax is [AAR95, section 9.7.2]:

```
timed_entry_call ::=
   select
     entry_call_alternative
   or
     delay_alternative
   end select;

entry_call_alternative ::=
   entry_call_statement [sequence_of_statements]
```

5.1.5 Timed Selective Wait

Often it is the case that a server task cannot unreservedly commit itself to waiting for communication using one or more of its entries. The timed selective wait allows a server task to time-out if an entry call is not received within a certain period of time. The time-out is expressed using the delay statement and can therefore be a relative or an absolute delay. If the relative time expressed is zero or negative, or the absolute time has passed, then the delay alternative is equivalent to having an "else part" [BW98, section 2.5.1] (see section 3.1.5). The syntax of the selective accept is [AAR95, section 9.7.1]:

```
selective_accept ::=
  select
      [guard]
      select_alternative
 or
      [guard]
      select_alternative
[ else
    sequence_of_statements ]
  end select;

guard ::= when condition =>

select_alternative ::=
   accept_alternative
   | delay_alternative
   | terminate_alternative

accept_alternative ::=
   accept_statement [sequence_of_statements]

delay_alternative ::=
   delay_statement [sequence_of_statements]

terminate_alternative ::= terminate;
```

5.2 GNAT Implementation

5.2.1 *Delay* and *Delay Until* Statements

The GNARL subprograms which implement these Ada statements are placed in child packages of the corresponding standard Ada packages: *Ada.Calendar.Delays* and *Ada.Real_Time.Delays*. The GNAT compiler translates the delay statement into a call to the corresponding GNARL subprogram.

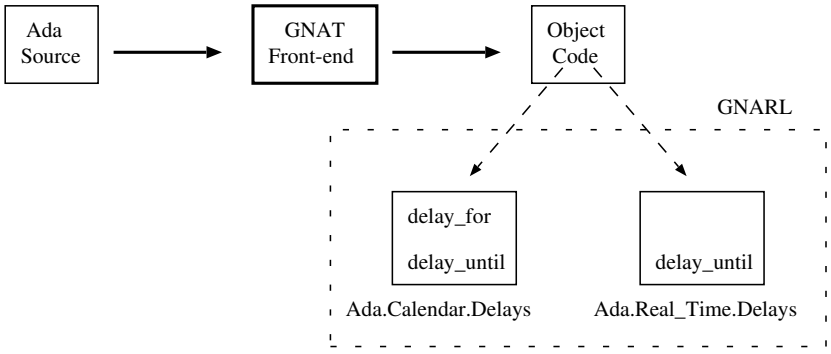

Figure 5.1: GNARL Subprograms for the Delay Statement.

GNARL provides two implementations of the delay statements: one for the case of an Ada program without tasks and the other for an Ada program with tasks. A link is used to access the proper subprogram (*Timed_Delay*[1]).

- In case of no tasking this link points to the GNARL procedure *Time_Delay_NT*[2], which calls the GNULL procedure *Timed_Delay*[3] (cf. Figure 5.2).
- In case of a program with tasks this link points to the GNARL procedure *Timed_Delay_T*[4], which calls another version of the GNULL procedure *Timed_Delay*[5] (cf Figure 5.3).

When the program has tasks, the GNARL procedure *Timed_Delay* performs the following actions.

[1] *System.Soft_Links.Timed_Delay.*
[2] *Ada.Calendar.Delays.Time_Delay_NT*
[3] *System.OS_Primitives.Timed_Delay*
[4] *System.Task_Initialization.Timed_Delay_T*
[5] *System.Task_Primitives.Operations.Timed_Delay*

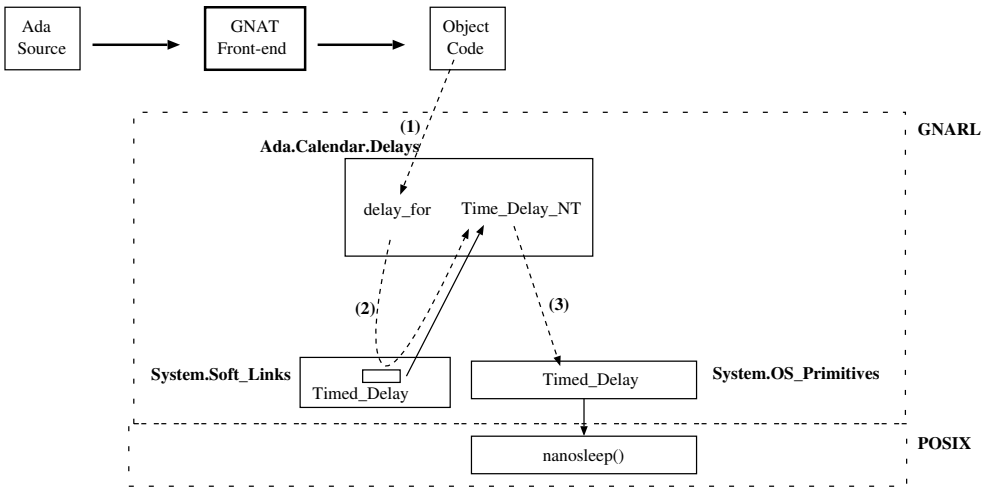

Figure 5.2: GNARL Subprograms for the Delay Statement in an Ada Program without Tasks.

- **Timed_Delay**[6]:

  1. Defer the abortion.
  2. Lock the ATCB of the calling task.
  3. If the specified delay is a relative time span (that is, a **delay** statement), this delay it is converted to absolute time span by adding the current value of the clock[7].
  4. If the specified time is a future time then
     (a) Set the state of the calling task to *Delay_Sleep*.
     (b) Call the POSIX function *pthread_cond_timedwait* to suspend the calling tasks until the specified time.
     (c) Set the state of the calling task to *Runnable*.
  5. Unlock the ATCB of the calling task.
  6. Yield the processor (this ensures that *"a delay statement always corresponds to at least one task dispatching point"* [AAR95, section D.2.2 (18)]
  7. Undefer the abortion.

[6] *System.Task_Primitives.Operations.Timed_Delay*
[7] *Ada.Calendar.Clock* or *Ada.Real_Time.Clock*.

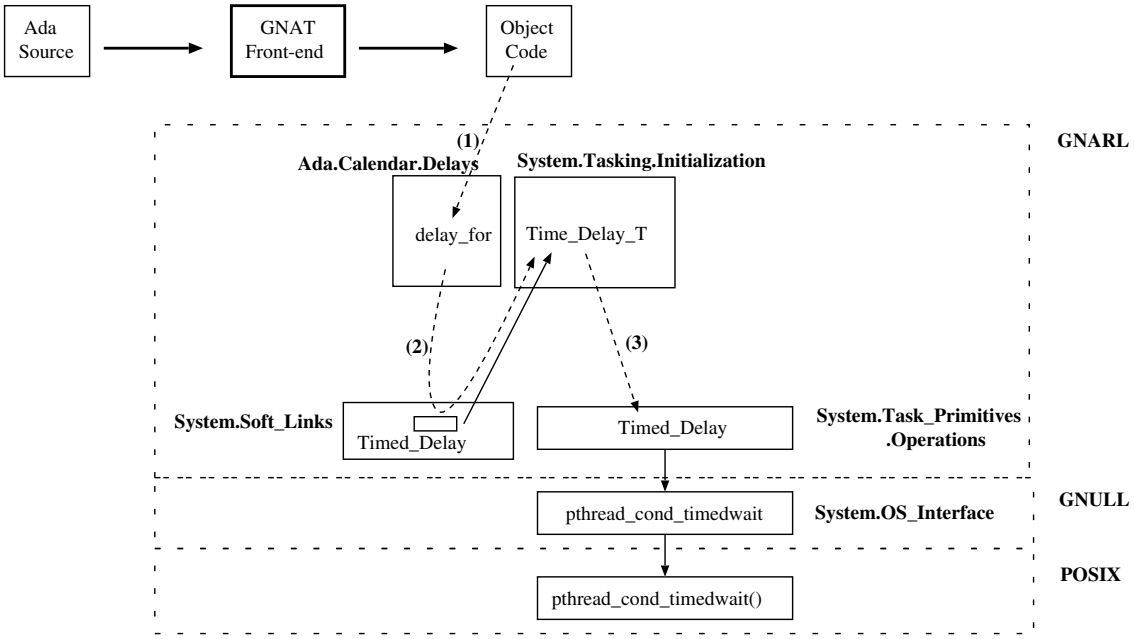

Figure 5.3: GNARL Subprograms for the Delay Statement in an Ada Program with Tasks.

5.2.2 Timed Entry Call

The timed task entry call is handled by the GNAT compiler in a similar way to the simple mode entry call (described in section 3.2.2). The compiler generates a call to the GNARL subprogram *Timed_Task_Entry_Call*[8]. Basically this procedure carries out the same actions described in the simple mode entry call (section 3.2.2). However, if the entry can not be immediately accepted, it does not simply block the caller; it calls another GNARL subprogram to arm a timer and block the caller until the timeout expires. Figure 5.4 shows the GNARL and GNULL subprograms involved in this action. If the entry call is accepted before this timer expires, the timer is un-armed; otherwise the entry call is removed from the queue.

The GNAT implementation of the timed protected entry call follows the same scheme described above. However, the only difference is that the compiler generates a call to the GNARL procedure *Timed_Protected_Entry_Call*[9].

[8] *System.Tasking.Rendezvous.Timed_Task_Entry_Call*
[9] *System.Tasking.Protected_Objects.Operations.Timed_Protected_Entry_Call*

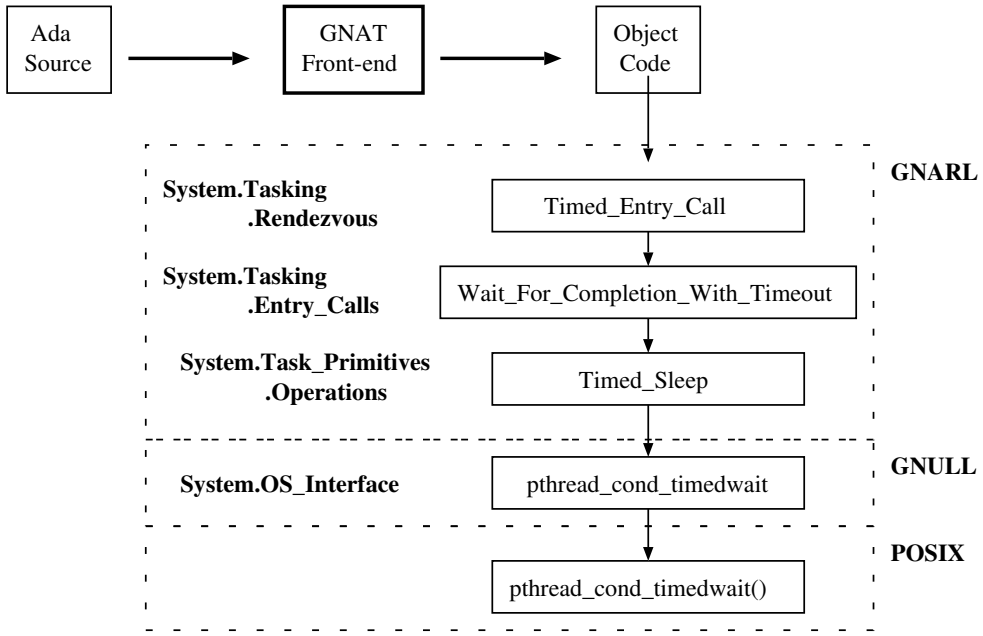

Figure 5.4: GNARL Subprograms for Timed Entry Call.

5.2.3 Timed Selective Accept

The timed task entry call is handled by the GNAT compiler in a similar way to the selective accept (described in section 3.2.7). The compiler generates a call to the GNARL subprogram *Timed_Selective_Wait*[10]. Basically this procedure carries out the same actions described in case of the selective wait (section 3.2.7). However, if there is no entry call that can be immediately accepted, it does not simply block the caller; it calls another GNARL subprogram to program a timer and block the caller until this timeout expires. Figure 5.5 shows the GNARL and GNULL subprograms involved in this action. If some entry call is received before this timer expires, the timer is un-armed; otherwise the statements after the **delay** sentence are executed.

5.3 Summary

GNAT provides two implementations for the simple **delay** and **delay until** Ada sentences: one for the Ada programs without tasks, and another for the Ada pro-

[10] *System.Tasking.Rendezvous.Timed_Selective_Wait*

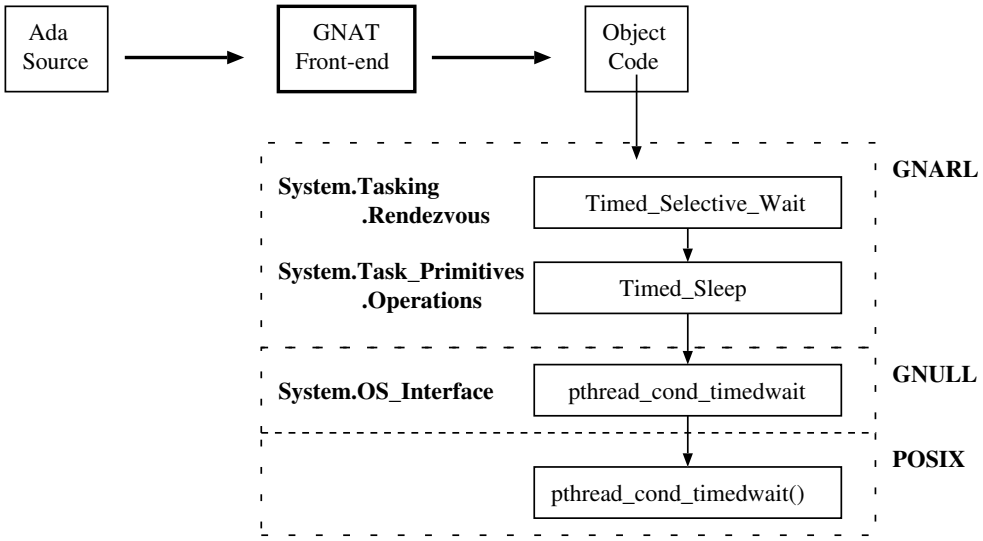

Figure 5.5: GNARL Subprograms for Timed Selective Accept.

grams with tasks. An access to a procedure is used to avoid multiple checks in the run-time to call the appropriate subprogram.

A timed entry call allows the task that executes it to make an entry call with the provision that it be awakened and the call canceled, if the call is not accepted before the expiration of a specified delay. As with the conditional entry call, provision is made for execution to resume in different places, depending on whether a rendezvous takes place. In addition to the processing required for a normal entry call, the timed entry call requires scheduling of a wake-up event if the call cannot be accepted immediately. If the call is accepted before this delay expires, the calling task must be removed from the delay queue. If the delay expires first, the task must be removed from the entry queue.

The GNAT implementation of the timed entry call sentences (to a protected entry or to a task entry) and the timed selective accept follow the same steps of the non-timed cases, though a timer is activated when the caller becomes blocked.

Chapter 6

Interrupts

Ada allows the programmer to associate a user defined interrupt handler to some interrupts [AAR95, section C.3(1)]. Although the interrupt handler can be a protected procedure or a task entry, currently the association of a task entry is considered an obsolescent feature [AAR95, section J.7] of the language. Therefore, in this chapter we will focus our attention on user defined protected procedure interrupt handlers.

This chapter is structured in two parts. In the first part the main aspects of the Ada attachment of user defined protected procedures to interrupt are presented. In the second part the main aspects of the GNAT implementation are described.

6.1 Ada Model of Interrupts

The AARM defines the following model of an interrupt [BW98, section 11.2] [AAR95, section C.3(1)]:

- An *Interrupt* represents a class of events that are detected by the hardware or system software.
- The *Occurrence* of an interrupt consists of its *Generation* and its *Delivery*.
- The *Generation* of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program.
- *Delivery* is the action which invokes a part of the program (called the interrupt *handler*) in response to the interrupt occurrence. In between the gen-

eration of the interrupt and its delivery, the interrupt is said to be *pending*. The handler is invoked once for each delivery of the interrupt.

- While an interrupt is being handled, further interrupts from the same source are *blocked*; all future occurrences of the interrupt are prevented from being generated. It is usually device dependent as to whether a blocked interrupt remains pending or is lost.
- Certain interrupts are *Reserved*. The programmer is not allowed to provide a handler for a reserved interrupt. Usually, a reserved interrupt is handled directly by the Ada run-time (for example, a clock interrupt used to implement the delay statement).
- Each non-reserved interrupt has a default handler that is assigned by the run-time system.

6.1.1 Interrupt-Handling Protected Procedures

Ada provides two styles of interrupt-handler installation and removal: *nested* and *non-nested*. In the nested style, an interrupt handler in a given protected object is implicitly installed when the protected object comes into existence, and the treatment that had been in effect beforehand is implicitly restored when the protected object ceases to exist. In the non-nested style, interrupt handlers are installed explicitly by procedure calls, and handlers that are replaced are not restored except by explicit request [Coh96, section 19.6.1].

A handler to be installed in the nested style is identified by the following pragma appearing in a protected declaration:

```
pragma Attach_Handler (Handler, Interrupt);
```

Handler is the name of a parameterless protected procedure in that protected declaration; *Interrupt* is an expression of type *Interrupt_ID*. The protected declaration must be library-level (it must not be nested in a subprogram body, task body, or block statement). However, if the protected declaration declares a protected type rather than a single protected object, individual objects of the type may be declared in these places. Dynamic allocation (by means of a **new** expression) gives greater flexibility: Allocating a protected object with an interrupt handler installs the handler associated with that object, and deallocating the protected object restores the handler previously in effect. The *Interrupt_ID* expression need

not be static; in particular, its value may depend on a discriminant of the protected type [Coh96, section 19.6.1]. For example:

```
package Nested_Handler_Example is

   protected type Device_Interface
     (Int_ID : Ada.Interrupts.Interrupt_ID) is

     procedure Handler;
     pragma Attach_Handler(Handler, Int_ID);

   end Device_Interface;

end Nested_Handler_Example;
```

A handler to be installed in the non-nested style is identified by the following pragma appearing in a protected declaration:

```
pragma Interrupt_Handler (Handler, Interrupt);
```

Again, *Handler* must be the name of a parameterless protected procedure. As with the Attach_Handler pragma, the protected declaration may not be nested in a subprogram body, task body, or block statement. However, this pragma has an additional restriction: if the protected declaration is for a protected type, objects of that type may not be nested in these places either [Coh96, section 19.6.1]. Therefore they must be dynamically created by means of a `new` expression.

6.1.2 Package Ada.Interrupts

Non-nested installation and removal of interrupt handlers relies on additional facilities of package *Ada.Interrupts* [AAR95, section C.3(2)]:

```
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID)
      return Boolean;

   function Is_Attached (Interrupt : Interrupt_ID)
      return Boolean;

   function Current_Handler (Interrupt : Interrupt_ID)
      return Parameterless_Handler;
```

```
      procedure Attach_Handler
         (New_Handler : in Parameterless_Handler;
          Interrupt   : in Interrupt_ID);

      procedure Exchange_Handler
         (Old_Handler : out Parameterless_Handler;
          New_Handler : in Parameterless_Handler;
          Interrupt   : in Interrupt_ID);

      procedure Detach_Handler
         (Interrupt : in Interrupt_ID);

      function Reference(Interrupt : Interrupt_ID)
         return System.Address;

   private
      ... -- not specified by the language
   end Ada.Interrupts;
```

The *Attach_Handler* procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If *New_Handler* is null, the default treatment is restored. If *New_Handler* designates a protected procedure to which the pragma *Interrupt_Handler* does not apply, *Program_Error* is raised [AAR95, section C.3.2].

```
with Ada.Task_Identification;
package Ada.Dynamic_Priorities is

   procedure Set_Priority
     (Priority : System.Any_Priority;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   function Get_Priority
     (T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task)
    return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

6.1.3 Priorities

The pragma *Interrupt_Priority* can be used to specify the ceiling priority of a protected object (Real-Time Systems [AAR95, Annex D]).

```
   pragma Interrupt_Priority (expression);
```

Omitting the expression is equivalent to specifying the ceiling priority of the system (*Interrupt_Priority'Last*). Interrupts of equal or lower priority are blocked while any operation of that protected object is in progress. To avoid priority inversion, any task calling an operation on that protected object has its priority raised to the ceiling priority while the operation is executed, reflecting the urgency of completing the operation so that interrupts will become unblocked. An interrupt handler executes at the priority of its protected object, which may be higher than the priority of the interrupt if the same protected object handles more than one kind of interrupt. In addition, the procedure *Set_Priority* provided by package *Ada.Dynamic_Priorities* can be used to dynamically modify this priority.

```
with Ada.Task_Identification;
package Ada.Dynamic_Priorities is

   procedure Set_Priority
     (Priority : System.Any_Priority;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   function Get_Priority
     (T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task)
     return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

6.2 GNAT Implementation

To foster a simple, efficient and multi-platform implementation, GNAT reuses the POSIX support for signals and adds the minimum set of run-time subprograms required to achieve the Ada semantics. This work is simplified because POSIX signals are delivered to individual threads in a multi-threaded process using much of the same semantics as for delivery to a single-threaded process [GB92, Section 5.1].

6.2.1 POSIX Signals

A POSIX signal[1] is a form of software interrupt which can be generated in several ways. A signal may be generated:

- By a hardware trap including division by zero, a floating-point overflow, a memory protection violation, a reference to a non-existent memory location or an attempt to execute an illegal instruction.
- Because a clock reaches a specified time, or a specified span of time has elapsed.
- By an asynchronous operation. Asynchronous input and output operations generate a signal when an operation completes, or if an operation fails.
- Because the user hits certain keys on the terminal that is controlling the process. Certain keys sequences allow the user to suspend, resume and terminate the execution of a process via signals.
- By a POSIX thread. POSIX threads may send a signal to another POSIX thread in the same process to notify it of an event, by calling *pthread_kill*.

Each POSIX thread has a signal mask: when a signal is generated for a thread and the thread has the signal masked, the signal remains pending until the thread unmasks it; the interface for manipulating the thread signal mask is *pthread_sigmask*. Only one pending instance of a masked signal is required to be retained; that is, if a signal is generated *N* times while it is masked the number of signal instances that are delivered to the thread when it finally unmasks the signal may be any number between 1 and N.

[1] The contents of this section are a summary of [DIBM96, section 2].

Each POSIX signal is associated with some *action*. The action may be to ignore the signal, terminate the process, continue the process, or execute a call to user-defined handler function (asynchronously and preemptively with respect to normal execution of the process). POSIX.1 specifies a default action for each signal. For most signals the application may override the default action by calling the function *sigaction*. The use of asynchronous handler procedures for signals is not recommended for POSIX threads, because the POSIX thread synchronization operations are not safe to be called within an asynchronous signal handler; instead, POSIX.1c recommends use of the *pthread_sigwait* function, which "accepts" one of a specified set of masked signals.

6.2.2 Reserved Signals

The definitions of "reserved" differs slightly between the ARM and POSIX. ARM specifies [AAR95, section C.3(1)]:

> *The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program unit can be connected to non-reserved interrupts.*

POSIX.5b/.5c specifies further [s-intman.adb]:

> *Signals which the application cannot accept, and for which the application cannot modify the signal action or masking, because the signals are reserved for use by the Ada language implementation. The reserved signals defined by this standard are:*
>
> - *Signal_Abort*
> - *Signal_Alarm*
> - *Signal_Floating_Point_Error*
> - *Signal_Illegal_Instruction*
> - *Signal_Segmentation_Violation*
> - *Signal_Bus_Error*
>
> *If the implementation supports any signals besides those defined by this standard, the implementation may also reserve some of those.*

The signals defined by POSIX.5b/5c that are not specified as being reserved are SIGHUP, SIGINT, SIGPIPE, SIGQUIT, SIGTERM, SIGUSR1, SIGUSR2, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, SIGIO, SIGURG and all the real-time signals.

The GNAT FSU Linux implementation handles 32 signals. In this case the reserved signals are:

Number		Name	REASON	Description
2	*	SIGINT	GNAT	Abort (used for CTRL-C)
4	*	SIGILL	POSIX (HW)	Illegal Instruction
5	*	SIGTRAP	GNAT	Trace trap
6	*	SIGABRT	GNAT	Tasks abortion
7	*	SIGBUS	POSIX (HW)	Bus error
8	*	SIGFPE	POSIX (HW)	Floating Point Exception
9		SIGKILL	POSIX	Abort (kill)
11	*	SIGSEGV	POSIX (HW)	Segmentation Violation
14		SIGALRM	POSIX	Alarm Clock
19		SIGSTOP		Stop
20	*	SIGTSTP	GNAT	User stop requested from tty
21	*	SIGTTIN	GNAT	Background tty read attempted
22	*	SIGTTOU	GNAT	Background tty write attempted
26		SIGVTALRM		Virtual timer expired
27	*	SIGPROF	GNAT	Profiling timer expired
31		SIGUNUSED		Unused signal

Signals marked with * are not allowed to be masked by the GNAT Run-Time. SIGINT can not be masked because it is used to terminate the Ada program when the CTRL-C sequence is pressed in the terminal that is controlling the process[2]. SIGILL, SIGFPE and SIGSEV can not be masked because they are used by the CPU to notify errors to the run-time. SIGTRAP is used by GNAT to enable debugging on multi-threaded applications. SIGABRT can not be masked because it is used by GNAT to implement the tasks abortion (described in chapter 8). SIGTTIN, SIGTTOU and SIGTSTP are not allowed to be masked so that background processes and IO behaves as normal C applications. Finally, SIGPROF can not be masked to avoid confusing the profiler.

[2]By keeping SIGINT reserved, the programmer allows the user to do Ctrl-C but, in the same way, disable the ability of handling this signal in the Ada program. GNAT Pragma *Unreserve_All_Interrupts* [BG01] gives the programmer the ability to change this behavior.

6.2.3 Architecture

Figure 6.1 presents the overall architecture of the GNAT implementation. In the GNARL level two package hierarchies are used to handle interrupts: the hierarchy of the standard Ada package (left side of the figure) and the hierarchy of the GNARL System package (right side of the figure).

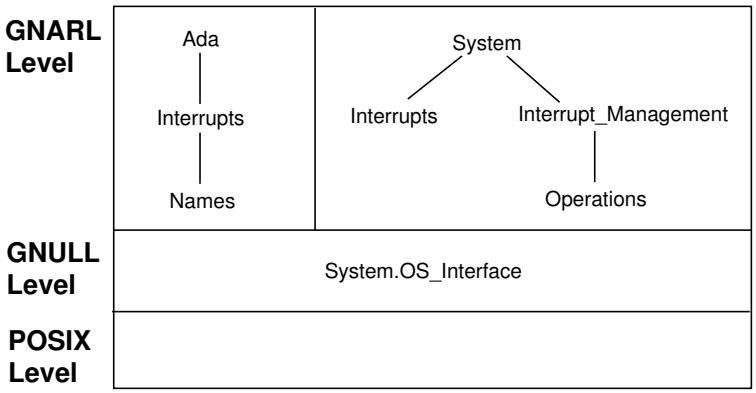

Figure 6.1: Architecture of the Implementation.

Package *Ada.Interrupts* is the standard Ada package described in section 6.1.2 (package used to attach and detach interrupt handlers in the non-nested style). Child package *Ada.Interrupts.Names* maps the high level Ada interrupts to the low level POSIX signals defined in package *System.OS_Interface* (35 Ada interrupts are mapped here to 32 POSIX signals).

Package *System.Interrupts* encapsulates the GNARL implementation of the signal handlers. It is a logical extension of the body of the standard package *Ada.Interrupts*. It is made a child of *System* to access various runtime system internal data and operations. Package *System.Interrupt_Management* associates a signal handler to the POSIX signals linked to Ada exceptions (SIGPFE and SIGILL signals raise Ada *Constrained_Error* exception and SIGSEGV signal raises Ada *Storage_Error* exception). Child package *Operations* is a low level package which issues calls to the GNULL level.

Let's see the type definitions associated with the high level *Ada.Interrupt_ID* data type. This simple example allows the reader to see the basic relations between these packages. *Ada.Interrupt_ID* type definition is based on the corresponding definition at *System.Interrupts*, which is based on the *Interrupt_ID* data type at *System.Interrupt_Management*; this data type is finally based on the corresponding SIGNAL type definition at the low-level package *System.OS_Interface*.

```
type Ada.Interrupt_ID
   is new System.Interrupts.Ada_Interrupt_ID;

type System.Interrupts.Ada_Interrupt_ID
   is new System.Interrupt_Management.Interrupt_ID;

type System.Interrupt_Management.Interrupt_ID
   is new System.OS_Interface.Signal;
```

6.2.4 Basic Data Structures

No matter the association style used, GNARL always uses the following tables indexed by the *Interrupt_ID* to handle interrupts.

- *Table of Reserved Signals*[3]: Booleans constant table[4] used to register reserved interrupts.

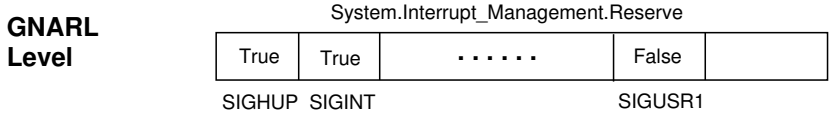

Figure 6.2: Reserved Interrupts Table.

- *User-defined Interrupt Handlers Table*[5]: Table used to register and unregister the reference to *User-Defined Interrupt-Procedures* (UDIP) during the life of the program. Each element of this table is a record with two fields: the access to the UDIP and a flag which remembers the association style (nested or non-nested).

Figure 6.3 represents one protected procedure attached to signal SIGUSR1 in nested style (static style). The GNAT compiler associates two subprograms P and N to each protected subprogram (described in section 4.2.4). As the reader can see, the run-time links the signal with the P subprogram: the reference to the P subprogram is stored in the corresponding field of the table, and the *Static* field is set to *True* to remember that it is a nested style association.

[3] *System.Interrupt_Management.Reserve*

[4] In the GNARL sources it is declared as variable just to be able to initialize it in the package body to aid portability.

[5] *System.Interrupts.User_Handler*

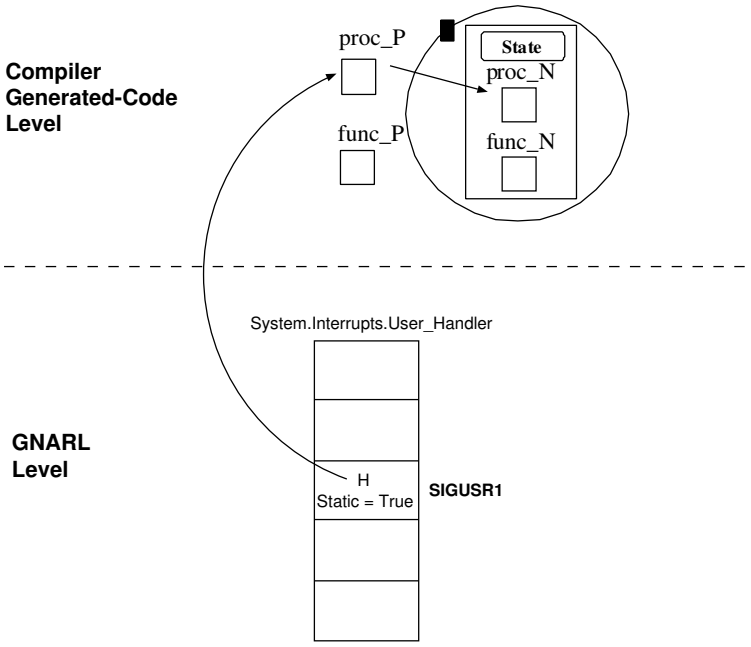

Figure 6.3: Table of User-Defined Interrupt-Handlers.

6.2.5 Attachment of Interrupt-Handling Protected Procedures

In the nested style the run-time must attach the UDIP to the signal when the protected object is elaborated. Thus the compiler adds one call to GNARL subprogram *Install_Handlers*[6] to the elaboration code of the protected object. This subprogram saves the previous handlers in one additional field of the object (*Previous_Handlers*[7]) and installs the new handlers.

To avoid penalizing all protected objects with this additional field, GNARL uses one data type for handling protected objects with no interrupt handlers (*Protection_Entries*[8], described in section 4.2.3), one type extension for protected objects with nested style interrupt handlers (*Static_Interrupt_Protection*[9]), and another type extension for protected objects with non-nested style interrupt handlers (*Dynamic_Interrupt_Protection*[10]). (In this latter case the type extension is only defined for homogeneity because the run-time does not add any additional field to the basic data type). Let's see the compiler transformation of protected objects for

[6] *System.Interrupts.Install_Handlers*
[7] *System.Interrupts.Previous_Handlers*
[8] *System.Tasking.Protected_Objects.Entries.Protection_Entries*
[9] *System.Interrupts.Static_Interrupt_Protection*
[10] *System.Interrupts.Dynamic_Interrupt_Protection*

these two latter cases:

Original Ada Code:

```
protected type PO (Discriminants) is
   procedure Handler;
   function  ...
   entry     ...
   pragma Attach_Handler (Handler, <INT-Number>);
private
   <Private_Data_Fields>
end PO;
```

Code transformation done by the GNAT compiler:

```
 1: type poV (Discriminants) is new Limited_Controlled with record
 2:    <Private_Data_Fields>
 3:    _object : aliased GNARL.Static_Interrupt_Protection (<Num>);
 4: end record;
 5:
 6: procedure Finalize (O : poV) is
 7: begin
 8:    --  Raise Program_Error to the queued tasks.
 9:    ...
10: end Finalize;
```

Let us now consider the compiler transformation of protected objects with non-nested style interrupt handlers.

Original Ada Code:

```
protected type PO (Discriminants) is
   procedure Handler;
   function  ...
   entry     ...
   pragma Interrupt_Handler (Handler);
private
   <Private_Data_Fields>
end PO;
```

Code transformation done by the GNAT compiler:

```
1: type poV (Discriminants) is new Limited_Controlled with record
```

```
 2:    <Private_Data_Fields>
 3:    _object : aliased GNARL.Dynamic_Interrupt_Protection (<Num>);
 4: end record;
 5:
 6: procedure Finalize (O : poV) is
 7: begin
 8:    --  Raise Program_Error to the queued tasks.
 9:    ...
10: end Finalize;
```

If we compare both translations, in line 3 we find the only difference: the data type used to define the *_object* field.

In case of nested style association, during the finalization of the protected object the run-time needs to restore the previous handlers (*Install_Handlers* does this work). In the non-nested style, nothing special needs to be done since the default handlers will be restored as part of task completion which is done just before global finalization.

In order to verify that all the non-nested style interrupt procedures have been annotated with pragma *Interrupt_Handler* ([AAR95, section C.3.2] requirement) the compiler adds calls to the GNARL subprogram *Register_Interrupt_Handler*[11] to register these interrupt procedures in a GNARL single-linked list. The *Head* and *Tail* of this list are stored in two GNARL variables[12] (cf. Figure 6.4). Every node keeps the address of one protected procedure associated with an interrupt in non-nested style. For simplicity, a single access to a protected procedure has been represented; however, each node has the access to its corresponding P subprogram. Before the attachment of one non-nested style interrupt handler to one signal, GNARL traverses this list to verify that the protected procedure is registered in the list; otherwise it raises the exception *Program_Error*.

6.2.6 Interrupts Manager: Basic Approach

The GNAT run-time uses one *Interrupts Manager*[13] task to serialize the execution of subprograms involved in the management of signals: attachment, detachment, replacement, etc. Figure 6.5 presents a simplified version of the automaton implemented by the Interrupt Manager. For simplicity we have considered only two basic operations: *Binding* and *Unbinding* User-Defined Interrrupt Procedures (UDIP) to interrupts.

[11] *System.Interrupts.Register_Interrupt_Handler*
[12] *System.Interrupts.Registered_Handler_Head* and *System.Interrupts.Registered_Handler_Tail*
[13] *System.Interrupts.Interrupt_Manager*

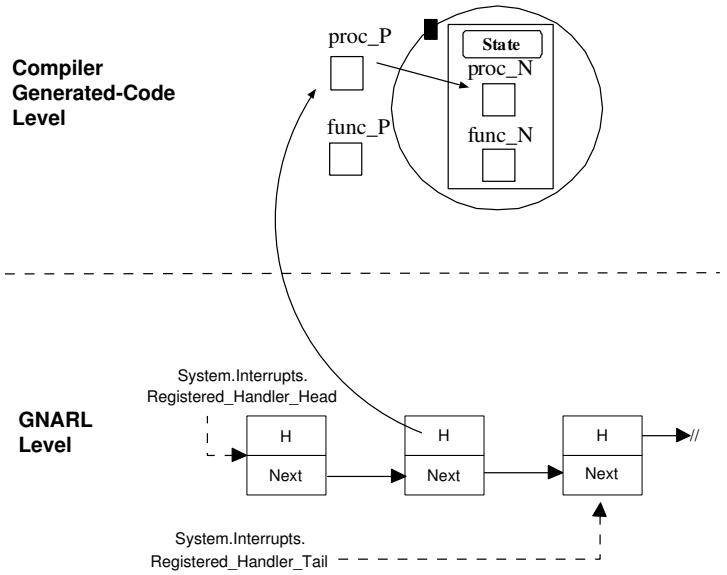

Figure 6.4: List of Interrupt Handlers in Non-Nested Style.

First the automaton calls GNARL subprogram *Make_Independent*[14] to do the *Interrupt Manager Task* independent of its masters. GNARL Independent tasks are associated with master 0, and their ATCBs are not registered in *All Tasks List* (described in section 2.2); thus they last until the end of the program. After the signal mask is set, the automaton goes to one state in which it waits for the next signal management operation.

- In case of signal *Binding*, GNARL saves the reference to the UDIP in its table, and blocks the POSIX signal (this allows GNARL to catch the signal with the *sigwait* POSIX service).
- In case of signal *Unbinding*, the reference to the UDIP is removed from the table, the POSIX default action is set, and the signal is unblocked.

6.2.7 Server Tasks: Basic Approach

The Ada run-time must provide a thread to execute the UDIP. There is a choice between dedicating one server task for all signals and providing a server task for each signal. The former approach looks attractive, since it saves run-time space, but it blocks other signals during the protected procedure call. This may result in

[14] *System.Task_Utilities.Make_Independent*

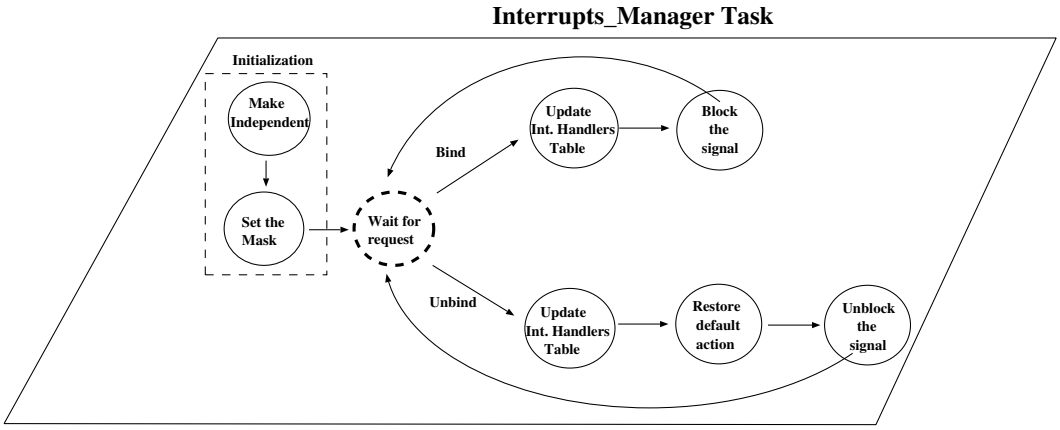

Figure 6.5: Basic Automaton Implemented by the *Interrupts Manager*.

delayed or lost signals. For this reason, GNARL provides a separate *Server Task*[15] for each signal [DIBM96].

Instead of create/abort *Server Tasks* when the user-defined interrupt handlers are attached/detached, GNARL keeps them alive until the program terminates. Thus they are reused by all UDIPs associated with the same interrupt during the life of the program. The run-time has a *Server_ID Table*[16] which saves *Server Tasks* references (cf. Figure 6.6).

Figure 6.7 presents a simplified version of the Server Tasks Automaton.

6.2.8 Interrupt-Manager and Server-Tasks Integration

Previous sections have been concerned with the basic functionality of the *Interrupt Manager Task* and the *Server Tasks*. However, the GNARL implementation is a little more complex because:

1. Ada nested style of interrupts implies that UDIPs are dynamically attached and detached to signals in the elaboration and finalization of protected objects. Therefore:

   (a) If no UDIP is registered GNARL must take the default POSIX action, and the simplified implementation of the *Interrupt Manager* did not consider POSIX default actions (cf. Figure 6.5).

[15] *System.Interrupts.Server_Task*
[16] *System.Interrupts.Server_ID*

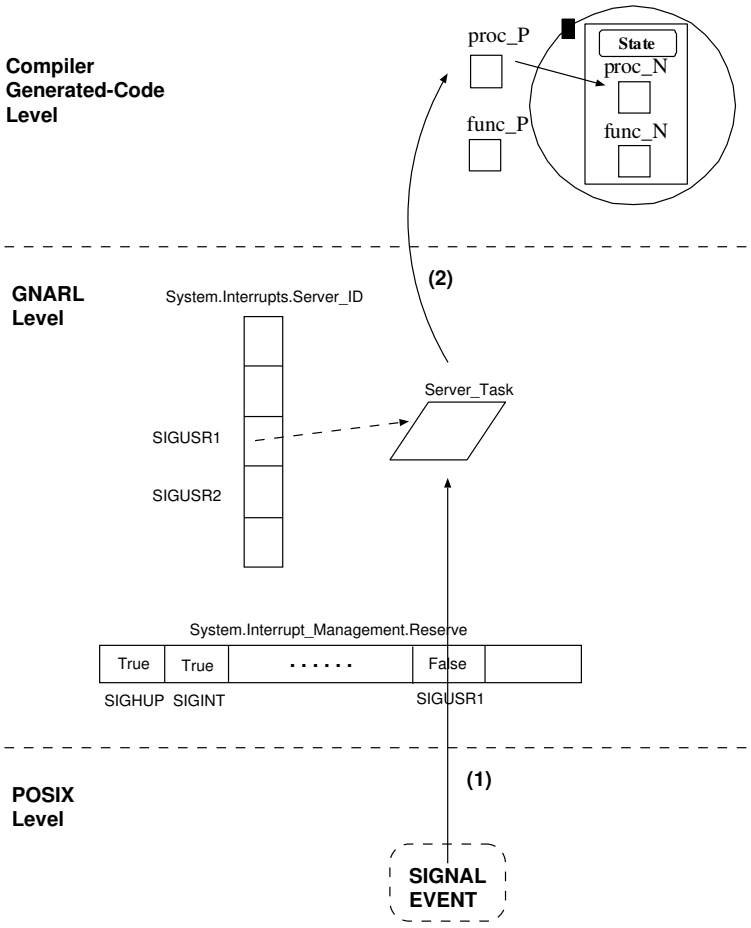

Figure 6.6: Server Tasks Signal Handling.

(b) When one UDIP is registered the signal is programmed to be handled by the UDIP. Following UDIPs registered to the same signal replace previous UDIPs.

(c) If all UDIPs are detached, GNARL must again take the default POSIX action. The previous implementation can not achieve this effect so long as the *Server Task* is sitting on the *sigwait*. Even if the POSIX *sigaction* command is used to set the asynchronous signal action to the default, that action will not be taken unless the signal is unmasked, and GNARL can not unmask the signal while the *Server Task* is blocked on *sigwait* because in POSIX.1c the effect is undefined. Therefore, GNARL must wake up the *Server Task* and cause it to wait on some operation instead for which it is safe to leave the signal unmasked, so that the default action can be taken [DIBM96].

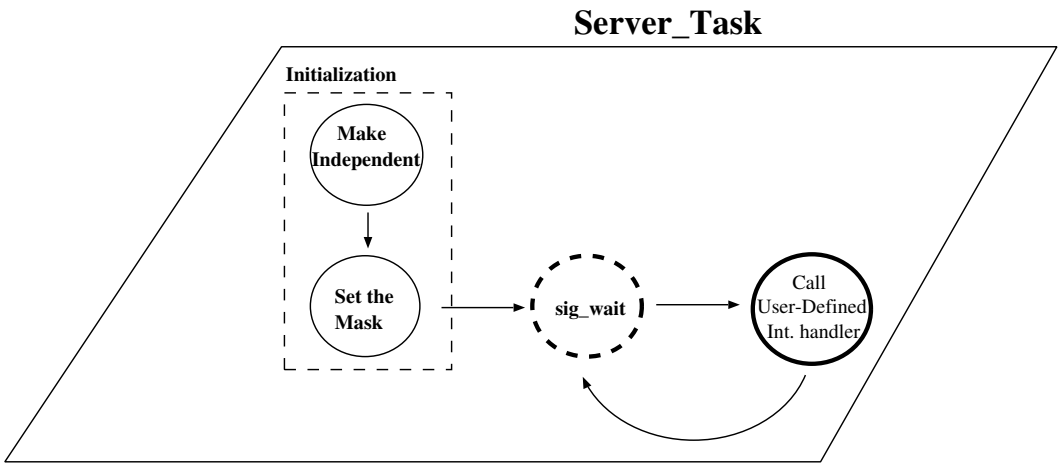

Figure 6.7: Basic Automaton Implemented by the Server Tasks.

2. GNARL must protect data structures shared by the *Interrupts Manager Task* and the *Server Tasks*. Therefore, some locks must be added.

The second requirement (*locks*) is easy to solve by means of POSIX mutexes. However, the first requirement is more complex. So let's focus our attention on the GNARL solution of the first requirement.

In order to better understand the GNARL implementation, we need to simplify the *Server Tasks Automaton* to its main states:

- *State 1*: The *Server Task* provides the POSIX default behavior of the signal.
- *State 2*: The *Server Task* has been programmed to call one UDIP.

In order to notify the automaton that it must jump from *State 1* to *State 2* GNARL uses one POSIX *Condition Variable*; in order to force the automaton to jump from *State 2* (waiting in the POSIX *sigwait* operation) to *State 1* the POSIX signal SIGABORT is used (this signal is used to kill the POSIX thread, and thus forces the *Server Task* to return from the POSIX *sigwait* operation). Figure 6.8 presents this automaton.

If we add these new transitions to our basic *Task Server Automaton* (cf. Figure 6.7) we have the real automaton implemented in GNARL (cf. Figure 6.9). In order to help the reading of the automaton all the states have been numbered. Inside dotted rectangles we find the states associated with the simplified states of the previous example (*State_1* and *State_2*).

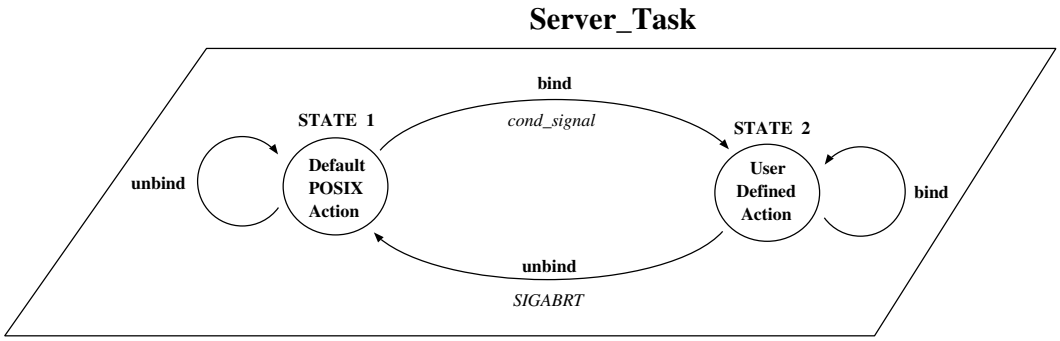

Figure 6.8: Simplified Server Tasks Automaton.

After the initializations (states numbered 1 to 3), the automaton verifies if any UDIP has been registered by the *Interrupt Manager* (state 4). Initially, because no UDIP has been registered, it takes the POSIX default action (state 9) and waits in the *Condition Variable* (*cond_wait*, state 10) until some UDIP is registered by the *Interrupt_Manager*.

When any UDIP is registered, the *Interrupt_Manager* signals the *Condition Variable* and the *Server Task Automaton* jumps to state 4, checks if some UDIP has been registered (now this evaluates to *True*) and jumps to state 5 to wait for the next signal occurrence. When the signal is received, it again checks if the UDIP is still registered (state 6), because it may have been removed by the *Interrupt Manager* while the automaton was waiting for the signal. Then it calls the UDIP (state 7) and again jumps to state 4.

While the *Server Task* is in state 5 waiting for the signal occurrence, it may happen that all UDIPs have been removed the *Interrupt Manager*. In this case the *Interrupt Manager* sends the SIGABRT signal to the *Server Task* to force it to jump to state 9. This signal wakes up the *Server Task Automaton*, which jumps to state 8 to reply to the *Interrupt Manager* with the same signal to inform it is not in state 5 (waiting for the signal). After this notification the automaton jumps to state 4 and, because no UDIP is found, it jumps to state 9.

6.3 Summary

In this chapter we have dealt with the main aspects related to Interrupts Management. Although Ada allows us to attach a task entry to an interrupt, nowadays this is considered an obsolescent feature of the language. Thus, we have only

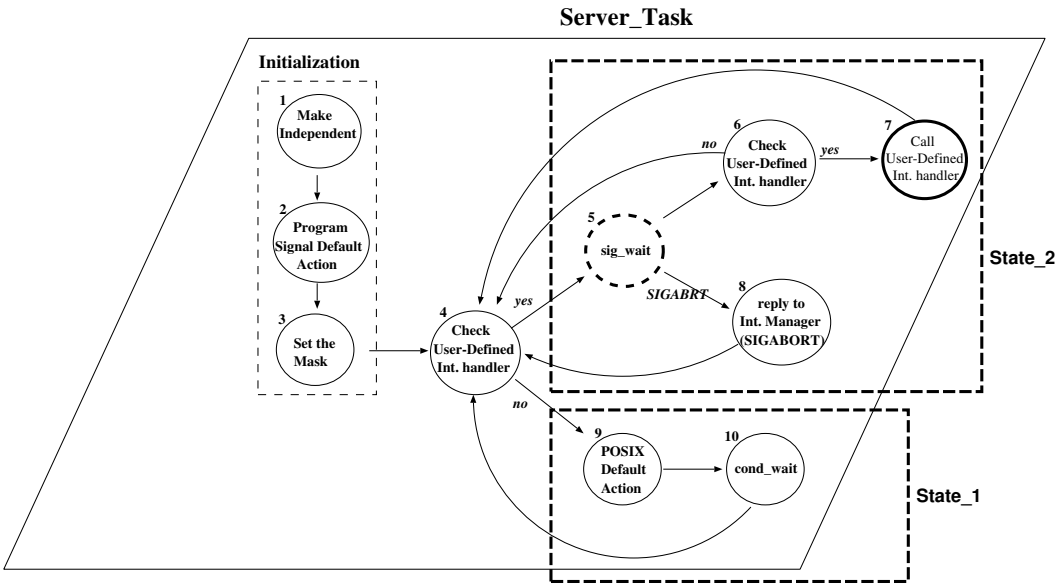

Figure 6.9: Server Tasks Automaton.

discussed the attachment of User-Defined Protected-Procedures to interrupts. The main features of the GNAT implementation are:

- GNARL associates Ada interrupts to POSIX signals.
- Each signal has a *Task Wrapper* responsible for the execution of the User-Defined Protected-Procedures.
- The protected subprogram P associated with the protected subprogram is attached by GNARL to the corresponding *Task Wrapper*.
- Ada provides two ways to attach a protected procedure to an Ada interruption: nested style (by means of the pragma *Attach_Handler*) and non-nested style (by means of the pragma *Interrupt_Handler*).
  - When the nested style is used, GNARL adds one field to the run-time information of the protected object to save and restore the previous handler.
  - When the non-nested style is used, a dynamic link list is used to register non-nested style UDIPs. This list allows GNARL to verify that only non-nested UDIPs have been marked with the right pragma.
- An *Interrupt Manager Task* is used to serialize all the signal-management operations.

Chapter 7

Exceptions

During the execution of a program, events or conditions may occur which might be considered "exceptional". With commercial or numeric computing, such conditions can be catered for by an appropriate run-time error message followed by program termination. This is not acceptable with embedded systems, where the software should be tolerant of both hardware and software faults. Two broad classifications of exceptions can be isolated [BW98, section 1.4]:

- *Error conditions* — arithmetic overflow, storage exhaustion, array-bound violation, subrange violations, peripheral time-outs, etc.
- *Abnormal program conditions* — errors in user input data, need for special algorithms to deal with singularities, etc.

In order to deal with error conditions, the run-time system must bring such errors to the program attention [BW98, section 1.4].

7.1 Ada Model of Exceptions

An exception represents a kind of exceptional situation; an occurrence of such a situation (at run-time) is called an *Exception Occurrence*. To raise an exception is to abandon normal program execution thus drawing attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *Handling* the exception [AAR95, section 11].

7.1.1 Exception Declaration

The syntax to declare an exception is [AAR95, section 11-1]:

```
exception_declaration ::= defining_identifier_list : exception;
```

7.1.2 Raise Statement

The syntax to raise an exception is [AAR95, section 11-2]:

```
raise_statement ::= raise [exception_name];
```

When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; that is, any portions of the execution that have not yet taken place are not performed. A raise statement without an exception name is a re-raise statement and therefore must be within an exception handler.

7.1.3 Exception Handling

When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an applicable exception handler, if any. To *handle* an exception occurrence is to respond to the exceptional event [AAR95, section 11-4]. The syntax to declare an exception handler is [AAR95, section 11-2]:

```
handled_sequence_of_statements ::=
     sequence_of_statements
   [exception
      exception_handler
      exception_handler]

exception_handler ::=
   when [choice_parameter_specification:]
        exception_choice
        | exception_choice =>
        sequence_of_statements

choice_parameter_specification ::= defining_identifier

exception_choice ::= exception_name | others
```

If an exception occurrence is not handled (or is re-raised) it is propagated to the innermost dynamically enclosing execution [AAR95, section 11-4]. If an exception occurrence is unhanded in a task body, the exception does not propagate further (because there is no dynamically enclosing execution). If the exception occurred during the activation of the task, then the activator would raise *Tasking_Error* [AAR95, section 11-4].

7.1.4 Package Ada.Exceptions

The following language-defined library provides additional facilities for exceptions handling[a-except.ads]:

```
package Ada.Exceptions is

   type Exception_Id is private;
   Null_Id : constant Exception_Id;

   type Exception_Occurrence is limited private;
   type Exception_Occurrence_Access is access all Exception_Occurrence;

   Null_Occurrence : constant Exception_Occurrence;

   function Exception_Name (X : Exception_Occurrence) return String;
   --  Same as Exception_Name (Exception_Identity (X))

   function Exception_Name (Id : Exception_Id) return String;

   procedure Raise_Exception (E : in Exception_Id; Message : in String := "");

   function Exception_Message (X : Exception_Occurrence) return String;

   procedure Reraise_Occurrence (X : Exception_Occurrence);

   function Exception_Identity (X : Exception_Occurrence) return Exception_Id;

   function Exception_Information (X : Exception_Occurrence) return String;

   procedure Save_Occurrence
     (Target :    out Exception_Occurrence;
      Source : in     Exception_Occurrence);

   function Save_Occurrence
     (Source : in Exception_Occurrence)
      return Exception_Occurrence_Access;

private
   . . .
end Ada.Exceptions;
```

Each distinct exception is represented by a distinct value of type *Exception_Id*. *Null_Id* does not represent any exception, and is the default initial value of the type *Exception_Id*. Each occurrence of an exception is represented by a value of the type *Exception_Occurrence*. Similarly, *Null_Occurrence* does not represent any exception occurrence; it is the default initial value of type *Exception_Occurrence*.

7.2 GNAT Implementation

The following paragraphs describe Data Structures used by the Run-Time System to manage all the exceptions.

7.2.1 Exception Identifier and Exception Occurrence

GNAT implements the exception identifier as an access to a record (*Exception_Data_Ptr*[1]). Figure 7.1 presents the fields of this record.

Exception_Data

Handled_By_Others	: Boolean
Lang	: String (1 .. 3)
Name_Length	: Natural
Full_Name	: String_Ptr
HTable_Ptr	: Exception_Data_Ptr

Figure 7.1: Exception Identifier.

The field *Handled_By_Others* is used to differentiate the user-defined exception from the run-time internal exceptions (i.e. task abortion) which can not be handled by the user-defined exception handlers. The field *Lang* defines the language where the exception is declared (by default "Ada"). The next two fields are used to store the full name of the exception. This name is composed of a prefix (the full path of the scope where the exception is declared) and the exception name. The last field is used to create linked lists of exception identifiers (describe in section 7.2.2).

When an exception is raised, the corresponding exception occurrence is stored by the GNAT run-time in the *Compiler_Data* field of the ATCB. The data type of this field is a record; the *Current_Excep*[2] field of this record stores the exception occurrence.

The *Exception_Raised* field is set to *True* to indicate that this exception occurrence has actually been raised. When an exception occurrence is first created, it is set to false; then, when it is later processed by the GNARL subprogram

[1] *System.Standard_Library.Exception_Data_Ptr*
[2] *System.Soft_Links.TSD*

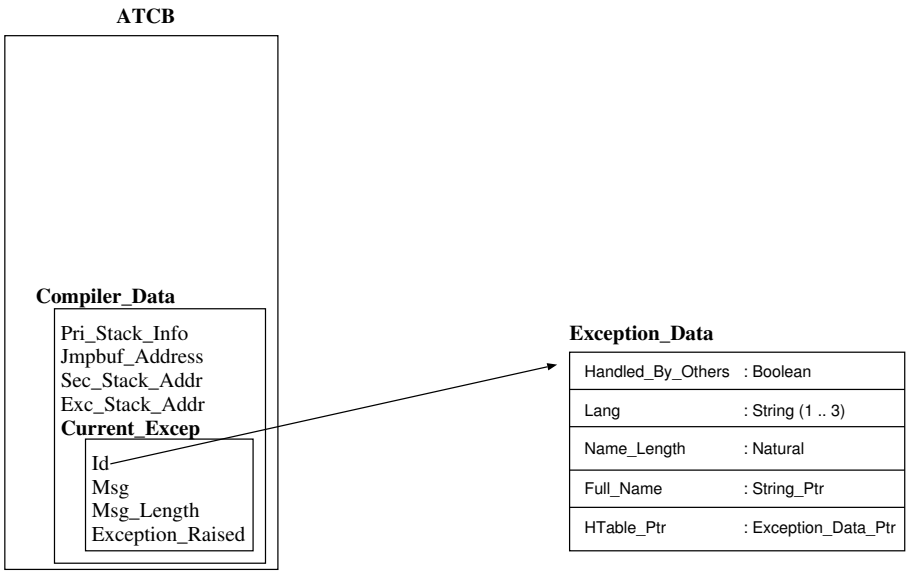

Figure 7.2: Occurrence Identifier.

Raise_Current_Exception[3], it is set to *True*. This allows the run-time to distinguish if it is dealing with an exception re-raise.

7.2.2 Exceptions Table

Because the visibility rules of Ada exceptions (an exception may not be visible, though handled by the **others** handler, re-raised and then again visible to some other calling scope) a global table must be used (*Exceptions_Table*[4]). In order to handle the exceptions in an efficient way, the Ada run-time uses a hash table (cf. Figure 7.3).

As the reader can see, an accesses table to the exception identifiers is used. A simple linked list of exception identifiers is used to handle collisions. The field *HTable_Ptr*[5] is used to link the exception identifiers.

When an exception is raised in a task, the corresponding exception identifier must be found. Therefore the hash function is evaluated, and the resulting linked list is traversed to look for the exception identifier. Then its reference is stored in the ATCB of the task. This reference is kept in the ATCB until the exception is

[3] *Ada.Exceptions.Raise_Current_Excep*
[4] *System.Exception_Table*
[5] *System.Standard_Library.TSD*

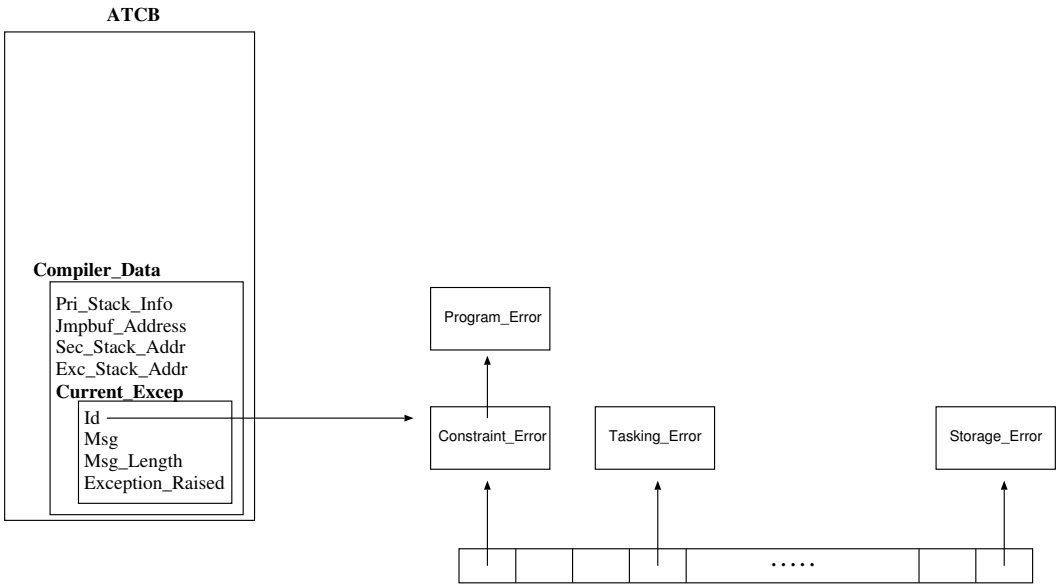

Figure 7.3: Hash Table.

handled (though the exception may not be visible in some exception handlers).

The elements in the hash table are never removed; they are kept until the end of the program.

7.2.3 Exception Declaration

The compiler translates an exception declaration to a variable declaration of type *Exception_Data*. In order to distinguish among the exceptions with the same name, the name of an exception is made up of the complete path of its scope. For instance, if we declare the exception *My_Error* in the function *Locate*, the name assigned by the compiler will be *Locate.My_Error*. Since it is necessary to include the associated data in the exceptions *hash* table, the compiler inserts a call to the GNARL subprogram *Register_Exception*[6]. This subprogram evaluates the hash function from the exception name and then inserts its data record at the beginning of the corresponding linked list. This record remains in the list until the end of the program.

[6] *System.Exception_Table.Register_Exception*

7.2.4 Exception Handler

This section presents a general overview of the exception handlers implementation done in GNAT via the POSIX *setjump/longjmp* mechanism. The long jump is a POSIX operation which transfers control from the current point of execution to the point of a previous call to *setjmp()* at a lower nesting level of stack frames. Specifically, *setjmp()* can be called as the condition of an **if** statement. In essence, *setjmp()* saves the state of the thread at the point of call into a *jump buffer* variable and returns the value zero. The *longjmp()* operation reloads the state from the jump buffer which transfers control to the *setjmp()* call causing *setjmp()* to return again, but this time with a non-zero value. Thus the *longjmp()* can be used to roll back a computation to an earlier conditional branching point and take the other branch (in our case, the exception handler) [GMB93, section 4.2].

The GNAT compiler translates the following Ada code

```
begin
   User_Code;
exception
   when Exception1 =>
      Exception1_Code;
   when Exception2 =>
      Exception2_Code;
   . . .
   when others =>
      Others_Code;
end;
```

. . . to the following code:

```
declare
   Env : Jmp_Buf;
begin
   if setjmp(Env) then
      User_Code;
   else
      declare
         Exception_ID: GNARL.Exception_ID;
      begin
         Identify the raised exception;
         GNARL.Undefer_Abortion;
         case Exception_ID is
            when Exception1 =>
                 Exception1_Code;
            when Exception2 =>
                 Exception2_Code;
            . . .
            when others =>
```

```
                    Others_Code;
            end case;
         end; -- declare
      end if;
end;
```

7.2.5 Raise Statement

Ada allows an exception to be raised in two different ways: (1) by means of the **raise** statement and (2) by means of the procedure *Ada.Exceptions.Raise_Exception* which allows the programmer to associate a message to the exception. In both cases, the compiler generates a call to a GNARL function which carries out the following actions:

1. To fill the ATCB exception occurrence.
2. To defer the abortion.
3. If there is one exception handler installed, then jump to it.
4. Otherwise (no exception handler can be called) terminate the execution of the program.

7.3 Summary

In this chapter the basic concepts of the GNAT exception handling implementation has been presented.

- The exception ID is an access to a record where the full name of the exception is stored.
- The exception occurrence is stored in the ATCB.
- All the exceptions are stored in a hash table.
- The exception handler can be implemented by means of the *setjmp/longjmp* POSIX mechanism.

Chapter 8

Abortion

When an error condition is detected in a program two basic mechanisms allow us to handle the error: *exceptions*, which can be used when the task itself detects the error condition and can explicitly raise (and handle) the exception, and the *abortion* which is used when another task detects the error condition. Ada provides two basic mechanisms to abort some execution: the **abort** statement, and the *Asynchronous Transfer of Control* (ATC).

8.1 Ada Abortion

8.1.1 Abort Statement

The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible. Tasks which are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut-down, and hence the ARM [AAR95, section 9.8] requires is that the task terminate before it next interacts with other tasks [BW98, section 10.2]. The syntax of the abort statement is:

```
abort_statement ::= abort task_name , task_name;
```

Each aborted task becomes *abnormal* and any non-completed tasks that depend upon an aborted task also become abnormal. Once all named tasks are

marked as abnormal, then the abort statement is complete, and the task executing the abort can continue. It does not wait until named tasks have actually terminated [BW98, section 10.2].

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted [BW98, section 10.2].

If a construct which has been aborted is blocked outside an abort-deferred operation (other than at an entry call), the construct becomes abnormal and is immediately completed. Other constructs must complete no later than the next *abort completion point* (if any) that occurs outside an abort-deferred operation. These are [BW98, section 10.2]:

- The end of activation of a task.
- The point where the execution initiates the activation of another task.
- The start or end of an entry call, accept statement, delay statement or abort statement.
- The start of the execution of a select statement, or of the sequence or statements of an exception handler.

Certain actions must be protected in order that the integrity of the remaining tasks and their data be assured. The following operations are defined to be abort-deferred [BW98, section 10.2.1]:

- A protected action.
- Waiting for an entry call to complete.
- Waiting for termination of dependent tasks.
- The execution of an "initialize" procedure, a "finalize" procedure, or an assignment operation of an object with a controlled part.

8.1.2 Asynchronous Transfer of Control

The asynchronous transfer of control allows the caller to continue executing some code while the entry call is waiting to be attended. The syntax is [AAR95, section 9.7.4]:

```
asynchronous_select ::=
  select
    triggering_alternative
  then abort
    abortable_part
  end select;

triggering_alternative ::=
    triggering_statement
    [sequence_of_statements]

triggering_statement ::=
    entry_call_statement
  | delay_statement

abortable_part ::= sequence_of_statements
```

If the triggering statement is queued (due to an entry call statement or to a delay statement) the abortable part starts its concurrent execution. When one of the parts finishes its execution it aborts the execution of the other part. This provides local abortion which is potentially cheaper than the abortion of the entire task.

There is a restriction on the sequence of statements that can appear in the abortable part. It must not contain an accept statement. The reason for this is to keep the implementation as simple as possible [BW98, section 10.3.2].

The asynchronous select statement can be nested. The abortable part may itself contain another asynchronous select statement. In this case, the ATC may have to propagate to an outer asynchronous select scope if its triggering statement completes. In the process, all inner asynchronous select statements have to be aborted [GMB93, section 4.1].

8.2 GNAT Implementation

8.2.1 Abort Deferral

At some predefined points the abortion can not be immediately attended. For example, abort deferral is required by the language for protected actions and finalization routines. It is generally also required during the execution of the Ada run-time, to ensure the integrity of run-time data structures. Implementing abort deferral can be divided into two parts [GB94b, Section 3.3]:

- Determining whether abort is deferred for a given task, at the point it is targeted for abortion.
- Ensuring deferred aborts are processed immediately when abort-deferral is lifted.

In general, the determination of whether a given task is abort-deferred must be carried out by the task itself. In a single-processor system, it may be possible for the task initiating an abort to determine whether the target task is abort-deferred. However, in a multi-processor system, or a single processor system where the Ada run-time is not in direct control of task scheduling, this is not possible. The abort-deferral state of the target task may change between the point it is tested and the point the target task is interrupted [GB94b, Section 3.3].

There are two obvious techniques for recording whether a task is abort-deferred. One technique is sometimes termed PCmapping. The compiler and link-editor generate a map of abort-deferred regions. Whether the task is abort-deferred can then be determined by comparing the current instruction-pointer value, and all the saved return addresses of active subprogram calls, against the map. To ensure the abort is processed on exit from the abort-deferred region, one overwrites the saved return address of the outermost abort-deferred call frame with the address of the abort-processing routine (saving the old return address elsewhere). The test for abort deferral may take time proportional to the depth of subprogram call nesting, but that occurs only if an ATC is attempted. Until that occurs, no runtime overhead is incurred for abort deferral. A restriction of this method is that abort-deferred regions must correspond to callable units of code. Another restriction is that the subprogram calling convention is constrained to (1) ensure the return addresses are always in a predictable and accessible location and (2) ensure this data is always valid, even if the calling sequence is interrupted. Unfortunately, that is not true for some architectures [GB94b, Section 3.3].

In the other technique the task increments and decrements a deferral nesting level (e.g. in a dedicated register or the ATCB), whenever it enters and exits an abort-deferred region. On exit from such a region, if the counter goes to zero, the task must check whether there is a pending abort and, if so, process the abort. This deferral-counter method imposes some distributed overhead on entry and exit of abort-deferred regions, but allows GNARL quick checking [GB94b, Section 3.3]. This is the technique used by the GNAT run-time. GNAT *Undefer_Abort*[1] subprogram is the universal polling point for deferred processing. It is responsible for:

1. *Base priority changes*. It verifies if some priority change was requested (*Pending_Priority_Change* ATCB field). In this case, the task yields the processor so that the POSIX scheduler chooses the next task to execute.
2. *Exception handling*. It verifies if there is some pending exception to raise (*Exception_To_Raise* ATCB field).
3. *Abort/Asynchronous Transfer of Control (ATC)*. It verifies if the internal exception *Abort_Signal* must be raised.

If some request is made to modify the priority of a task, or to abort an abort-deferred task, the ATCB field *Pending_Action* is set to True (and the abortion will be done later by the GNARL *Undefer_Abortion* procedure).

8.2.2 Abort Statement

In general, processing an abort requires unwinding the stack of the target task, rather than immediately jumping out of the aborted part (or killing the task, in the case of entire-task abortion). There may be local controlled objects, which require the execution of a finalization routine. There also may be dependent tasks, which require the aborted processing block until they have been aborted, finalized, and terminated. The finalization must be done in LIFO order and the stack contexts of the objects requiring finalization must be preserved until the objects are finalized [GB94b, Section 3.4]

The GNARL implementation of the Ada abort statement is made up of:

- One flag in the ATCB (*Aborting*). While set, this flag prevents a race between multiple aborters and the aborted task.

[1] *System.Tasking.Initialization.Undefer_Abort*

- One internal exception (*_Abort_Signal*). This exception is not visible to user code and can only be caught by run-time system code. When this exception is raised, it propagates back with finalization being done along the way [BG93, Section 5.2.3]. In order to avoid the handling of this exception by the **others** exception handlers, one additional field has been added to the GNARL data type used to identify the exceptions: *Not_Handled_By_Others*. It is only set to *True* in this special case.
- One POSIX signal (SIGABRT), which can not be masked.

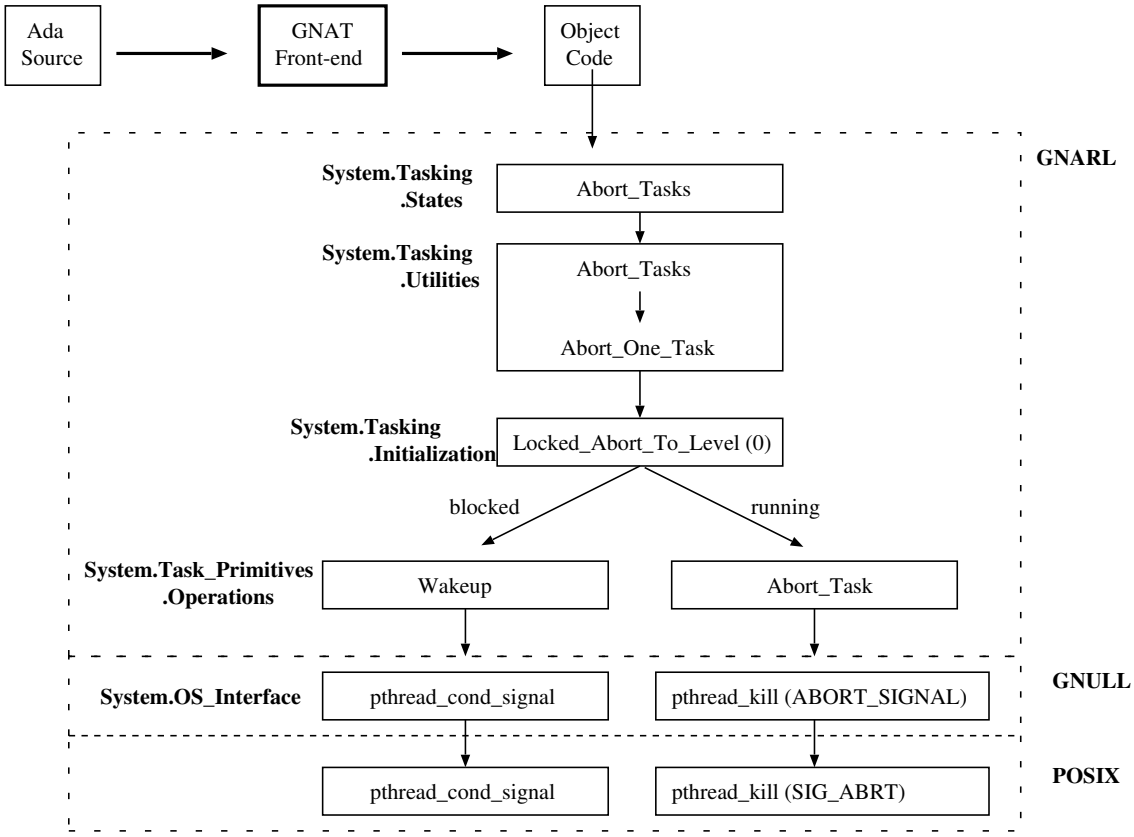

Figure 8.1: GNARL Subprograms for the Abort Statement.

Figure 8.1 presents the sequence of run-time subprograms involved in the task abortion. The GNARL procedure *Locked_Abort_To_Level*[2] sets to true the ATCB flag *Pending_Action.* and, depending on the current state of the target task (blocked or running) it calls *Wakeup*[3] or *Abort_Task*[4].

[2] *System.Task_Initialization.Locked_Abort_To_Level*
[3] *System.Task_Primitives.Operations.Wakeup*
[4] *System.Task_Primitives.Operations.Abort_Task*

- If the task to be aborted is in a *blocked* state (task states are described in section 2.1.1), then it is in a deferred abortion section. After the aborted task is waked up, it continues its execution and executes the GNARL *Undefer_Abortion*[5] subprogram. At this moment the *Pending_Action* ATCB flag will be checked. Because it is true, the ATCB flag *Aborting* is set to true and internal exception *_Abort_Signal* is raised in the aborted task.

- If the task is in the *running* state then the aborter sends to it the signal *SIG_ABRT* and then the *Abort Handler*[6] asynchronously raises the internal exception *_Abort_Signal* in the aborted task.

In both cases the internal exception *Abort_Signal* can not be handled by the user defined exception handlers and unwinds the stack of the aborted task.

8.2.3 Asynchronous Transfer of Control

An implementation of ATC must address the following issues [GB94b, section 3]:

- Interruption of the target task and abortion initiation.
- Deferral of abort over certain regions.
- Execution of finalization procedures for any local objects in the aborted part, each in its correct context.
- Finding the proper location and context to continue execution, after the ATC.
- Handling nested scopes, including nested asynchronous select statements.
- Ensuring safety of compiler-generated code sequences, including subprogram call and return when interrupted by ATC.

Since ATC is not likely to be used in most (non real-time) Ada programs, a key objective of any implementation should be to impose little or no distributed overhead for the existence of this feature. Subject to this constraint, the implementation of ATC should be as efficient as possible [GB94b, section 3].

[5] *System.Task_Initialization.Undefer_Abortion*
[6] *System.Task_Primitives.Operations.Abort_Handler*

There are two implementation models for the ATC, which can be classified according to the number of threads required for its implementation. One thread model and two threads model.

- One-Thread model

  In this model the caller starts with the triggering statement (basically it executes the steps described in 3.2.2). If it can be completed immediately, the abortable part is never executed at all; the task continues with the triggered statements and then jumps to the end of the **select** statement. Otherwise, if the entry call remains abortably queued the task does not suspend execution; instead it proceeds to execute the abortable part. If triggering event occurs before the task completes the abortable part, the task is interrupted and forced to first execute finalization code for the abortable part, then transfer control to the triggered statements. Otherwise, if the abortable part completes before the completion of the entry call, an attempt is made to cancel the entry call and, if successful, jumps to the end of the **select** statement [BW98, section 10.3.2] [GB94b, section 3.1].

  Similarly, if the triggering statement is a delay statement, the delay time is calculated and, if it has not passed, the abortable part is then executed. If this finishes before the delay time expires, the delay is canceled and the execution of the asynchronous select statement is finished [BW98, section 10.3.2].

  If the cancellation of the triggering event fails, because the protected action or rendezvous has started, or has been requeued (without abort), then the asynchronous select statement waits for the triggering statement to complete before executing the optional sequence of statements following the triggering statement [BW98, section 10.3.2].

- Two-Thread model

  In this model, before blocking on the triggering statement, the task executing the asynchronous **select** creates an agent thread of control to execute the abortable part. The ATC is carried out by aborting the agent thread, if the original thread wakes up at the triggering statement before the agent completes [GB94b, section 3.1].

Proponents of the two-thread model have argued that it simplifies the implementation of several aspects. One is that it preservers two useful invariants of the original Ada tasking model namely: (1) a thread that is waiting for an event is not executing; (2) a thread never waits for more than a timeout and one other

event. Another simplification is that the two thread model eliminates the need for one thread to asynchronously modify another thread's flow of control, which is not possible in some execution environments. If there is a way to kill a thread, it should be sufficient to simply kill the agent thread and wake up the client [GB94b, section 3.1].

The two-thread model seems to complicate the implementation at least as much as it simplifies it. It violates a key invariant of Ada tasking, that there is a one-to-one correspondence between tasks and threads of control. This assumption pervades the semantics, and is the foundation of existing Ada tasking implementations. Loss of this invariant has many ramifications. Among these, data that previously could only be accessed by one thread of control becomes susceptible to race conditions. Thus, there are new requirements for synchronization, and new potential for deadlock within a single task. Also, just killing the agent thread is not such a simple solution as it might seem. There remains the problem of how to execute the agent's finalization code. If the operation that kills a thread does not support finalization, some other thread must perform the finalization. To do so, it must wait for the killed thread to be terminated to be able to obtain access to the run-time stack of the terminated thread. The latter may not be possible in systems where killing a thread also releases its stack space [GB94b, section 3.1].

The one-thread model can be implemented using a signal and *longjmp()*. The trigger (entry call or delay) is pending on the thread while the abortable part is executed. If the abortable part completes first, the pending trigger is removed. If the trigger completes, an abortion signal is sent to the thread. The signal handler for the abortion signal then transfers control out of the abortable part into the triggered statements [GMB93, section 4.3]. Due to the disadvantages of the two-threaded model, GNAT implements the one-thread model. The non-local jump is performed by raising the internal exception in a signal handler. The propagation of this exception aborts one or more levels of abortable parts [GB94a, section 4.3].

ATCs can be nested. This allows a task to issue another entry call while it is waiting to complete a previous entry call (in the abortable part of the ATC). Therefore, the Ada run-time must store all these pending entry calls. The GNAT run-time associates an *Entry Call Stack* to each Ada task (*Entry_Calls* ATCB field—figure 8.2). The top of this stack (*ATC_Nesting_Level* ATCB field) is initialized to 1, indicating that the task has not issued any entry call. Before an entry call, the task increments ATC_Nesting_Level. Therefore, level 1 is not used. The *Pending_ATC_Level* field is used to signal an abort. In order to distinguish between the *Abort* statement and the end of an asynchronous request the GNAT run-time defines the following rule:

- In the case of an **abort** statement, *Pending_ATC_Level* is set to 0.
- In the case of an ATC finalization, *Pending_ATC_Level* is set to the level in which the caller was just before the entry call was made (*ATC_Nesting_Level* minus one).

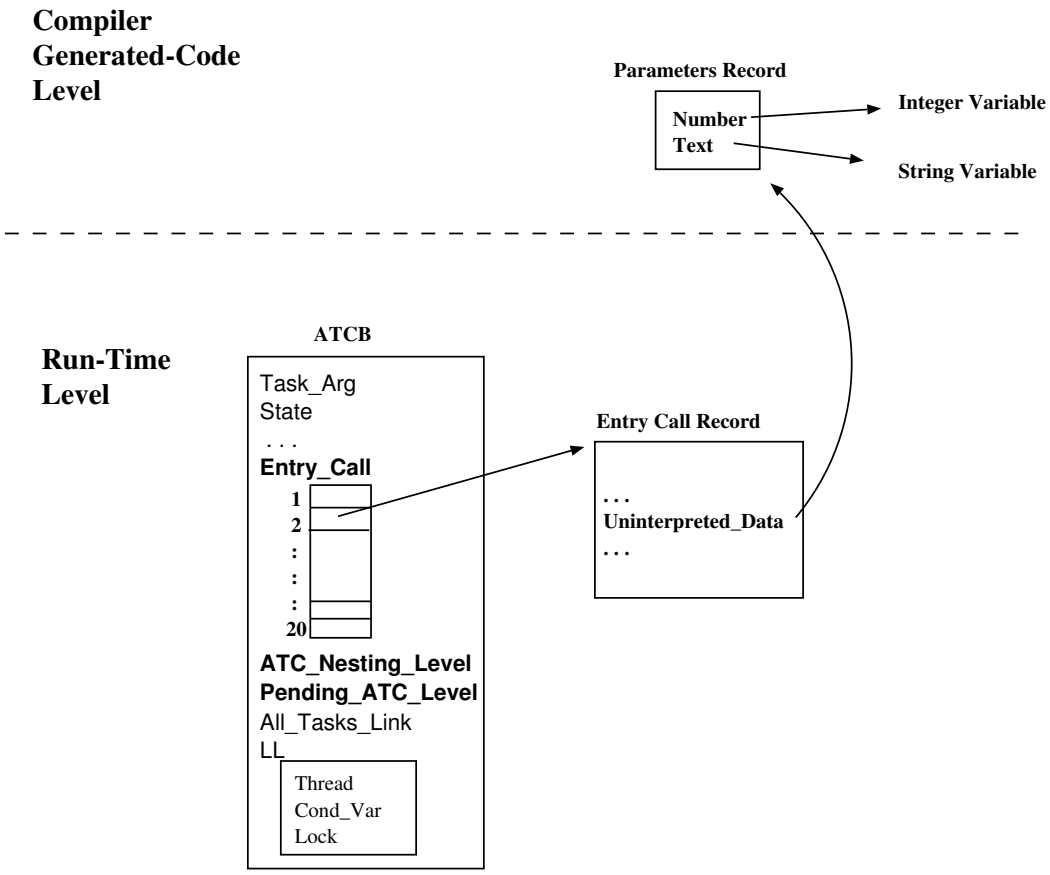

Figure 8.2: Entry Calls Stack.

8.2.4 GNAT Implementation of the One-Thread Model

Below we present the translation done by the compiler to implement the ATC.

```
declare
   P          : Parms := (Parm1, Parm2, ..., ParmN);
   Successful : Boolean;
begin
```

```
    GNARL.Defer_Abortion;
    GNARL.Task_Entry_Call (Task_ID, Entry_ID, P'Address, Successful);
    begin -- Abortable Part Scope
       begin
          GNARL.Undefer_Abortion;
          << Abortable Part >>
       at end
          GNARL.Entry_Call_Cancellation
       end;
    exception
       when Abort_Signal =>
          GNARL.Undefer_Abortion;
    end;
    if not Successful then
       [ Parm1 := P.Parm1; ]
       [ Parm2 := P.Parm2; ]
       [ ...               ]
       <<  Triggered Statements >>
    end if;
end;
```

The first action made in the scope associated with the ATC is to defer the abortion[7]. Without this, an abortion that occurs between the time that the call is made and the time that the abortable part's cleanup handler is set up might miss the cleanup handler and leave the call pending). The ATC request is also handled by the GNARL procedure *Task_Entry_Call* but in this case the whole sequence of actions is:

- **Task_Entry_Call:**[8]

  1. Increment the ATC nesting level.
  2. Elaborate one *Entry Call Record* and associate to it the *Entry Parameters Record*.
  3. Call *Task_Do_Or_Queue*.

- **Task_Do_Or_Queue**:

  (Exactly the same sequence of steps done for the simple mode entry call (described in section 3.2.2).

[7] *System.Taskin.Initialization.Defer_Abort*
[8] *System.Tasking.Rendezvous.Task_Entry_Call*

8.3 Summary

In this chapter the basic concepts of the GNAT implementation of the Ada local and global abortion have been presented.

- The GNARL implementation of the Ada abort statement is made up of:
  - One flag in the ATCB: *Aborting*. While set, this flag prevents a race between multiple aborters and the aborted task.
  - One internal exception: *_Abort_Signal*. This exception is not visible to user code and can only be handled by run-time system code.
  - One signal (SIGABRT), which can not be masked.
- There are two models to implement the ATC. GNAT implements the canonical one-thread model.
- Each task has one *Entry Call Stack* in its ATCB which is used to implement nested ATC entry calls.

Appendix A

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of

subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with

changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with

at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[AAR95] AARM. *Annoted Ada Reference Manual (Technical Corrigendum 1)*. ISO/IEC 8652:1995(E), 1995.

[BG93] T.P. Baker and E.W. Giering. *Gnu Low-Level Interface Definition*. Florida State University, 1993.

[BG94] T.P. Baker and E.W. Giering. *PART/GNARL Interface Definition*. Florida State University, 1994.

[BG01] T.P. Baker and E.W. Giering. *GNAT Reference Manual. Version 3.15p.* Ada Core Technologies, Inc., 2001.

[BR85] T.P. Baker and G.A. Riccardi. Ada tasking: from semantics to efficient implementation (PostScript / PDF). *Florida-State University*, 1985.

[BW98] A. Burns and A. Wellings. *Concurrency in Ada (2nd edition)*. Cambridge University Press, 1998.

[Coh96] N.H. Cohen. *Ada as a Second Language (2nd edition)*. McGraw-Hill, 1996.

[DIB94] O. Dong-Ik and T.P. Baker. *Optimization of Ada'95 Tasking Constructs (PostScript / PDF)*. Florida State University, 1994.

[DIBM96] O. Dong-Ik, T.P. Baker, and S.J. Moon. The GNARL Implementation of POSIX/Ada Signal Services. *Reliable Software Technologies. AdaEurope'96*, (LNCS 1088):275–286, June 1996.

[GB92] E.W. Giering and T.P. Baker. Using POSIX Threads to Implement Ada Tasking: Description of Work in Progress. *TRI-Ada'92 Proceedings*, pages 518–529, ACM, November 1992.

[GB94a] E.W. Giering and T.P. Baker. The Gnu Ada Runtime Library (GNARL): Design and implementation. *Wadas'94 Proceedings*, 1994.

[GB94b] E.W. Giering and T.P. Baker. Ada 9X Asynchronous Transfer of Control: Applications and Implementation. *Proceedings of the SIGPLAN Workshop on Language, Compiler, and Tool support for Real-Time Systems*, 1994.

[GB95] E.W. Giering and T.P. Baker. Implementing Ada Protected Objects. Interface Issues and Optimization. *TRI-Ada'95 Proceedings*, pages 134–143, ACM, Anaheim, California, 1995.

[GMB93] E.W. Giering, F. Mueller, and T.P. Baker. Implementing Ada 9X features using POSIX Threads: Design Issues. *TRI-Ada'93 Proceedings*, pages 214–228, ACM, Seatle, Washinton, September 1993.

[GMB94] E.W. Giering, F. Mueller, and T.P. Baker. Features of the Gnu Ada Runtime Library. Florida State University 1994.

[MGGM99] J. Miranda, F. Guerra, A. Gonzalez, and J Martin. *How to modify the GNAT Run-Time to Experiment with Ada Extensions*. University of Las Palmas de Gran Canaria, Canary Islands, Spain. ISBN: 84-87526-68-3. Available at http://www.iuma.ulpgc.es/users/gsd/Drago, 1999.

[SGC94] E. Schonberg, F. Gasperoni, and C. Comar. The GNAT Project: A GNU-Ada9X Compiler. New York University, 1994.

[Sta92] R.M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1992.

Index