Technical Report

CMU/SEI-87-TR-026
ESD-TR-87-189

Ada for Embedded Systems: Issues and Questions

Nelson H. Weiderman
Mark W. Borger
Andrea L. Cappellini
Susan A. Dart
Mark H. Klein
Stefan F. Landherr

December 1987

Technical Report
CMU/SEI-87-TR-26
ESD/TR-87-189
December 1987

Ada for Embedded Systems: Issues and Questions

Nelson H. Weiderman
Mark W. Borger
Andrea L. Cappellini
Susan A. Dart
Mark H. Klein
Stefan F. Landherr

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler SIGNATURE ON FILE
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Ada for Embedded Systems: Issues and Questions

Abstract: This report addresses issues and questions related to the use of Ada for embedded systems applications; it contains some preliminary recommendations for compilation system implementors, application developers, program managers, and Ada policy makers. The issues and questions provide the context for the Ada Embedded Systems Testbed (AEST) Project at the Software Engineering Institute, where staff members are investigating software development and performance issues for real-time embedded systems.

1. Introduction

1.1. Purpose

The purpose of this report is to provide a framework for articulating and investigating the issues and questions related to the use of Ada for embedded systems applications. These issues and questions provide the context for work on the Ada Embedded Systems Testbed (AEST) Project. The primary purpose of the project is to use a hardware and software testbed to investigate software development and performance issues for real-time embedded systems. The SEI testbed enables us to assess the readiness of the Ada language and Ada implementations for developing embedded systems. It also makes it possible for us to provide advice to contractors on how to address problem areas, advice to vendors on what tools and features need to be provided, and advice to the DoD on what is possible with currently available commercial products.

The contents of this report will help us develop the criteria for expanding the testbed and for performing evaluation work. As the issues, questions, and criteria become clear, we will implement the strategy for addressing a subset of the issues and questions in the context of the project. This report should not be considered a finished product, but rather a living document that will grow and become more refined as more issues and questions arise during the experimentation phase of the project.

This report deals with issues at a reasonably general level because to do otherwise would require several hundred pages. The categories of issues will be enumerated, and various examples will be given. We do not attempt to be exhaustive in enumerating every implementation option or runtime feature. Based on our research into the issues and questions, we make some preliminary recommendations for compilation system implementors, application developers, program managers, and Ada policy makers. These recommendations will be refined as we gain more experience with real application systems implemented on real hardware/software configurations.

The report is intended as a consolidation and synthesis of existing work for a rather high-level audience. We have presented the issues in a form understandable to managers and administrators as well as to highly technical personnel. The report will indicate to compiler vendors the general directions in which the technology should be going. References to previous work will be preferred over inclusion of the work in this report.

The purpose of this chapter is to provide the context for the remainder of the report. In the following section, we present the problems in general terms. We then briefly describe previous and ongoing work that explores related issues and questions. Finally, we discuss some broad questions to indicate the type of issues of interest.

Chapter 2 describes the embedded system problem domain. It is important to recognize that the real-time characteristics and resource constraints impose vastly different requirements on the Ada capabilities than do ordinary information system problems. It should also be noted that some of the issues identified in the report are not unique to embedded systems, but they are included for completeness.

Chapter 3 is concerned with Ada language issues from the point of view of the application developer. It deals with those features that specifically address embedded systems and also discusses the tradeoffs involved in using or not using these features.

Chapter 4 is concerned with the Ada language from the point of view of the compiler implementor. It focuses on the abstract machine which must be provided by the implementor in the form of runtime services. These services include task management, storage management, exception management, and interrupt handling. Because the language definition allows considerable latitude for the implementor, there are numerous choices that must be made.

Chapter 5 is concerned with the support tools for developing software in the embedded system domain. Because of the host-target configuration of this environment, there must be a variety of tools which do not exist when the target is the same machine as the host. The existence of these tools is critical to the software development process.

Chapters 3 through 5 make it clear that there are many choices available to the participants in Ada software development. The application developer chooses which Ada features to use. The implementor chooses which machine-dependent Ada features to include, how to implement certain difficult features, and what additional facilities to provide. The Ada program managers choose from various compilers and associated tool sets with widely differing characteristics. The Ada policy makers must decide which Ada tools are practical and when to mandate their use. Decisions must also be made regarding funding support for the development of advanced tools.

Chapter 6 summarizes the major issues and questions for each of these groups and makes some preliminary recommendations. It also outlines the next steps for the project.

An annotated bibliography is included as an appendix to this report. It covers papers concerned with the use of Ada in embedded systems, with particular emphasis on Ada runtime issues.

1.2. Background

In the last two years, Ada compilers have reached a state of maturity that justifies their use in production applications that are not time critical [Foreman 87]. However, Ada software development environments are lagging behind compilers in maturity and sophistication [Weiderman 87]. Many of them lack tools and interfaces we have come to expect in modern development environments, but they are still quite usable.

In contrast to Ada compiler and environment technology, the ability to use Ada in real-time embedded systems is just now being explored. Several vendors sell compilers which generate code for embedded systems targets such as the Motorola 68000 family, the Intel iAPX86 family, and the MIL-STD-1750A microprocessors, but the state of embedded system support tools is uncertain. Since the language was originally designed for embedded systems applications, it is critically important that the SEI gain expertise in this area and evaluate the readiness of the tools and techniques for developing and testing software for such applications. Furthermore, this expertise and the evaluation results must be disseminated to the broad community.

Embedded systems software is quite different from commercial data processing and most scientific data processing software. It is frequently written in assembly language by hardware engineers rather than software engineers. Software developers have to deal with odd hardware restrictions and constraints. The programs must be particularly efficient in order to respond to real-time inputs from a variety of sensors. The ability of Ada, and particular Ada implementations for target machines, to handle hardware dependencies must be demonstrated if Ada is to be employed successfully in embedded systems.

Using Ada for embedded systems involves significant changes for developers. Programmers once wrote systems entirely in assembly language and had complete control over every action and every byte of storage. More recently there has been a trend toward high-level languages, with real-time executives providing certain runtime services; but programmers are still accustomed to a high degree of control. Ada is meant to raise the level of abstraction further in order to permit the resulting product to be more cost-effective to produce, more maintainable over its lifetime, and more portable between different systems. As a result, the Ada runtime system (normally supplied by the compiler vendor) provides services previously supplied or controlled by the applications programmer. These services include scheduling, synchronization, timing, and memory management. Thus, the major issue is whether Ada technology can address the low-level efficiency and control problems of embedded systems applications and, at the same time, be high-level enough to solve the problems of modifiability, transportability, and cost-effectiveness.

1.3. Previous Work

A significant amount of research and development is being conducted with regard to using Ada for real-time embedded systems. This section gives a brief overview of some of the more important and visible efforts. Numerous references concerning this topic are included in an annotated bibliography appended to this report.

1.3.1. ARTEWG

The most active and visible group raising questions and issues regarding the use of Ada for embedded systems is the Ada Runtime Environment Working Group (ARTEWG). This group was established in the spring of 1985 by SIGAda, a special-interest group of the Association for Computing Machinery (ACM). The group was formed "to establish conventions, criteria, and guidelines for Ada runtime environments that facilitate the reusability and transportability of Ada program components, improve the performance of those components, and provide a framework which can be used to evaluate Ada runtime systems" [ARTEWG 86a]. The ARTEWG is a volunteer organization consisting of approximately thirty principal members, who have met quarterly since May 1985.

At the SIGAda meeting held in Charleston, West Virginia, in November 1986, the ARTEWG distributed 5 reports which are the culmination of 18 months of work. The first, "A White Paper on Ada Runtime Environment Research and Development" [ARTEWG 86a], provides some background and an overview of the work of the group. An updated version of this report has been published in *Ada Letters* as "A Framework for Describing Ada Runtime Environments" [ARTEWG 87a]. The second report, "A Canonical Model and Taxonomy of Ada Runtime Environments" [ARTEWG 86b], provides some of the basic definitions and concepts surrounding the Ada runtime model. The third is entitled "A Catalog of Interface Features and Options for the Ada Runtime Environment" [ARTEWG 86c]. Its objective is to "propose and describe the first of a common set of user-RTE interfaces, with which a programmer can both request services of the RTE and tailor the RTE to meet application specific requirements." The fourth report, "Catalog of Ada Runtime Implementation Dependencies" [ARTEWG 86d], provides a single source for those areas of the *Reference Manual for the Ada Programming Language* [LRM 83] which permit implementation flexibility. The fifth document is the "First Annual Survey of Mission Critical Application Requirements" [ARTEWG 86e]. The purpose of that report is to define the requirements of real-time embedded systems by using a common survey instrument to collect information from a variety of applications.

1.3.2. Ada-Europe

J. C. D. Nissen and B. A. Wichmann are the primary authors of a report entitled "Guidelines for Ada Compiler Specification and Selection" [Nissen 82], which provides a set of questions that can be used to evaluate Ada compilers and runtime environments. While this report is not aimed at the embedded system domain, the answers to the questions for any particular Ada implementation are important to application developers in assessing Ada tools.

1.3.3. AdaJUG

The Ada-JOVIAL Users Group (AdaJUG) is an outgrowth of the JOVIAL Users Group (which predates Ada). This group, which has had over thirty meetings, is particularly oriented to real-time embedded systems and their requirements. In two of their 1986 meetings [Graumann 86a, Graumann 86b], they held panel discussions of "Why Ada Insertion into Embedded Systems is Failing." These panels have been organized, not to denigrate Ada, but to raise the difficult issues and questions related to this particular application area. The panels have emphasized the lack of maturity and efficiency of the currently available Ada compilers and runtime systems.

1.3.4. Benchmark Test Suites

R. M. Clapp et al, of the University of Michigan and the Performance Issues Working Group (PIWG) of SIGAda have prepared and are distributing test suites of Ada programs. The tests have been executed on a number of systems, but there is little data for embedded system targets. The theoretical basis for timing and calibration of these tests is documented in the *Communications of the ACM* [Clapp 86]. The PIWG group has held workshops on performance issues [Squire 85, Squire 86]; the benchmark tests have been distributed to over 100 sites; and the group is coordinating the collection and reporting of the data being received. The MITRE Corporation also has developed a series of tests for an Ada runtime environment supporting JAMPS [Ruane 85].

1.3.5. The Evaluation and Validation Team

The Evaluation and Validation (E&V) team was established under the auspices of the Ada Joint Program Office (AJPO) in June 1983 [E&V Team 84a]. The goal of the E&V team is to develop the tools and techniques that will provide a detailed and organized approach to assessing Ada programming support environments (APSEs). Most notable of the work is the report on the requirements for APSEs and the prototype Ada Compiler Evaluation Capability (ACEC). They are in the process of funding a full ACEC which will address many of the questions raised when Ada is used for embedded systems. Among their more relevant reports are the "Requirements for Evaluation and Validation of Ada Programming Support Environments" [E&V Team 84b] and the "E&V Classification Schema Report" [TASC 86].

1.3.6. SofTech

A series of technical reports written by V. Grover and N. Lomuto [Grover 83, Grover 85a, Grover 85b, Lomuto 82, Lomuto 83] have provided significant insights into the issues and questions described in this report. Those reports deal with the Ada runtime in general, options for Ada implementations, possibilities for a user-tailorable runtime "kit", and the design of real-time systems in Ada.

1.4. Broad Issues and Questions

To provide a foundation for the more detailed issues raised in subsequent chapters and to set the stage for these chapters, we will itemize here some of the most important issues and questions related to the use of Ada in embedded systems.

- Are the Ada implementations for tasking, exception handling, and interrupt handling fast enough to be used in time-critical applications?
- Are Ada runtime environments small enough to reside with embedded applications on typical military processors?
- Are the Ada timing mechanisms sufficiently precise for periodic scheduling of embedded system activities?
- Can the Ada runtime environments be tailored by the user or compiler vendor so that the user does not have to pay time and space performance penalties for features of the language that are not used?
- Is it practical and desirable to provide highly optimized runtime support primitives (e.g., Ada packages containing semaphores and high-precision timing) to supplement or replace predefined language constructs?
- What is the current compile-time and runtime performance of Ada cross-compilers? What is the quality of the generated code?

- What is the current state of the tools for supporting development and testing of embedded systems? Are there intelligent linkers? Can code be put into read-only memory (ROM)?
- Should embedded systems programmers be willing to give up some of the control of the runtime environment in order to obtain the benefits of the software engineering principles gained with Ada?
- Is there a danger that the proliferation of Ada runtime environments and Ada tool sets will have as deleterious an effect on portability and reusability as the proliferation of languages had in the 1970s?

A fundamental issue is the ability of Ada and Ada-based software engineering tools to satisfy the conflicting requirements of generality (the ability to solve problems in a wide range of applications and over a wide range of embedded systems) and efficiency (the ability to use limited resources in an expedient manner). It is this challenge that must be addressed by all participants in the embedded system community.

2. The Embedded Computing Systems Problem Domain

The purpose of this chapter is to describe a set of computing problems that are characteristic of software in the real-time embedded systems problem domain. Some of these problems are not unique to this domain, but they are important nevertheless. We will also identify the tools of a development environment that are required specifically for this problem domain. This will set the stage for analyzing Ada's viability in this arena.

The goal of this chapter is not to provide an exhaustive coverage of the entire embedded systems domain. Rather, it will focus on those embedded systems in which constraints on computing resources are important factors. Section 2.1 characterizes the embedded systems problem domain. Section 2.2 describes a set of computing problems associated with this problem domain. Section 2.3 briefly enumerates developmental requirements and tools that are unique to this problem domain.

2.1. Characteristics of Embedded Computing Systems

2.1.1. Overview of the Embedded Systems Problem Domain

The parent domain is the realm of complex, heterogeneous, industrial and military systems, such as process-control systems, robotic systems, avionic systems, weapon systems, and C^3I systems. These systems consist of subsystems that interact and operate in parallel. Historically, these subsystems were manual, mechanical, electro-mechanical, or hydraulic. Electronic digital processors (i.e., computers) were introduced into these systems because of their greater flexibility, potentially greater numerical accuracy, and potentially greater computing power.

The digital processing subsystem is one of the subsystems embedded in the larger system—hence, it is referred to as the embedded computer system (ECS)—and it must interact with and operate in parallel with the other subsystems. The other subsystems may be sensors, actuators, displays, or recording devices. The ECS may be required to interface with these other subsystems in a variety of ways, such as through A/D or D/A converters or by direct electronic signals. This type of complex interaction is characteristic of the mission-critical computer resources (MCCR) often found in command and control or weapon systems.

A system containing an ECS generally has greater functionality and versatility because of the processing power and the programmable nature of the digital processor(s). However, there are difficulties in inserting computers into real-world systems and developing the software for them, as evidenced by the all too many projects that are late, over budget, only partially successful, or outright failures—the so-called "software crisis."

There are many reasons why developing software for an ECS is so difficult. One is the phenomenon of "creeping expectations"; more and more system functions, especially the difficult and poorly understood ones, are allocated to the ECS. Another, more fundamental reason is that concurrency is natural for "physical" subsystems but is a mismatch with von Neumann style digital processors (i.e., the vast majority of current computers). Another reason is the lack of adequate tools to handle the sheer complexity of specifying, designing, coding, and testing the software components of an ECS.

2.1.2. Definition of Embedded Computer Systems

The first step in characterizing embedded computer systems is to formulate a clear definition. Because of the complexity of the topic, many definitions of an ECS have been proposed. Two of these are listed below:

- An ECS is "a component of a larger system and provides computation, communication, and control functions for that system. ECSs are often implemented using microprocessors and often incorporate concurrency and interrupt-driven real-time processing" [Fairley 85, p. 229].
- An ECS is "a computer system that is integral to a larger system whose primary purpose is not computational; for example, a computer system in a weapon, aircraft, command and control, or rapid transit system" [IEEE 83, p. 18].

The following definition has been adopted for the purposes of this report:

> An embedded computer system (ECS), consisting of microprocessors and resident software, is an integral component of a larger system whose primary purpose is not necessarily computational. The ECS provides real-time computation, communication, and control functions for the encompassing system.

2.1.3. Characterization of Embedded Computer Systems

Although there is great variability in embedded computer systems, it is possible to extract a number of significant characteristics.

ECSs have several characteristics in common with most other computer systems. Systems must be:

- Functional: The ECS must perform all the required functions.
- Correct: The ECS must produce correct results from valid inputs.
- Robust: The ECS must handle imperfect inputs in a reasonable manner.
- Secure: Often the ECS must provide some security features (e.g., a C^3I system).
- Economical: The ECS must be economical to design, implement, maintain, and use.
- Adaptable: The ECS must be adaptable to corrections, improvements, and modifications to the operating environment.

There are a number of characteristics that distinguish an ECS (especially the small-to-medium ECS emphasized in this report) from other types of computer systems. In any particular ECS, each of the characteristics listed below is present to some degree.

- Real-Time Operation: The ECS is nearly always required to interact with the overall system in real time. "Real-time software works in a time-critical environment to control some system of devices" [Allworth 81, p 3]. "Real-time software measures, analyzes and controls real-world events as they occur" [Pressman 82, p 14]. Real-time may imply "hard" timing requirements, where any missed deadline will result in system failure, or "soft" timing requirements, where a missed deadline need not result in system failure.
- Physical Constraints: Often there are severe size, weight, and power limitations. In addition, tolerance to environmental extremes may be required (e.g., shock, vibration, radiation, humidity, and temperature).
- Complex Interfaces: The ECS is usually interfaced to a heterogeneous set of devices, some of which may be unique to the particular application. Furthermore, both inputs and outputs may be periodic, randomly distributed in time, or may occur in sporadic bursts.
- Criticality: Often the ECS is part of a critical system, in which a failure could have

serious safety or economic consequences. The system, and hence the ECS, is often required to operate nonstop for extended periods. The ECS frequently endows the entire system with the nature of its behavior and thus is a critical component whose integrity affects the integrity of the larger system.

- Parallel Development: Often ECS software and system hardware are developed simultaneously.
- Novelty: Often the entire system development (and especially the ECS development) is a first-of-a-kind attempt to grapple with a poorly understood or poorly specified problem.

2.2. Computing Problems Associated with Embedded Computer Systems

This section presents a summary of the computing problems associated with embedded computer systems software. If Ada is to be considered a suitable language for constructing ECS software, it must provide facilities to appropriately handle the problems discussed below. A discussion of Ada's suitability with respect to this set of computing problems will be presented in Chapters 3 and 4.

Our working definition of an ECS leads to the generic requirement that one or more microprocessors with limited processing power and memory must simultaneously sense, process, and control multiple environmental parameters in real time. A number of computing problems are generated by this generic requirement. Simultaneous control of multiple parameters introduces the problem of concurrency. Concurrent processing can be simulated on a uniprocessor by appropriate scheduling of CPU usage. Concurrent processing may also be implemented by a communicating set of dedicated distributed processors. Control of environmental parameters with limited processing power, memory, and input/output capabilities introduces the general problem of resource allocation. I/O interactions tend to occur with a set of heterogeneous devices; and the overall complexity of the application will, in general, require mechanisms for abstractly modeling the environment.

The computing problems associated with embedded systems have been divided into the following categories: multiprocessing, distributed processing, input/output, and modeling the problem domain. Discussion of these areas is followed by a discussion of the constraints imposed on real-time systems.

2.2.1. Multiprocessing

For multiple activities that proceed in parallel, it is natural to use a model that employs parallelism. Multiprocessing allows one to logically view independent threads of execution as proceeding in parallel. However, the parallel model gives rise to a set of computing problems. Multiple processes functioning in a coordinated fashion require interprocess communication. Multiple processes will also vie for limited resources. We have divided the multiprocessing problems into three major categories: problems relating to interprocess communication, problems relating to scheduling the CPU, and problems relating to resource allocation.

1. **Interprocess Communication**
    - Controlled Data Sharing: Interacting processes may need a mechanism for accessing common data. Maintaining the integrity of this shared data requires mutual exclusion facilities.

- Process Synchronization: Interacting parallel processes often have points where processing must be synchronized between two or more processes. A mechanism must be provided for one process to wait on an event which may be signaled by another process and for a process to signal another that an awaited event has occurred.

2. **Process Scheduling**
   - Process State Management: All scheduling disciplines require management of the state of all existing processes. In general, a process can be *executing*, *ready to execute*, or *waiting* for a resource or event to occur before it can become eligible (or ready) to execute. If an executing process needs a resource (for example, if it is waiting for I/O), the processor should be used by another ready process.
   - Interrupts: Both time-slicing and event-based scheduling require that the CPU be interrupted. Interrupts must be controllable to prevent interruption of a critical section.
   - Real-Time Clocks: Since all time-slicing mechanisms require a cognizance of time, access to a clock is necessary.
   - Process Priorities: Preemptive, priority-based scheduling requires that processes be assigned priorities. It is conceivable that these priorities may change as a function of time and state, requiring dynamic assigning of priorities.

3. **Resource Allocation**
   - CPU Allocation: Care must be taken not to impinge upon the real-time requirements of the system. For example, if critical sections are implemented, they must be short enough so that they do not unnecessarily delay other processes. In general, usage of the CPU by the runtime system must be predictable. The multiprocessing overhead should be independent of the number of processes. (See also Process Scheduling.)
   - Memory Allocation: Both static (at compile/link time) and dynamic (at runtime) allocation of memory may be required. Dynamic allocation must be efficient in time and space. Memory deallocation (garbage collection) must be efficient, controllable, and predictable.
   - Device Allocation: Like memory allocation, device allocation may be static (at system build time) or dynamic. Because devices must be available when they are needed, they must be allocated ahead of time.

2.2.2. Distributed Processing

The previous section considered multiprocessing from a logical point of view, independent of a particular physical implementation. When implementing multiprocessing in a distributed environment, several additional problems not previously mentioned must be considered. Although the CPU is less of a limited resource in this environment, coordination of multiple processes now involves the added complication of communication between processors. The distributed processors may exhibit various degrees of coupling. Loosely coupled processors may act relatively independently, communicating only by messages passing over a network. Alternatively, they may be more closely coupled, sharing memory and I/O buses. For each of the major categories previously listed, we describe below the peculiarities introduced in a distributed environment.

- Interprocess Communication: In a distributed environment, the extent of processor coupling and the potential heterogeneity of multiple processors become issues. In a closely coupled system, data may be shared through common shared memory. In a loosely coupled system, data sharing may require message passing. These issues affect common data access and process synchronization.

- Process Scheduling: Managing processes across a distributed network may require more sophistication than a process table commonly used in a uniprocessor configuration. In addition, for the entire distributed system to have the same notion of time, either dedicated clocks must be synchronized or a central clock must be accessible.
- Resource Allocation: In a closely coupled system, memory may be shared, requiring processor coordination for memory allocation. Timely interprocessor communication becomes an important factor in coordinating device allocation in a distributed environment.

2.2.3. Input/Output

Embedded computer systems tend to be input/output intensive; they are often interfaced with many heterogeneous devices and are often required to handle different types of I/O . Also, each class of processor has its own peculiarities regarding I/O.

- Low-Level Input/Output: I/O ports and registers must be directly addressable in order to interface to specialized (custom) devices. In addition, it must be possible to write interrupt handlers that appropriately handle nondeterministic events.
- Data Representation: Data from special-purpose devices may be encoded in special ways. Mechanisms to convert the data to other representations or to provide access at the bit and byte level is imperative.

2.2.4. Modeling the Problem Domain

As the complexity of problems increases, the ability to model a problem in its own terms and to abstract problem details becomes increasingly important.

- Numeric Computation: An ECS must be able to represent real-world entities and quantities and to perform related manipulations and computations. Thus, there should be support for numerical computation, units of measure (including time), and calculations and formulae from physics, chemistry, etc.
- Cognizance of Time: Process scheduling and similar runtime activities require precise measurement of time intervals. In addition, nearly all ECSs require an accurate time-of-day clock for proper interaction with the outside world.

2.2.5. Operational Constraints

It should be recognized that the problems summarized above have associated constraints; the principal one is time criticality, which is compounded by limited processing power and memory constraints.

- Time Criticality: An ECS must provide specified amounts of computation within required time intervals. The consequences of missing a real-time deadline can vary from reduction of throughput to numerical inaccuracy to partial loss of system functionality, or even to total system collapse. Therefore, the time taken to perform system functions such as process initiation, process termination, and context switching is crucial in a real-time multiprocessing system. System start-up time is also important, as is the time taken to change operating modes, to reconfigure the system after a partial failure, or to restart the system after a total failure.
- Reliability: An embedded system may have to operate nonstop for an extended period of time. In applications such as NASA's space station, the software must remain operational even during installation of revisions. Predictable exception handling for expected

errors (e.g., invalid inputs) is required. Tolerance for unexpected errors (e.g., hardware failures) is also desirable.

- Limited Resources: Apart from processor speed, memory and I/O capabilities may be limited. Limited memory must be used economically. Operating systems support will be minimal; typically only a kernel of runtime services can be accommodated.

2.3. The Embedded Computer Systems Development Environment

The development of ECS software has many aspects in common with the development of software for other types of systems. The programming language and the development environment must support the design, implementation, and testing of a complex software product, often for a poorly understood or poorly specified problem. Therefore, the language and environment should provide support for abstract data types and the ability to develop software in stages through separate compilation, incremental build and test, and rapid prototyping.

However, software development for an ECS also presents a set of problems that place unique requirements on a supporting development environment. These problems are primary ramifications of the fact that the target execution environment is different from the host development environment. Some of the tools required for an ECS development environment are described in Chapter 5.

2.3.1. Unique Development Requirements

- Host-Target Environment: A small-to-medium ECS, such as the ones considered in this report, usually executes on dedicated processor(s), with minimal or no operating system support and with little or no disk storage. Such computers are *not* suited to software development. Therefore, ECS software development requires hardware and software tools not often needed when constructing other types of software. The basis of this work is a host-target development environment in which the majority of the software tools reside on the host computer.
- Target Simulation on Host: Sometimes the target computer itself is developed in parallel with the ECS software to allow meaningful software testing, It is necessary to have a simulation of the target computer available on the host computer.
- Non-Invasive Testing: The usual technique for monitoring, measuring, and testing computer software is the use of a symbolic debugger; but this approach affects the time and space behavior of the program that is under examination. Such perturbations can usually be tolerated when subsets of the program are being developed, but they cannot be tolerated in certain time-critical areas of the program, nor when the complete program is subjected to a full-load test. In these cases, independent hardware instrumentation facilities, such as logic analyzers and in-circuit emulators, are commonly used.
- Custom Testbed: Development and testing of ECS software requires a testbed incorporating actual devices as well as extensive instrumentation facilities.
- Testing and Verification: In general, testing and verification of ECS software is more difficult than that for other types of software:
    - Anticipating all "real life" scenarios is difficult.
    - Setting up the testbed and using it to simulate "real-life" scenarios can be difficult.
    - Non-invasive testing is difficult, but it is necessary for credible results.
    - Error manifestations are often difficult, or impossible, to reproduce.

2.3.2. Embedded Systems Tool Kit

As with any software, the development of ECS software is facilitated by good tools for specifying complex problems, designing complex software, and managing large software projects. Developers need a high-level language in order to cope with the complexities of the application and a compiler that produces efficient machine code. In addition, the development of ECS software requires some specialized tools, such as the following:

- cross-compiler and cross-assembler, executing on the host
- specialized linker and downloader
- real-time executive or runtime kernel
- symbolic remote debugger, executing on host and controlling target machine
- performance analyzer
- sensor simulators

These tools will be discussed in more detail in Chapter 5.

2.4. Summary

The embedded systems problem domain involves a characteristic set of problems. These problems are both computational and developmental. The computing problems are typically addressed by special real-time operating systems or runtime systems of a language. The development problems are addressed by appropriate features of the programming language or by the special tools of the programming environment. The general problems discussed in this chapter will be analyzed in the context of the Ada language in the following chapters.

3. Ada in Embedded Systems

The purpose of this chapter is to show how some features of Ada specifically address the real-time embedded system problem domain from the point of view of the application developer. The chapter is organized into two sections. First, we discuss Ada real-time features, as defined in ANSI/MIL-STD-1815A [LRM] that are relevant to problems in the embedded system domain characterized in Chapter 2. Second, we discuss the issues, questions, and programming alternatives (including the corresponding tradeoffs) relative to using Ada's real-time features in an embedded system.

3.1. Ada Real-Time Features

The Ada language was designed as a common high-order language for programming large-scale and real-time systems. In keeping with the theme of investigating the issues related to the use of Ada for embedded systems applications, this section discusses, from a real-time application developer's viewpoint, specifically how the Ada language addresses the embedded system problem domain. Emphasis is on its real-time features (e.g., tasking, exceptions) rather than on its modern software engineering features (e.g., packages, generics). Paramount in this discussion are the following issues:

- concurrent control of system components (e.g., multitasking, synchronization)
- time control (e.g., time deadlines, accurate time measurement)
- input/output (e.g., interrupt handling, polling)
- control over internal representations of data, error handling (e.g., error detection and recovery)
- numerical computation

3.1.1. Concurrent Control

As discussed in Chapter 2, one of the requirements for embedded software is parallelism. Ada provides a mechanism called a *task* that supports this capability [LRM, Chapter 9]. A task is one of Ada's four primary program units, the others being subprograms, packages, and generic units. An Ada task is similar in form to a package; both are comprised of two textual parts: a specification that describes its external appearance (i.e., callable interfaces) and an executable body which defines its internal behavior. The major difference between these two units is that a package is a passive construct that provides visibility control, whereas a task is an active construct that provides the capability of parallelism. Ada tasks can be created statically. Moreover, since a task specification declares a task type, tasks are objects that can be created dynamically at runtime and can also be components of other data objects such as records and arrays.

With multiple tasks executing independently, the software must provide a facility for communication and synchronization. Task communication in Ada is called *rendezvous*. Two tasks execute independently until the time when they must synchronize (e.g., rendezvous) and possibly exchange data. Once the rendezvous is complete, the tasks continue their independent execution. For a rendezvous to take place, one task *calls* another task. The client task, say Task A, calls an *entry*, declared in the specification of the server task, say Task B. For every entry declared, one or more associated *accept* statements will be given in the body of Task B. Each accept statement has an optional body whose code is executed during the rendezvous. This sequence of statements is delimited by the reserved words *do* and *end*. The rendezvous is executed at the higher of the priorities of the client and server tasks.

Since real-time software typically must handle an unpredictable sequence of operations, decisions about task rendezvous may have to be made at runtime. The construct in Ada that provides this option is the *select* statement, different forms of which are used by clients and servers. There are three kinds of select statements: selective wait, conditional entry call, and timed entry call (discussed in Section 3.1.2). The selective wait involves selecting one *accept* from several alternatives. When a server task reaches a selective wait statement, it may rendezvous with any of the possible entries in a nondeterministic manner, i.e., there is no language-defined rule as to which rendezvous will be chosen. The selective wait guarantees that one of the immediately possible rendezvous will be performed if there are any. A variation of the selective wait statement allows the programmer to control which rendezvous will be eligible for selection. A *guard* (Boolean expression) is placed before the accept, and only when the guard is true will the corresponding rendezvous be considered eligible.

The conditional entry call, which involves only one entry call, has two branches. The first branch is an entry call; the second is an *else* alternative, which is just a sequence of statements. When a client task reaches a conditional entry call, the first branch is chosen if the rendezvous can be performed immediately; otherwise, the second branch (else alternative) is chosen and executed.

Task termination in Ada can be complex. One must be aware of the parent-child task dependencies and other situations that influence the termination of a task (Chapter 4). Two mechanisms for termination are provided: the *abort* statement, which permits a task to be terminated from any point in the program where the task is visible; and the *terminate* alternative in a select statement, which will be executed if no other alternatives are possible and if conditions warrant task termination. The detailed issues surrounding Ada task termination are well documented [Barnes 84, Borger 86, SofTech 84] and will not be covered here.

In some real-time applications, it may be necessary to know how much storage a task object is allocated or what state a task is in. Ada has special operations, called *attributes*, which can provide that information [LRM, Annex A]. The attributes CALLABLE and TERMINATED provide information about the state of a task. The COUNT attribute yields the number of entry calls waiting for a particular entry. The attributes SIZE and STORAGE_SIZE provide information about storage assignments for task objects and types. These attributes can also be used in *length clauses* (discussed in Section 3.1.4) to specify an exact size (amount of storage) to be associated with a task type. Finally, the ADDRESS attribute yields the first machine code address associated with the body of a particular task type.

Ada provides mechanisms for ranked tasks and shared variables. Real-time systems typically have several functions that need to be performed simultaneously. The computer resources may be such that it is not possible to handle every function at a given time. With this in mind, Ada provides a predefined *pragma* (i.e., compiler directive), namely PRIORITY, for associating a priority with a task [LRM, Annex B]. Another predefined pragma that may be helpful for the applications programmer is SHARED. With multiple tasks executing, there may be an instance where the same nonlocal variable must be accessed. Pragma SHARED is the mechanism which designates that a variable is shared by two or more tasks. It should be noted that Ada implementations are required to recognize language-defined pragmas and may optionally define a set of implementation-specific pragmas; however, in either case, an implementation is allowed to ignore any pragma that it does not support. In other words, an implementation will not *necessarily* perform any action on these pragmas. An application programmer must be aware of what pragmas an Ada implementation *truly* supports.

3.1.2. Time Control

Strict timing demands must be satisfied by a real-time system. The system is required to respond to external stimuli and generate results within a strict, and sometimes fixed, deadline. Ada provides a *delay* statement to aid meeting time constraints on the execution of tasks (LRM, Section 9.6). The delay statement suspends execution of a task containing the statement for a minimum specified duration. The language guarantees the lower bound for the length of the delay but imposes no requirement on the upper bound. The time given in the delay statement is expressed in fractions of seconds and is a fixed point type defined in the predefined package CALENDAR. The delta of this fixed point type, DURATION, is implementation dependent and is defined by the value of the attribute SMALL. The language requires that DURATION'SMALL be less than 20 milliseconds and recommends a value not greater than 50 microseconds. One type of select statement, *timed entry call*, makes use of the delay statement. In a timed entry call, a client task will rendezvous if it can be started within the delay time specified; otherwise, a set of statements in the delayed alternative will be performed.

Ada offers three pragmas for reducing execution time. Pragma INLINE requests that calls to a specified subprogram be replaced inline by the body of that subprogram, thus eliminating the execution overhead associated with calling the subprogram, passing parameters, and returning from the subprogram. Pragma SUPPRESS allows a compiler to omit certain runtime checks (e.g., Range_check, Storage_check) in a program as a tradeoff between automatic runtime error detection and execution efficiency. Pragma OPTIMIZE with the parameter TIME allows a compiler to choose time over space as the primary optimization criterion.

3.1.3. Input/Output

A common characteristic of embedded software systems is a strong dependence on real-time input and output. Real-time input/output has unique properties in that it tends to occur at the hardware device level, can be subject to strict timing requirements, and can be either synchronous or asynchronous. Handling I/O for a specialized hardware device requires a special interface which has to provide all the capabilities typically found in device drivers and interrupt handlers. For example, a real-time application needs to enable, disable, and handle device interrupts; it may need to send control signals to and request status from a device; finally, it probably will have to move data to and from the data register(s) or I/O memory of a device. Independent of an embedded system's I/O device configuration (e.g., bus I/O, direct memory access, memory mapped I/O) and the nature of the I/O operations (i.e., synchronous or asynchronous), two techniques for handling real-time I/O operations are common: interrupt handling and polling. This section addresses the features Ada provides in support of these two approaches.

Typically, low-level asynchronous I/O operations to and from hardware devices tend to be interrupt driven. As such, a real-time application developer needs low-level I/O support and the ability to handle hardware interrupts efficiently in software. Ada provides mechanisms to handle both of these real-time I/O requirements. A low-level I/O package is provided in Ada's predefined language environment, namely LOW_LEVEL_IO (LRM, Section 14.6). This package provides control primitives, namely the SEND_CONTROL and RECEIVE_CONTROL procedures, for I/O operations on a physical device; however, as one might expect, the details of their parameters (i.e., device kind and data packet format) are implementation defined. For handling hardware interrupts in the application software, Ada supports task interrupt entries by allowing bindings, via address clauses, between a task entry and a hardware device that may cause an interrupt. Note that address clauses can also be used to specify the address of data objects or the starting address of the machine code associated with subprograms, tasks, or packages. In the case of interrupt entries, an interrupt acts as an entry call made from a conceptual hardware task whose priority is higher than the main program and any other user-defined task.

In situations where an application's computing behavior relies heavily on incoming data, it is common practice to poll the input device for the current data values. Particular polling techniques can be any combination of blocking or nonblocking and periodic or aperiodic. The distinction between blocking and nonblocking I/O operations is that a blocking I/O request will be suspended until it is satisfied; however, for a nonblocking operation, control is returned to the caller without suspension if the request cannot be satisfied immediately (or within a specified time-out period). A blocking I/O operation is analogous to a normal Ada task entry call, whereas a nonblocking I/O request coincides with a timed or conditional entry call. Periodic and aperiodic polling techniques can be distinguished by their respective temporal behavior: periodic polling has a deterministic behavior since the I/O requests occur at regular time intervals (e.g., every 10 milliseconds), whereas aperiodic polling is nondeterministic. Any of these polling mechanisms can be implemented in Ada using the delay statement and a general-purpose loop construct.

3.1.4. Internal Representation

As mentioned in Section 3.1.3, embedded software systems tend to need low-level interfaces to hardware devices; furthermore, because of the hardware-dependent nature of these systems, efficient data representation in terms of the underlying machine's architecture is required. Typically, these low-level interfaces consist of tightly packed data structures, dedicated memory locations, and special-purpose registers. Unlike most other high-order languages, Ada provides representation clauses [LRM, Section 13.1] which allow one to specify how the data types of an application are to be mapped onto the underlying machine architecture. Representation clauses can take one of two forms, either as a type representation clause or as an address clause. Since address clauses were discussed in Section 3.1.3, this section will focus on type representation clauses.

Ada's type representation clauses fall into three categories: length clauses, enumeration representation clauses, and record representation clauses.

A length clause allows one to specify four different size requirements: the size, in bits, to be allocated for an object of a particular type; the collection size in machine-dependent storage units (e.g., bytes, LRM 13.7.1) for an access type; the amount of space, in storage units, to be allocated for activating a task object of a particular task type; and the value of the actual delta for a fixed point type. For instance, when an application needs strong control over dynamic storage allocation, the length clause can be used to limit the total amount of storage available for the collection of objects of a given access type. Allocation can be removed either automatically by an Ada implementation or explicitly by the application using the predefined generic library subprogram UNCHECKED_DEALLOCATION.

An enumeration representation clause specifies internal codes that are to be used to represent the enumeration literals defined for any enumeration type. This is useful, for instance, when mapping integer error codes onto enumeration literals.

A record representation clause specifies the storage layout of a record type, including the alignment, order, storage place, storage unit positions within the storage place, and the size of the components. The record representation clauses and length clauses allow for the specification of tightly packed data structures for interfacing with the underlying hardware. Ada also defines an optional representation pragma named PACK for influencing a compiler's mapping of entities (either arrays or records) onto the underlying machine. Pragma PACK indicates to the compiler that minimizing storage is the main criteria for selecting the underlying representation of objects of either an array or record type.

3.1.5. Error Handling

Real-time embedded software systems must be reliable, where reliability is typically measured in terms of the system's availability, the mean time between failures, the mean time to repair, and the frequency of failure [Allworth 81, Hood 86]. The normal approach developers have taken in order to meet reliability requirements is to design the real-time system in such a manner that it can recover from its faults (i.e., they make it fault tolerant). To this end, real-time software must be able to both detect and subsequently recover from errors. The Ada language provides a mechanism for dealing with errors or other exceptional situations during program execution, namely the *exception* [LRM, Chapter 11].

Ada exceptions come in two varieties: predefined and user-defined. The former are predefined in the language (e.g., CONSTRAINT_ERROR, STORAGE_ERROR) and are automatically raised during program execution when the exceptional situations with which they are associated are detected. The latter are defined by the real-time application designer for exceptional situations unique to the application. In the case of user-defined exceptions, the burden is on the developer to supply code which detects exceptional situations and raises the corresponding user-defined exception.

For designing fault-tolerant software systems, the use (declaring, detecting, raising, and handling) of user-defined exceptions applies well. The real-time application designer/developer can plan for possible exceptional execution states by declaring corresponding user-defined exceptions. When abnormalities are detected by the software, the appropriate exception can then be raised and subsequently handled by the error recovery (exception handling) code. This approach offers a viable alternative to the more traditional technique of returning a status code from every subprogram invocation. It is attractive from the real-time application developer's point of view, since the tedious and time-consuming checks for the value of the returned status code can be eliminated from the program's source code and replaced by an automatically generated and optimized control structure.

3.1.6. Numerical Computation

For the representation and implementation of physical quantities, Ada provides two classes of real data types: fixed point and floating point [LRM, Subsection 3.5.6]. Intuitively, *fixed point* means a fixed number of places before the decimal point and a fixed number after; *floating point* means there is a fixed number of significant digits and an exponent. Since real data types are only approximations, internal representation of objects will be inaccurate in either case. Floating point values have a roughly constant relative error, whereas fixed point quantities have a constant maximum absolute error. For both classes of real data types, Ada defines two machine-dependent attributes: MACHINE_ROUNDS determines if rounding is performed for predefined arithmetic operations on values of a specified type, and MACHINE_OVERFLOWS specifies whether the exception NUMERIC_ERROR is raised in overflow situations. Specifically for floating point data types, Ada defines several other attributes (i.e., MACHINE_RADIX, MACHINE_MANTISSA) that provide characteristics of the underlying machine representation. Finally, in real-time systems, computations of mixed numeric types may be necessary. Since Ada is a strongly typed language, explicit type conversions are available between any two numeric types.

3.2. Issues and Questions

Program managers, despite a DoD directive, are faced with the decision of whether or not to use Ada. Factors such as a steep learning curve, maturity of Ada compilers (i.e., runtime performance, implemented options), and the availability of complete programming environments all influence this decision. From the real-time application developer’s viewpoint, the decision to adopt Ada largely depends on whether those Ada features discussed in Section 3.1 can be used. In this section, we discuss the ramifications of using those Ada real-time features and, where appropriate, alternatives to their use along with resulting tradeoffs. This section also includes a high-level discussion of possible programming idioms involving the more broadly based language features (e.g., tasking, exceptions).

3.2.1. Concurrent Control

Section 3.1.1 introduced the Ada tasking mechanism, which provides the real-time application programmer with a facility to do multitasking. The decision to use Ada multitasking depends mainly on the scheduling requirements of the application. In MacLaren [MacLaren 80], real-time applications are classified by their inherent scheduling complexity as follows:

- Level 1 consists of the purely cyclic (periodic) applications. The schedules are rigid and invariant since no asynchronous events will occur.
- Level 2 applications are mostly cyclic with some asynchronous events and possible variations in computing loads.
- Level 3 applications are event driven and contain little or no periodic processing.

Common practice has been to employ a cyclic executive for all three levels, but MacLaren shows that Level 2 and 3 applications can be approached using Ada multitasking. The benefits of Ada multitasking can be realized for applications at these levels. (Multitasking supports asynchronous events, monitors intertask dependencies, controls task interaction, and supports cyclic processing at arbitrary frequencies.) Level 1 applications are currently better approached by implementing a cyclic executive. With Ada multitasking, the runtime system is responsible for scheduling tasks. With a cyclic executive, the application programmer controls the scheduling. Cyclic executives can be written in Ada, but at the cost of obscuring the underlying structure of the problem. The outstanding issue is whether the constraints imposed on the scheduler by the language prevent a predictable, high-performance implementation capable of handling Level 1 applications. Further research and experimentation is required to resolve this issue.

When Ada multitasking is used with a runtime system scheduler, the issue of portability must be considered. Different implementations will undoubtedly use different scheduling algorithms, which will most likely affect the order of task execution. The application programmer must be aware of this ramification when employing different implementations.

For Ada tasks, the rendezvous facility is used for synchronization and communication (i.e., data passing). During a rendezvous, the client task is suspended until the server task completes its accept statement. To keep this suspension to a minimum, the accept statement (with associated body) should be kept as simple and small as possible. This still may be too restricting, which would make the application programmer consider the alternatives. Without rendezvous, tasks would execute totally independent of each other. This implies that all common data would be shared (possibly with pragma SHARED), causing the typical shared variable problems such as simultaneous updates, consistency, and correctness. Tradeoffs associated with these alternatives are the degree of program complexity versus the degree of performance and runtime overhead (storage and execution time) incurred by the rendezvous.

For the termination of tasks, the abort statement was discussed in Section 3.1.1. The results of an abort statement depend on several factors, including the state of the task to be aborted when the abort statement is reached (if more than one task is named, the order of abortion is undefined). Also, the semantics of the abort statement do not guarantee immediate completion of the named task(s). Completion must happen no later than when the task reaches a synchronization point. The ramifications of *abort* make the statement an extreme measure for terminating a task. An alternative to this is to use an entry in each task, which could signal completion and perform an orderly task completion from within itself. This alternative may also be used instead of using *terminate*. The entry-per-task alternative carries with it added program complexity and runtime overhead, but it may be a small price for an orderly termination which could be more efficient and may result in better performance.

Combining Ada tasks to solve a specific real-time embedded problem can be a complex undertaking. As stated in Borger [Borger 86], Ada tasks can be classified as actors, servers, or transducers based on their functional behavior. *Actor* tasks are active in nature and make use of other tasks to complete their function, whereas *server* tasks have a passive nature as they react to external requests from other tasks. *Transducer* tasks are similar in behavior to servers except that they are not totally passive since they require resources provided by other tasks.

Ada tasking structures for common constructs used in the design of real-time applications can be classified in one of the categories. For instance, a monitor is commonly used for controlling a systems resource. Such a task performs a "watchdog" function and would be classified as an actor task. In [SofTech 84], several approaches to implementing a monitor are discussed. A buffering technique is commonly employed. A buffer typically acts as a link between some producer/consumer task pair (both actors) and thus is a server task. Discussions on buffering implementations can be found in [Barnes 84] and [SofTech 84]. Another interesting example of a server task is an agent task [Barnes 84]. Agent tasks perform some action on behalf of another task. Typically a user task will make a request of an agent task. While the agent task is processing that request, the user task can perform other functions until the request is processed. This scenario promotes greater parallelism. Agent tasks are typically implemented with access types and thus are created dynamically.

3.2.2. Time Control

One of the most important concerns for a real-time application programmer is satisfying strict timing demands. The delay statement in Ada was designed to help satisfy those demands. Some applications have periodic timing demands, where it may be necessary to suspend a task for some predetermined amount of time. The semantics of the delay statement are such that the task will be delayed for at least the specified time. The programmer does not know the exact length of the delay. The actual delay time is determined by the runtime system and depends on many factors, including the degree of multiprocessing and the execution of higher priority tasks (see Chapter 4). Code inclusion (i.e., dummy loop) and clock readings could be used in place of the delay, though some of the same factors that effect the delay statement time will still influence the outcome. One obvious tradeoff between these two approaches is usage of processor time. A task with a delay statement temporarily releases the processor during the delay, thus enabling other tasks to execute. With dummy code, processing time is essentially wasted. The tradeoff here is the use of scheduling, which, in the case of the delay, accounts for the resulting uncertainty. Other tradeoffs are the simplicity of one statement and the degree of portability where specific implementation timing features may be used. One has to look closely at an implementation timing facility as well as investigate the issue of clock synchronization if the application is executing in a multiprocessor environment.

3.2.3. Input/Output

As mentioned in Section 3.1.3, embedded software systems often need low-level interfaces to hardware devices. More specifically, an Ada embedded system must have potential access to I/O ports, to control, status, and data registers (for a memory mapped scheme), to direct memory access (DMA) controllers, and to a mechanism for enabling and disabling device interrupts. The language-defined LOW_LEVEL_I/O package provides the SEND_CONTROL and RECEIVE_CONTROL procedures for interfacing to a hardware device. These procedures have two parameters, the device type and the format of data packets. Their specification is implementation defined since their kind and format depend on the physical characteristics of a particular device. Although there are only two such device control primitives declared in the language definition, they were provided for the purpose of being overloaded for various device types and data formats in order to cater to the diverse physical characteristics of many different devices. One can specify specialized device interface requirements to an Ada implementor and have these interfaces implemented by a tailored LOW_LEVEL_I/O package. The issue is whether the implementors provide what is needed for the programmer to have complete control and to exploit hardware as if programming in assembly language.

If an implementation is deficient in this area, an alternative to using this LOW_LEVEL_I/O package is to write your own device-specific interfaces either in Ada, using representation pragmas and clauses, or in another language provided that the interfaces can be linked with Ada code via the pragma

INTERFACE or code statements. The tradeoffs associated with these programming alternatives include the generality of the LOW_LEVEL_I/O primitives (i.e., "least common denominator effect" since they were designed to cater to many devices) versus tailored interfaces designed exclusively for a given device, and the application developer's loss of control over the implementation of these interfaces when they are provided by the Ada implementor via the LOW_LEVEL_I/O package.

When a real-time application developer chooses to process low-level asynchronous I/O operations to hardware devices using an interrupt-driven scheme, the software must be able to efficiently handle hardware interrupts. As described in Section 3.1.3, Ada supports interrupt handling via task interrupt entries. Issues relevant to the decision of whether to use this facility include:

- Performance: interrupt latency, runtime overhead, speed of interrupt scheme employed.
- Implementation restrictions on these interrupt entries: Can they be called from the application code? Can they have parameters?
- Implementation restrictions on the type of interrupt entry call: Is it a normal Ada entry call, a timed entry call, or a conditional entry call?
- Scheduling behavior.

Alternatives to using Ada interrupt entries include:

- Coding the entire interrupt handler (e.g., subprogram) in Ada and associating its starting memory location with the interrupt either through an address clause or by calling an executive service routine (in the case of an underlying operating system).
- Performing the above interrupt linkage but implementing the declared Ada interrupt service routine in another language (probably assembler) and using either pragma INTERFACE or code statements for the subprogram body.
- Relying on an existing device driver and interrupt service routine to handle device interrupts.

In situations when a real-time application developer chooses to process incoming data by polling the input device, the language must provide support for any combination of blocking or nonblocking and periodic or aperiodic polling schemes. As pointed out in Section 3.1.3, Ada can support these various polling techniques since it provides both timing control and a general-purpose looping structure. Issues relevant to the decision of whether to use Ada for polled I/O include its implementation of blocking I/O (e.g., a pending I/O request should not keep other tasks from the processor) and the accuracy of the timing control mechanism in support of periodic polling. In general, there is no better alternative than the Ada loop construct for implementing a generic polling algorithm; however, one may want to code (in assembly language) a more accurate timing mechanism than that offered by Ada runtime systems. The tradeoffs of this technique include better timing accuracy and control versus the cost, complexity, and maintainability of either having the runtime system provide a finer timing granularity or accessing the real-time clock directly through low-level application code.

3.2.4. Internal Representation

Inherent in the need of embedded software systems to interface to hardware devices is a requirement for efficient data representation in terms of the underlying machine's architecture. As pointed out in Section 3.1.4, Ada provides representation clauses which allow one to specify how the data types of an application are to be mapped onto the underlying machine architecture. Issues relevant to the decision of whether to use these clauses include whether or not they are supported; the degree to which they are implemented; the effect of their implementation; the supported granularity of control (e.g., bit level, byte level); and an application's need to control the maximum dynamic storage size, the internal representation of data objects, the internal codes used for enumeration literals, and the optimal use of storage. Alternatives to using the representation clauses include accepting the compiler's default layouts and mappings, or going outside of Ada and coding space-critical subprograms in assembly language. The tradeoffs between these choices include degraded execution efficiency and the loss of strict control over storage size, data layout, and storage optimizations for Ada code without representation clause versus increased complexity and decreased maintainability of the non-Ada code.

3.2.5. Error Handling

As noted in Section 3.1.5, the Ada language provides predefined exceptions and the potential for user-defined exceptions for dealing with errors or other exceptional situations during program execution. By default, during program execution an Ada runtime system automatically detects the exceptional situations with which the predefined exceptions are associated; when such a situation is detected, the runtime system raises the corresponding predefined exception. These runtime checks can be suppressed via the pragma SUPPRESS if it is supported. Issues relevant to the decision of whether to suppress these runtime checks include the execution overhead of performing them, the associated code size overhead, and the additional application code needed to provide the same level of error detection if they are turned off. The alternative to performing these runtime checks is suppressing them and either completely ignoring the possibility of runtime errors or providing additional application code to replace the suppressed checks. The latter is probably unrealistic since the developer has already chosen to suppress the runtime checks to obtain greater execution speed. The tradeoffs one must consider when deciding to suppress the runtime checks include whether or not the detection mechanism can be disabled (exceptions may be automatically detected by hardware and signaled via error trap interrupts), execution speed and code size optimization, loss of runtime error detection, loss of code safety, and the amount of additional application code needed to replace a subset of the suppressed checking.

The issues regarding whether to use user-defined exceptions are identical to those for the predefined exceptions; however, more programming alternatives exist. For instance, a real-time application developer can instrument the code with both pre- and post-subprogram call checks for examining the validity of the program's execution state and, thus, for detecting faults. Another alternative is to pass error return code parameters between both subprogram and entry calls and check their values after call return. The new tradeoffs associated with these choices include the amount of additional application code to perform pre- and post-call tests, the execution overhead associated with these checks versus the Ada runtime overhead for exceptions, the execution time overhead associated with passing extra parameters, and the complexity of non-exception implementations.

If an application developer chooses to use exceptions as the means of error handling, various programming idioms should be considered. Exceptions can be handled locally or moved to any unit in the dynamic calling chain. They can be handled by name or anonymously ("when OTHERS"). Depending on the programming style, exceptions may have to be declared in package specifications to export their name and to allow proper propagation. Another programming decision to consider is whether to provide the corresponding error handling code inline or remotely in a special, centralized error processing package. The method of use for predefined exceptions is a stylistic concern; however, it affects the testing and debugging process when application code depends on predefined exceptions. See the Texas Instruments report [TIAIM 85] for a further discussion of these issues.

3.2.6. Numerical Computation

The application programmer has the option in Ada to define data types that closely represent true physical quantities. Fixed and floating point types can be defined with constraints corresponding to actual limitations of the data. Constraint checking and data consistency can then be handled automatically by the compiler. (Pragma SUPPRESS can be used to eliminate constraint checking.) Because of strong typing, predefined operations are restricted to manipulating data of the same type. To get around this, explicit conversion may be used; but the benefits of automatic compiler checks may be lost. Another alternative is to use only predefined types instead of tailored user-defined types for each physical quantity. Using this approach, the benefits of strong typing are lost; practice reverts to FORTRAN-like code. Finally, operators (e.g. ***/*, *, +**) can be defined to accept data of a specified type (overloading [LRM, Section 6.7]).

3.3. Summary

The purposes of this chapter were to identify the features of the Ada language which specifically address computing problems characteristic of real-time embedded systems and to raise relevant issues, pose fundamental questions, and discuss possible programming paradigms relative to using these features. Ada's real-time features as identified in this chapter include the following:

- tasking: the predefined LOW_LEVEL_I/O package
- pragmas: PRIORITY, SHARED, INLINE, SUPPRESS, INTERFACE
- representation clauses: length, record, enumeration
- address clauses
- exceptions
- numeric data types
- the generic UNCHECKED_CONVERSION function
- the generic procedure UNCHECKED_DEALLOCATION

Below we present a list of issues, questions, programming alternatives, and programming idioms relevant to the use or non-use of each of these language features. Note that a programming idiom is a coding paradigm in which a feature can be used, whereas a programming alternative is an option to use when a particular language feature is not used.

Tasking

- What is the scheduling strategy employed? Is there more than one scheduling algorithm implemented? If so, can it be selected via a pragma?
- What selection algorithm is used to choose between many open accept statements?
- What causes rescheduling?
    - interrupts

    - expired delays
    - I/O
    - rendezvous

- How long is a task context switch?
- Do I/O operations from tasks suspend the rest of the program?
- How is runtime storage handled for tasks? heap? stack?
- What are performance metrics for common tasking operations?
    - activation
    - rendezvous
    - termination
    - abort

- How do the following affect program performance? [Pierce 86]
    - number of activated tasks
    - number of ready tasks
    - number of select alternatives in an selective wait
    - number of rendezvous parameters
    - parameter size
    - nested accepts

- What are the implemented semantics of the abort statement?
- What are the implemented semantics of delay statement?
- What is the value of DURATION'SMALL?
- What is the value SYSTEM.TICK?
- Programming idioms
    - static versus dynamic task objects
    - minimal code in accept bodies
    - terminating versus nonterminating tasks
    - actor/server task pairs
    - use of agent task pools

- Programming alternatives
    - cyclic executive
    - global data areas and semaphore primitives or a monitor task
    - global data areas and pragma SHARED
    - abort statement versus "kill process" kernel service

LOW_LEVEL_I/O Package

- Are interfaces to any standard devices provided (e.g., analog/digital converter)?
- Will the Ada implementor tailor this package to the application's devices characteristics?
- Are there any restrictions on the type and format of the primitive's parameters?
- Do the primitives provide an adequate interface to the system's devices?
- Can I/O ports and memory mapped I/O be handled in a straightforward manner?

- Programming alternatives
    - Write your own device-specific interfaces in Ada using representation pragmas and clauses.
    - Write your own device-specific interfaces in another language provided that they can be linked with Ada code via the pragma INTERFACE or code statements.

Pragmas (PRIORITY, SHARED, INLINE, SUPPRESS, INTERFACE)

- Which of the above pragmas are supported?
- How many task priority levels are supported?
- How are shared variables implemented?
- What is the effect of INLINE on execution performance?
- What is the effect of SUPPRESS on execution performance and code size?
- What languages can be interfaced with Ada?

Address Clauses

- Are address clauses supported in general?
- Are interrupt entries supported?
- Performance of interrupt entry calls
    - interrupt latency
    - runtime overhead associated with calling interrupt entry
    - use of a fast interrupt scheme
- Implementation restrictions on interrupt entries
    - Can they be called from the application code?
    - Can they have parameters?
- Implementation technique for an interrupt entry call
    - fast entry call
    - normal Ada entry call
    - timed entry call
    - conditional entry call
- How do interrupt entry calls effect scheduling behavior?
- Programming alternatives
    - Code entire interrupt handler (e.g., subprogram, task) in Ada and specify its starting memory location via an address clause.
    - Rely on an existing device driver and interrupt service routine to handle device interrupts.
    - Implement the interrupt handling code in another language and link it with Ada code via the pragma INTERFACE.

Representation Clauses (length, record, enumeration)

- Which of the above representation clauses are supported?
- Are there any restrictions on the use of the supported representation clauses?
- What is the supported granularity of control (e.g., bit level, byte level)?
- Does the Ada compiler use a space-efficient storage allocation strategy?
- What are the compiler’s default values for the size, in bits, for the predefined types?
- Programming alternatives
  - Accept the compiler’s default layouts and mappings.
  - Code space-critical subprograms in assembly language.

Exceptions

- Can exceptions be suppressed?
- What is the runtime overhead associated with managing exceptions?
- What is the code size overhead associated with using exceptions?
- Programming idioms
  - Handle exceptions locally without propagation.
  - Propagate back to and handle by the frame that is at the head of the call chain.
  - Handle by some intermediate frame.
  - Propagate anonymously back through the frames on the call chain and eventually trap by such frame via a "when OTHERS" clause.
  - Provide the corresponding error handling code in-line or remotely in a special (centralized) error processing package.
  - Restricted use of predefined exceptions.

- Programming alternatives
  - Instrument code with both pre- and post-subprogram call checks for examining the validity of the program’s execution state.
  - Pass error return code parameters between both subprogram and entry calls and check their values after call return.

Numeric Data Types

- What are the the compiler’s intrinsic integer representations?
- What is the value of MIN_INT, MAX_INT?
- What is the value of MAX_DIGITS?
- What is the value of MAX_MANTISSA?
- What is the value of FINE_DELTA?
- What are the values of MACHINE_ROUNDS, MACHINE_OVERFLOWS, MACHINE_RADIX, MACHINE_MANTISSA, MACHINE_EMAX, and MACHINE_EMIN?
- Programming alternatives (to relax strong typing and inherent constraint checking)
  - Use pragma SUPPRESS.
  - Use predefined types and avoid introducing new numerical (sub)types.
  - Overload common operators **(+, -, *, /)**

Generic UNCHECKED_CONVERSION Function

- Are there any restrictions on the source and target types?

Code Statements

- Which languages are supported?
- What restrictions apply to code statements?

For the most part, the programming alternatives for particular language features involve either using other features of Ada to produce the desired effect (e.g., busy wait loop versus the delay statement) or interfacing to a non-Ada solution. The subsections of Section 3.2 each, in turn, discuss specific tradeoffs associated with these alternatives; however, in all cases, there are common tradeoffs associated with choosing the appropriate implementation. A major consideration is the runtime system implementation and its performance, i.e., the execution and code size overheads associated with a feature. When using a non-Ada solution, other considerations include the adaptability, complexity, maintainability, readability, and portability of Ada versus non-Ada application code.

4. Ada Runtime Implementation Issues

This chapter describes issues concerning the implementation of Ada runtime systems that will affect embedded systems software. We present, in an informal manner, the Ada runtime semantics as defined by the *Reference Manual for the Ada Programming Language* [LRM 83], and we highlight the implementation-defined options that are available to the Ada compiler implementor. These are of considerable importance to an embedded systems designer as well as to any Ada compiler buyer in general. The runtime system that the compiler designer builds is one of the most important parts of an Ada programming environment. Much of the discussion in this chapter relates to the compiler designer and the choices available for developing the Ada runtime system implementation. We assume that the reader is familiar with the Ada language as given in the LRM.

The purpose of this chapter is to emphasize to embedded systems builders that the LRM permits Ada compiler implementors to develop quite different compilers for use in embedded systems. Further, it suggests the need for compiler buyers to be involved with the compiler designer's decisions about implementing many of the runtime features; and it discusses some of the overheads at execution time due to the Ada runtime system implementation (ART)—information an embedded systems designer needs to know.

A runtime system is the combination of software and hardware which supports language features and the execution of application programs. It consists of all the support mechanisms needed in the execution environment for a program and includes the instruction set, executive, processor, microcode, and runtime library. This chapter discusses the compiler-related aspects of the runtime system implementation. Ada has language concepts that require a complex runtime system to support them. For example, Figure 4-1 presents a picture of the contents of an Ada runtime system.

For an embedded system, it is most likely that the runtime system will be implemented on a bare machine—that is, one without significant operating system support, or one with a minimal kernel. The Ada programming environment must provide the real-time support facilities. An Ada compiler will translate the Ada program into target-machine code that includes procedure calls to the runtime routines that will support the execution semantics of Ada. Note that the distinction between the predefined runtime support library and the conventions of a compiler and its data structures is not always obvious [ARTEWG 86b]. This makes it difficult to separate the discussion between compile-time issues and runtime issues.

Compared to a language such as Pascal, Ada's high-level features reduce the amount of support that the programmer must provide but increase the support that the compiler and its runtime library must provide. Ada has thus eliminated fundamental design choices that were traditionally made by real-time system designers.

This chapter is divided into the following areas:

- memory management
- multitask management
- time management
- subprogram management

- input/output, arithmetic, and exception management
- pragmas
- implementation-dependent features found in LRM Chapter 13

The intent of this organization is to focus the discussion on a particular runtime function, such as memory management; this is in contrast to the organization of the LRM, which concentrates on language constructs. Each section discusses the implementation issues and options available to the compiler designer that affect execution characteristics.

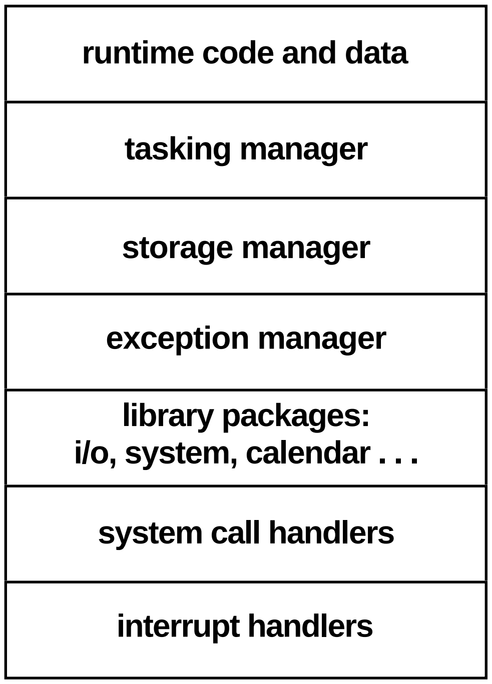

Figure 4-1: Aspects of an Ada runtime environment

4.1. Memory Management

An executing Ada program and its runtime library require storage for code and data. A runtime memory manager is needed to control the dynamic allocation and deallocation of data space for objects requested by the programmer, such as tasks, records, arrays, access collections, and procedures. It is also manages data required by the Ada runtime system implementation, such as attributes, task context control blocks, and spare storage units.

Memory management is quite complex because of the nature of the data types. The storage manipulation presents considerable execution overhead in terms of the amount of work to allocate/deallocate space, check status, and respond to exceptional and error-based situations.

Ada is a scoped language like Pascal but has more data types of greater complexity. It has local, shared, and dynamic data structures such as pointers, objects created by allocators, dynamic (unconstrained) arrays and records, and tasks. All designated objects pertaining to an access type are dynamic in that they need to be created and initialized at execution time. Runtime stacks for tasks are manipulated along with a dynamic pool of data for access types and for unused memory space.

Since embedded systems have limited memory resources, they require discretionary use of storage. Controlled use of Ada concepts, along with certain implementation strategies, can help an embedded system reduce overheads. Ideally, to reduce runtime overheads, all memory requirements must be known at compile/link time; but Ada has dynamic types. Experience has recognized difficulties with Ada's memory management [Bamberger 86, Hood 86, Laird 86, Ruane 85, Sonicraft 86].

The rest of this section is divided into four parts:

- storage allocation
- storage deallocation
- working storage limits
- storage layout

4.1.1. Storage Allocation

Implementation of memory allocation, apart from typical Pascal-like storage management such as stack allocation, involves:

- finding a chunk of memory of a suitable size for use as a task's runtime stack, or for an access collection, or for ART data such as a task's control block (TBC)
- managing the space within an access collection for access structures
- extending the access collection chunk if requested
- monitoring overflow of storage usage
- minimizing wasted space (if requested by the programmer via pragmas)
- maintaining spare storage chunks

Figure 4-2 represents a conceptual view of a memory space during an Ada program's execution. The task control block that serves as the runtime data descriptor for a task's executing context can contain task context information for suspended tasks such as register contents, as well as pointers to the task's runtime stack and associated components such as dependent tasks. Each task is likely to have a runtime stack that serves as the working memory for the execution environment of that task. The stack may contain information such as local variables and procedure-related data (e.g., static links).

Any storage allocation will involve seeking a free storage chunk of an appropriate size and then updating and initializing any runtime data structures to indicate the allocation. If there is no space available, the Ada runtime system implementation must raise STORAGE_ERROR or expand the memory space available. Storage allocation is affected by the following Ada features:

- creation and initialization of tasks
- entry of a new scope
- dynamic structure invocation via allocators
- task communication and synchronization
- pragma PACK and representation specifications

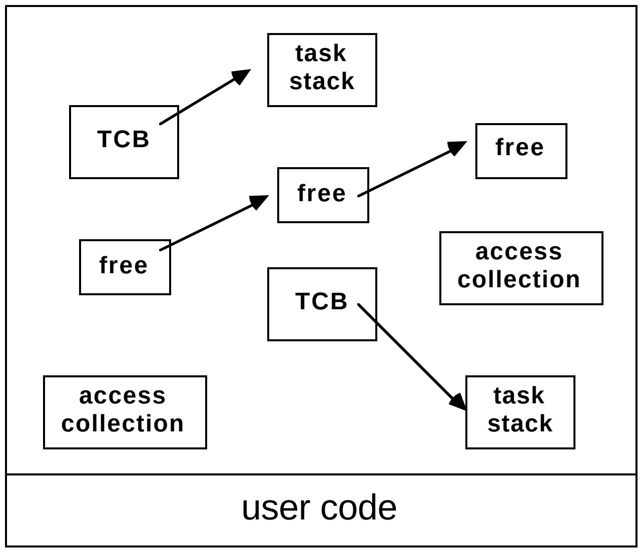

Figure 4-2: A conceptual view of an Ada program's runtime memory

Entry into a scope (for example, commencement of the main program, task creation, dynamic array allocation, or allocator evaluation) requires allocation of storage space along with data initialization. For example, an allocator involves dynamic storage allocation overheads since space must be found for the access structure and its access value must be delivered. Such storage must be free of the normal block-structure scope implementation mechanisms since the lifetime and size of the data may be unknown. This is commonly known as heap storage. One example of allocation complexity concerns variables defined within a package that are not visible outside the package. These are similar to Algol 60 "own" variables—that is, they do not change value between calls issued from outside the package to subprograms declared within the visible part. Such variables are created when the package is elaborated (within a program unit's declarative part or at library initialization) and are destroyed when the package goes out of scope. The ART must ensure this lifetime. Initialization can represent a considerable runtime overhead, particularly if it involves complex operations such as function calls, package elaborations, and task activation, which themselves involve significant storage allocation (and, possibly, deallocation). Library units need to be elaborated before the main program's code can begin executing.

Task interactions present overheads in terms of memory allocation. Each task requires some sort of task control block, by which the ARTs tasking manager can monitor its status and coordinate context switching. A task also needs workspace for a runtime stack. Tasks can be created and destroyed dynamically, which requires the ART to allocate task space and TCBs as needed. Tasks communicate and synchronize via a rendezvous which involves:

- Passing rendezvous parameters (generally based upon the method chosen by the compiler designer). The parameter passing may involve both the tasks' stacks or the allocation of a separate area for passing large structures. Returned rendezvous parameters may involve the copying of data from one task's data area to the other's. Entry call parameters have the same options as do subprogram parameters, as described in Section 4.4.
- Performing the appropriate constraint checking.
- Determining the availability of space.
- Maintaining any parameter's lifetime. (For example, a function may have a return parameter which is a task that is local to the function. The task is out of scope once the return from the function occurs, but status information must be returned for that task since queries can be made by the caller to that task.)

Pragma PACK [LRM 13.1(11)], if implemented by the compiler, requires the compiler to pack records and arrays. The compiler designer specifies the semantics of this pragma (such as alignment), but there is no requirement to implement any better packing strategy than the one the ART could provide regardless of this pragma.

Miscellaneous aspects that need better definition by the compiler designer and could incur runtime overheads, include:

- Elaboration order of components within the library: tasks, arrays, and other dynamic data will require storage; differing elaboration order can affect the sequence of initializations and possibly the memory requirements; this elaboration needs to be carried out before the main program begins.
- Allocation strategy for access types: algorithms such as best-fit or first-fit strategy for finding collection space from the spare memory have different runtime performance.
- Extension of memory: memory may need to be expanded; for example, unconstrained arrays/records with discriminants could require more memory than that allocated by default.
- Storage attributes SIZE, STORAGE_SIZE and the effect of pragma MEMORY_SIZE: the value returned from the attribute operation may or may not include the size of the runtime descriptor for each type; similarly, the designer determines whether the value for task STORAGE_SIZE includes the space for dependent tasks.
- Access types: the designer should determine whether any storage overheads can be eliminated if no access types are used [Grover 83].
- Library-level package: the designer should consider whether tasks in library-level packages can be statically allocated [ARTEWG 86d] rather than dynamically allocated.
- Overlap: a component may be allowed to overlap a storage boundary within record representation clauses [ARTEWG 86d].
- Record components with no component clause: an implementation places a record component which has no component clause in a record representation clause [ARTEWG 86d].
- Strategies for memory use: embedded systems use various strategies for making efficient use of memory; overlaying and selective linking may need to be implemented.
- Memory optimizations: swapping, partitioning, segmentation, and virtual memory may need to be considered for the embedded target system [Grover 85a, Lomuto 83]. A compiler buyer will need some set of guidelines and tests to determine differences in memory usage when applying these concepts [Ruane 85].

4.1.2. Storage Deallocation

Deallocation means that memory once named and used can become spare memory, to be used again for allocation. When deallocating dynamic storage, the ART must update its runtime descriptors of available space and add this freed space to the spare memory. Ordinary Pascal-like stack deallocation occurs for concepts such as procedure exit. More complex deallocation is discussed below in relation to:

- garbage collection
- pragma CONTROLLED
- UNCHECKED_DEALLOCATION procedure
- exception propagation

The compiler designer determines whether to implement garbage collection, which will reorganize the spare memory space, making more space available and allocation operations more efficient. Various strategies for garbage collection [Lomuto 83, Grover 85a] are dependent upon the allocation algorithm chosen, such as *automatic* or *on demand*. The semantics of Ada imply that a compiler can leave garbage collection to the latest possible time, if it is done at all, which may not suit embedded systems. A compiler buyer will need tests to determine when collection should actually be done. Tradeoffs must be recognized by the programmer when using garbage collection.

The compiler designer needs to consider the garbage collection strategy, including:

- How garbage collection is performed, for example, via reference counts.
- When it is performed, for example, upon scope exit or on the fly.
- How fragmentation of spare memory is handled, for example, coalescing is done periodically.

Noncontiguous memory could create problems with garbage collection [Sonicraft 86]. A generic storage manager [Bamberger 86] for supporting efficient deallocation of different data types may be worthwhile.

Pragma CONTROLLED [LRM Annex B], if implemented, requires the ART to delay any automatic storage reclamation for designated objects until the scope of the access type declaration (which could be the end of program) is left. This pragma could significantly affect the garbage collection algorithm which the ART uses. Pragma CONTROLLED may not necessarily provide any benefits regarding when storage is deallocated.

If the compiler designer chooses to implement the generic procedure UNCHECKED_DEALLOCATION [LRM 13.10.1], the ART must deallocate the allocated object without regard for any "dangling pointers," that is, without checking whether all objects have finished referencing the allocated object. In effect, the programmer now has the responsibility for ensuring that it is safe to reclaim storage. The LRM does not define the consequences of subsequently accessing the deallocated object via aliasing, so the ART is free to implement this deallocation in any way. The compiler designer defines completely the semantics of the UNCHECKED__DEALLOCATION procedure.

Depending on the exception-handling mechanism that the compiler designer chooses, exceptions may involve deallocation of stack space during exception propagation. For instance, searching for an exception handler may involve allocation of data space and forced deallocation of stack space during exception propagation by, for instance, "unwinding" the stack in order to propagate out of scopes. When an exception propagates to outer scopes, the system requires termination actions for each inner scope, which will involve deallocating any local space for tasks, access collections, arrays, and any other dynamic structures. This subject is discussed further in Section 4.7.

4.1.3. Working Storage Limits

Working storage is the memory space available to the executing program for its lifetime. The LRM allows the compiler designer to choose the amount of allocated memory for data as long as the minimum size, if requested by the programmer, is allocated. Hence, it may be obvious to the programmer if space allocated could be larger. In an embedded system, actual memory space used is a significant issue to the programmer. Features concerning the amount of memory allocated in Ada include the following:

- representation specifications
- length clauses
- pragma MEMORY_SIZE
- exceptions

Representation attributes such as POSITION, SIZE, and ADDRESS [LRM 13.7.2] can be interrogated in order to gain information about storage characteristics. In some cases, the compiler designer chooses the interpretation of the value to be returned. For instance, X'SIZE gives the number of bits used to hold object *X*. If *X* is a task object, it is not clear what value is returned—it could include the number of bits making up the task control block, or just the pointer to the TCB. Similarly, the meaning of representation attributes for querying addresses and storage limits is defined by the compiler designer.

Ada's length clauses [LRM 13: SIZE, STORAGE_SIZE, SMALL], if implemented, require the compiler to set limits on the amount of storage associated with a type (e.g., integer, access collection, or task). The sizes allocated must be at least the minimum requested by the programmer but can vary beyond that. If the programmer does not specify a size in the program, the compiler chooses a default size. The compiler designer chooses whether the size associated with the length clauses includes the space needed for the runtime descriptor associated with the type. The designer also determines whether task storage allocation has fixed limits, what types of storage are fixed, and the default values for the amount allocated.

Pragma MEMORY_SIZE [LRM Annex B] requires the compiler to set the bound for working storage size for the program. The compiler designer decides whether that value includes the space for dynamic structures such as dynamically allocated objects. Pragma MEMORY_SIZE requests only a minimum working storage space. The compiler could actually allocate a larger space. A programmer may need to know the default size of blocks allocated for units.

Overflow of memory space must be monitored by the ART when length clauses are used. In this case, the STORAGE_ERROR exception should be raised or memory should be automatically extended. Monitoring overflow is generally an expensive operation because the checking routine must be called frequently. For example, it could occur every time some space is required on the runtime stack. Any solution will provide overhead at compile time, procedure prologue/epilogue time, or any time storage space is required on the runtime stack, for example, for a temporary variable. A programmer will want to know if there is a mechanism for determining the availability of working storage [Grover 85a, Lomuto 83].

Following are some other issues which the compiler designer must resolve in order to assist the embedded systems programmer:

- information about the absolute addressing limitations on memory [Grover 85a]
- tests for measuring maximum/minimum allocations for tasks, subprograms, packages, and access types [Ruane 85]
- memory saturation

Questions that must be asked include:

- How many levels of nesting of procedures are possible and how much dynamic memory is available to individual tasks?
- What diagnostics are available?
- Can memory tests be performed?
- Can thrashing be detected?
- What are the difficulties in detecting overwrites [Grover 85a, Lomuto 83]?
- How much storage is used by the ART and its data?

4.1.4. Storage Layout

The Ada semantics imply a rather complex runtime organization of storage space. Code and data space is required for every task. The sharing of data by tasks (discussed in Section 4.2.4) requires that data be accessible by all the appropriate tasks. Structuring the compiler's runtime system code itself could affect some optimizations since only the ART must access the TCBs, whereas user tasks need not. Also, dynamic nesting of program units adds another level of indirection to accessing data, which can add runtime overhead. This can have important ramifications for microprocessors that have unusual segmented memory techniques.

Structure is important to embedded systems since memory is a scarce resource and needs to be used efficiently, by placing data structures in readily accessible places. Some targets have specific code and data areas (for example, read-only memory), and virtual memory may be limited. Embedded hardware systems usually have novel requirements for code versus data spaces. The runtime system requires a special area for its own data structures to which user data will not need access.

The following features affect the layout of runtime storage:

- pragma PACK
- representation specifications for enumerations and alignment
- address clauses

Pragma PACK [LRM 13.1], if implemented, requires the compiler to implement an allocation algorithm for arrays and records which will attempt to minimize storage fragmentation within the structure. This may place an overhead on top of the normal runtime component access mechanism. This pragma has been previously discussed in Section 4.1.1.

For enumeration type representation clauses [LRM 13.3], the compiler must generate the programmer-defined codes for enumeration types instead of its own. This is likely to affect the nature of optimized code and hence execution time since some sort of mapping table may need to be accessed—the compiler maps the representation clause for record component layout into a form that the programmer requested, which may not be that which the compiler would naturally choose. This may affect efficiency of component access since the normal access algorithm must be overridden.

Address clauses [LRM 13.5], if implemented, require the ART to force an object, such as a variable or constant, to reside at a particular address. The ART must ensure that a subprogram, package, or task unit starts its object code at that address. The ART may override its default allocation strategy to impose this restriction. Specific pragmas can be defined by the compiler designer which affect storage structure at runtime. In particular, the Ada designers recognize that overlaying of memory is a desirable feature but they leave it to the compiler designer to specify and implement in any way (as long as it is not implemented by overloading address clauses).

Some miscellaneous issues that the compiler designer needs to resolve:

- Whether the initialization routine can become overlayed by working storage.
- The definition of new pragmas.
- The support of partitioning, swapping, overlaying, segmentation, virtual memory [Ruane 85].
- The procedures for orderly shutdown of the whole system and how they affect the contents of memory.
- Consideration of reconfiguration of memory for fault tolerance.

4.2. Multitask Management

Ada provides concepts for concurrent execution of sequential pieces of code. An ART may implement these concepts on a single machine or a multiprocessor system. It must monitor, create, delete, switch, suspend, resume tasks, query the status of tasks, schedule, and allow tasks to communicate in an orderly fashion. Ada's tasking facilities may not meet some of the performance requirements of an embedded system where time criticality is top priority, unless some customizing and optimizing are carried out or certain programming paradigms are used. This section is divided into subsections discussing the concurrency model, scheduling, communication, shared memory, and multiprocessors.

4.2.1. Concurrency Model

Ada provides tasks that can be dynamically created, activated, destroyed, and run concurrently. Ada semantics imply considerable runtime overhead. The complexity of implementing the tasking model is described below through the following features:

- task relationships and elaboration
- status information
- task abortion, completion, and termination
- exceptions

There is a strong relationship between tasks that the ART must recognize and maintain. This relationship affects the amount of parallelism and the order of task termination. A task spawning another task is the parent of that task; the parent task is delayed while the child task is activated. Any problems in activation are relayed back to the parent, who assumes responsibility for taking corrective action. LRM 9.4 sets up a master-slave relationship. A master (that is, a unit whose execution creates the task object) cannot terminate or abort until the slave task (and its siblings, at least) have done so. Slave tasks can be collectively terminated, that is, a group of related tasks are forcibly terminated under a complex set of conditions involving their state and that of the master, the master's siblings and the group's own siblings, and the inability to rendezvous.

The ART must keep track of the relationships and status of tasks, along with saving the machine's context when switching between task executions. For each task, some sort of runtime tasking data descriptor, a task control block will contain all necessary status information. Tasks can complete normally, be aborted, or terminate collectively if no rendezvous is available. This must all occur in the appropriate order based on the task dependency hierarchy, which the ART maintains and adds to when tasks are created [Reino 86]. The master-slave, parent-child relationship of tasks gives rise to complexity in the ART and affects the efficiency of elaboration/termination/abortion of tasks. Task termination based on the dependency hierarchy is very costly and complex to implement because of data structures and accessing. Efficient algorithms for task termination are needed [Reino 86].

Tasks provide a considerable overhead in the runtime system because the status must be maintained for context switching, task dependencies, task abortion and termination, and task communication and synchronization. The depth of the nested task hierarchy will influence the amount of runtime overhead. Compilers that remove tasks by optimization are needed. Tasks can be reduced to a sequence of procedure calls [Pepper 86]. An ART without the tasking code will obviously reduce runtime code size and save context switch time. Context switch time becomes a very important measure for a system with a large number of tasks. Other important measures for users include: maximum number of tasks allowed in the execution environment at any one time; maximum length of entry/delay queues [Ruane 85]; and time consumed for task elaboration and for activating and terminating a task [Clapp 86].

Although not explicitly stated in the LRM, a main program can be considered a task by the ART. This is obvious since a main program has a priority and can perform actions which only tasks can, such as delaying itself and making an entry call. Hence, the main program task may require a parent and master so that it fits into the task dependency hierarchy. The ART must have its own elaboration phase whereby the initial runtime data structures and library packages are elaborated. The compiler designer must decide when and how library tasks are terminated. The requirement by the LRM is that the library tasks do not terminate at least until the main program has completed, and these tasks need never terminate. The LRM does not define what happens to tasks declared in library packages when the main program terminates [ARTEWG 86d]. Further guidance, though, from the Language

Maintenance Panel indicates that the user's program—not the compiler designer—determines whether termination of the tasks occurs.

The ART must provide operations whereby the program can query the status of tasks. This is accomplished through task and entry attributes [LRM 9.9]: T'CALLABLE, T'TERMINATED, and E'COUNT. The information will most likely be kept in the TCBs. Queries must be implemented as indivisible operations.

A task can abort another task within scope (including itself) at any stage. Synchronization points for a task's execution status are defined as in LRM 9.10.6. Synchronization points occur at:

- abort statements
- the end of a task's own activation
- a task's activation of another task
- an entry call
- the start or end of an accept statement, a select statement, a delay statement, an exception handler

Exception handlers require the ART to check the status of tasks for abnormality and exceptions before allowing the task to execute past the synchronization point. Abnormality checks (that is, language-required runtime checks for the abnormal status of a task) impose considerable overhead for tasking since they need to be carried out at every synchronization point. The ART must instigate task abortion or exception propagation before the next synchronization point is reached if conditions are ripe for doing so. For task abortion, the ART must ensure that tasks terminate based upon the task dependency hierarchy. This hierarchy entails considerable runtime overhead, especially for task abortion. Embedded systems generally need to delete tasks so that the resources such as space can be reused. Minimizing tasks can save context switches and scheduling time. Task abortion does not involve an instantaneous, sledge-hammer kill, but rather a graceful, orderly shutdown of processes based on task dependencies. A special form of "instantaneous kill" may need to be supported by the implementation. The compiler designer determines whether a task can be aborted (and go to completion) while updating a variable [ARTEWG 86d]. It also determines the semantics of an abort [LRM 9.10] statement with multiple tasks that can indicate any ordering for the task abortion. Programming style–or rather, structuring of source code–can considerably affect the runtime characteristics of tasks. For example, tasks terminate via a terminate alternative depending on who the master is; if the master is a library package, then tasks need not terminate.

The ART must guarantee that exceptions do not propagate outside of the scope of tasks except in the case of a rendezvous (for an unhandled exception) and when a child task has an error during its elaboration.

A user would like to know the maximum number of active tasks and main programs; length of entry and delay queues; the maximum number of entries per task and level of nesting of tasks; and the number of parameters that can be passed during rendezvous [Ruane 85].

4.2.2. Task Scheduling

Some of the complex runtime issues related to task scheduling are:

- priority
- scheduling strategy
- suspension and resumption

As long as tasks with higher priority run ahead of those with lower priority when it is sensible to do so, Ada permits any scheduling strategy [LRM 9.8]. The language defines lower as "a lower degree of urgency" [LRM 9.8]. The programmer gives a priority to a task (or the main program) via pragma PRIORITY [LRM Appendix B], or the compiler gives a default priority. Priorities may not be implemented at all, but the ART must have scheduling rules for all tasks. The compiler designer determines the priority of a rendezvous without explicit priorities [ARTEWG 86d]. Priorities are static in the sense of "once set, always set"—except during a rendezvous, where the ART must change that of the task with lower priority to temporarily assume that of the higher. Priority can only affect task scheduling; it cannot affect tasks in queues, for example, awaiting rendezvous. There are known problems with Ada's priority scheme, such as the priority inversion problem [Cornhill 87], which a compiler for embedded systems may have to circumvent. For instance, since a task is suspended until the completion of subtask activation, it is possible that a low-priority activation could result in a very long suspension of a high-priority task. This is definitely a problem for embedded systems designers who must circumvent these semantics of Ada.

The compiler designer determines pragma PRIORITY, the values of the attributes FIRST and LAST of the subtype PRIORITY, and the default priority level values. The designer also decides the effect of priorities on queuing for rendezvous and real memory, task start-up/elaboration, exception raising, task termination, and I/O access. Ada's priority-based scheduling strategy provides an implementation-dependent option for resolving fairness and starvation in scheduling; for example, round-robining or FIFO resumption of delayed tasks of equal priority. A cooperative or preemptive scheduling model can be implemented. Priorities are static in Ada, but an embedded system is likely to want to dynamically change priorities based on certain events for reconfiguration, fault tolerance, etc. The library system may need to include some mechanism for customizing priorities [ARTEWG 86c]. The priority range must be sufficient to suit the programmers' needs.

A cooperative (that is, run until blocked) or a preemptive (that is, run until interrupted) scheduling strategy can be implemented. Foreground tasks (tasks with higher priority requiring frequent execution) and background tasks (lower priority tasks which only need to execute when the processor has any spare cycles) are a normal, embedded system executive strategy [Hood 86]. Round-robin and time-slicing scheduling may need to be supported [ARTEWG 86c]. Ada does not permit the primitive control over the runtime system that allows a cyclic executive to be efficiently implemented [Grover 83]. Coding a cyclic executive in Ada does not solve any new problems and is not as efficient in solving many of the problems traditionally addressed by cyclic executives. A data-driven executive may not suit embedded systems [Grover 85a]. The compiler designer determines whether user-defined schedulers are implemented and whether it is necessary to write the scheduler in assembly code for sake of efficiency.

The LRM does not require deadlock detection, but it is possible that a compiler can detect any obvious deadlock situations such as a task calling its own entry. An embedded system cannot afford to "hang," so it is likely that the ART will implement extra features for detection and invocation of recovery/restart (hot or cold). The compiler designer determines whether to implement any capabilities for detecting and recovering from deadlock [Ruane 85] and for detecting infinite loops.

As part of the scheduling model, the ART must suspend and resume a task's execution for an amount of time. The delay statement [LRM 9.6] and timed entry call [LRM 9.7.3] require suspension of tasks. The ART will most likely have several queues of tasks: one for tasks that are executable and one for tasks that are delayed. Using some sort of timer mechanism, the ART suspends the task. The ART is only required to delay the task for at least the minimum requested time. The compiler designer can choose when, beyond the minimum amount of time, and whether to resume the task, if at all. Such an implementation may not be accurate enough for an embedded system where events must occur at a certain time or within a time period.

The following measures are likely to be important for embedded system designers:

- the range of context switching time
- overhead time for task creation, termination, and abortion
- the duration in which all interrupts may be inhibited
- the precision of delays and timed entry calls

A compiler designer may develop library packages for programmer code which may require non-preemptible sections [ARTEWG 86c]. Task identification requires customized features [ARTEWG 86c].

4.2.3. Task Synchronization and Communication

Implementing communication tasks is complex because of the following features:

- different kinds of rendezvous
- exceptions
- interrupts

To implement a rendezvous, the ART must ensure that tasks are synchronized. This means the ART must be able to suspend a client task until the server task is ready to communicate. A rendezvous involves a client task that makes an entry call to the server task, which will at some stage make a corresponding accept (if it has not done so already) to start the rendezvous. The ART will place the client task in the FIFO entry queue of the server task for that particular entry. It must suspend the client task (if necessary) to await a corresponding accept and during the rendezvous itself. At the end of the rendezvous, the ART must resume the client task after checking for unhandled exceptions and abortion. Conditional rendezvous add complexity to implementing a rendezvous:

- For a server task, the compiler designer defines a strategy for determining which select alternatives within a selective wait [LRM 9.7.1] are candidates for execution. The ART chooses the best alternatives in accordance with LRM semantics. If a rendezvous is possible, the ART initiates it. Otherwise if there is a delay alternative, the ART suspends this server task and prepares for a timeout on that task. Otherwise, if there is a terminate alternative, then ART changes the status of runtime descriptors indicating that this task can potentially terminate. Then, using a complex policy based upon the task dependency hierarchy regarding master-slave relationships, it determines whether to initiate the task's termination. There may be a limit to the number of entries per task. If not, minimizing the entries per task may help reduce access times (depending on how the ART implements entries and relates them to tasks).

- For a client task, a conditional entry call [LRM 9.7.2] requires the ART to disallow a potential rendezvous if the server task is not ready at the time of the entry call to communicate; the alternate actions of the call must be initiated by the ART.
- For a client task, a timed entry call [LRM 9.7.3] requires the ART to suspend the client task for a minimum amount of time and initiate some sort of timing mechanism which will cause a rendezvous to occur if the server task is ready to synchronize/communicate within a specific period of time; otherwise, it resumes the client task.

A compiler designer determines such issues as: the order of evaluation for guard conditions in a selective wait; when the delay alternative (if present) is evaluated; when the family indices (if present) are evaluated; for selective wait alternatives, the algorithm to determine the selection from the open alternatives in a selective wait; and, for delay alternatives, the algorithm to determine selection from delay alternatives of the same delay [ARTEWG 86d].

Fairness and determinism in selecting rendezvous and interrupts are options the compiler designer must choose. Designers can choose special optimization techniques which could include the following:

- Within a standard library package, generic synchronization primitives for coding efficient generic real-time concepts such as semaphores and buffers.
- Compiler optimizations for minimizing context switches during rendezvous such as Habermann-Nassi, monitor clusters, Rajeev and Greene [Hood 86].
- Transforming tasks into procedures to avoid cyclic scheduling [Pepper 86].
- Fast interrupts—that is, rendezvous which are treated like procedure calls and executed on the caller's runtime stack can eliminate tasking overheads and reduce interrupt response time. This may require all runtime tasks to be visible to the entire system, which may not be possible due to memory limitations [ARTEWG 86c, Lomuto 83, Pepper 86, Hood 86]. Asynchronous or synchronous or mixed task synchronization facilities may be needed [Lomuto 82]. Techniques for implementing synchronous and asynchronous events exist [Grover 85b, ARTEWG 86c].

At the start of a rendezvous, the ART must raise the TASKING_ERROR exception within the client task if the server task has already terminated or been aborted. Also, if there is an unhandled exception at the completion of the rendezvous, the ART must raise it at the client task site as well as raising it within the server task and initiating all the necessary exception handling and propagation actions.

Interrupts [LRM 13.5.1] are treated as entry calls by a hardware "task" whose priority is higher than that of the main program and any user-defined task. This interrupt can be any kind of entry call (timed, conditional, or normal). Implementing a hardware interrupt involves mapping a hardware signal into a high-priority, Ada entry call. This hardware signal must look like an ordinary software entry call to the ART. The compiler designer defines any semantics for this kind of entry call, such as parameters, storage area, and exception raising. The interrupt must be given highest priority, locking out other activities. The LRM does not define whether interrupts are treated as conditional entries and therefore lost if not serviced immediately. An implementation may define additional conditions for terminating the task that contains the entry. The hardware could directly execute the accept statement. The interrupt entry call needs to have only the minimum semantics given in LRM 13.5.1 (rather than that of a task in rendezvous). An interrupt need not invoke any scheduling actions. Enabling/disabling interrupts must be customized, and the priorities of nested interrupts must be resolved [ARTEWG 86c]. The compiler designer defines any further semantics as to where the rendezvous is executed, e.g., on a stack. The designer also determines the restrictions on terminate alternative—further requirements that are imposed by an implementation for selecting the terminate alternative that may appear in the same select statement with an accept alternative for an interrupt entry [ARTEWG 86d].

Measures that embedded systems designers would be interested in include:

- minimum rendezvous time [Clapp 86]
- the maximum time duration in which all interrupts may be inhibited [Ruane 85]
- timing accuracy for the delay statement and for timed entry calls
- interrupt response time [Clapp 86]

For many real-time applications, a 12 millisecond time for synchronizing during a rendezvous is too long. A basic context switching time of 20 microseconds is reasonable [Laird 86].

4.2.4. Shared Memory

Tasks that share data, indicated by pragma SHARED [LRM 9.11], seem to require the ART to implement mutual exclusion with the guarantee that reading or updating this data will be treated as critical. The compiler designer defines the status of the data if its defining task is aborted [LRM 9.10(8)]. The shared data will cause an overhead due to the mutual exclusion operations required on the data. Local copies of the data are possible. Designers need to define the implementation mechanism for maintaining the local copy and resolving any update anomalies as well as defining whether shared variables are protected by rendezvous and whether multiple reads are allowed simultaneously. Pragma SHARED can be applied only to scalar and access objects [LRM 9.11(10)].

In general, tasks can share data as a result of common scope. The ART is not required to implement this data within a critical region, so the programmer must take responsibility for guaranteeing safety of data access.

Generic units suggest a sharing of code. There is no requirement on the compiler as to how it implements generic units. It can choose to share code for each instantiation, which can help minimize memory usage, or it can generate a copy of the unit's code with actual parameters. This could reduce some execution time for accessing data but quite likely would add to the amount of memory used.

4.2.5. Multiprocessors

The LRM provides no explicit facilities to address implementation on multiprocessors. But the intent is that the semantics of the runtime system do not prohibit such an implementation. It is likely that the ART will need mechanisms [Ardo 83] that the compiler designer must choose for doing the following:

- Detecting status of tasks across machines: Remote procedure call for distributed rendezvous will probably be needed for a rendezvous. A communication protocol in effect needs to be implemented. [Ardo 83]

- Scheduling tasks across machines: How is distributed task scheduling carried out? How are task termination dependencies (abortion/collective termination) enforced for distributed processors? What is the algorithm that determines the execution order of the activation of tasks on different processors? [ARTEWG 86d] How sensitive is preemptive scheduling to distribution of tasks over processors? [ARTEWG 86d] When is allocation of tasks to processors performed at compile/link or runtime? Can the programmer control the allocation [Lomuto 83]? Can a task allocate, rendezvous with, or abort a task residing on a different processor? [Lomuto 83]
- Coordinating data access: What technique is used for shared data, for example, a common memory or local copies? Is there a common pool or local pool of dynamic storage for access types?
- Distributed rendezvous: For a rendezvous among separate processors, how are objects transmitted, and what are the time delays due to rendezvous between different processors? For parameter passing, which will need a protocol, what are the storage ramifications and the method? With two tasks in a multiprocessor system rendezvous, how are task priorities among processors resolved? On which processor is the rendezvous performed? [ARTEWG 86d] What are typical time delays for rendezvous between tasks on different processors? [Ruane 85]
- Global timing facilities: What are the clock synchronization problems among different processors? Will there be local or global clocks?
- Code sharing for objects of the same task type: Can the code of a task body be shared among multiple occurrences of the same task type?
- Miscellaneous: Can generics be instantiated for remote processors? Can program units be distributed to remote processors? How can auditing tasks and data be performed? What are the ramifications for exception propagation out of remote tasks?

4.3. Time Management

Time management relates to the following features:

- package CALENDAR and system timer
- type duration
- delay precision

The ART must implement the package CALENDAR [LRM 9.6], which provides the implementation-dependent definitions of time and date types (e.g., TIME, YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER, DAY_DURATION) along with their operations (such as Clock, Year, Month, Day, Seconds, etc.) and TIME_ERROR exception. Implementing these operations involves the mapping of the target system's timer into the ART. Some ranges of timing types are language defined. The accuracy of the timing functions will depend upon the precision of the embedded system's clock and the precision of fixed point arithmetic. A timing range of up to one day (in seconds) is required. Durations must be implemented with, at most, a maximum value for DURATION'SMALL of 20 milliseconds and a recommended value of 50 microseconds. SYSTEM.TICK must be given a value to represent the basic clock period.

Upon a delay statement [LRM 9.6] or a timed entry call [LRM 9.7.3], the ART is only required to guarantee the suspension of the task for the minimum of the time specified. On average, a delay is likely to be longer than specified because of the time it takes the ART to recognize the expiration of a delay, reschedule, and resume the task. This is often acceptable, but (for example) in a cyclic program intended to execute at a prescribed frequency, every repetition comes a little late; there is a cumulative drift in the time of execution, as well as jitter in the actual intervals. Techniques for eliminating the drift may be needed [Downes 82]. Ada does not permit delaying up to a certain point in time, either for a task or the main program.

Issues which the compiler designer must resolve include: What is the CLOCK accuracy regarding DURATION'FIRST, 'LAST, and 'DELTA? What are the values of FINE_DELTA and TICK? Does package CALENDAR support Julian days? What is the accuracy of the timer? Does the embedded system have more than one timer? If so, does the ART make use of each or just one? What is the frequency with which all time-dependent conditions are checked (for example, expiration of delays, next time-slice)? Should there be facilities for time zones, different calendars, universal time, within package SYSTEM? What is the representation of types DURATION, DURATION'SMALL, and SYSTEM.TICK? Is there a known upper bound on a delay? A delay may be required that is less than that incurred by the execution-time overhead of the ART's implementation of the delay. Is the delay expiration a scheduling event? What is the overhead of a call to, and return from, function CLOCK? [Clapp 86]. Are there any facilities for providing an accurate measure of elapsed time between events? Can a programmer use a "pool" of timers as software "watchdogs"? [Lomuto 83]. ARTEWG [ARTEWG 86c] suggests possible solutions that could provide for commonality among Ada compilers.

4.4. Subprogram Management

Procedures and functions are subprograms [LRM Chapter 6]. These imply a stack-based implementation mechanism. Parameter modes *in*, *in out*, or *out*, demand that read-only and update capability be provided by the compiler. The compiler designer must decide upon the mechanism for passing parameters given that scalar types (integers, reals, and enumerations) and access types have to be passed by copy. Other parameters can be passed by copy or by reference as determined by the compiler designer. Association between formal and actual parameters (which can be evaluated in any order), along with returned parameters, must be chosen by the compiler designer. The compiler designer may decide to give the user some control over the parameter choice of passing mechanism [ARTEWG 86d].

Mechanisms for returning results, especially record and unconstrained array types, can affect space requirements and efficiency. The option may depend upon memory structure and access times. For any copying, this will need to be done within a critical region. Parameter size can affect passing strategy. For example, is it necessary to pass a large parameter by copy? Some miscellaneous issues which the compiler designer must resolve include: Where is the space for parameters allocated? What is the effect of using global data for parameters? What is the order in which they are passed? In addition, depending on the size and timing requirements, tradeoffs for the method of parameter passing need to be determined. An evaluation order policy for association between formal and actual parameters and returning results is determined by the compiler designer.

Pragma INLINE [LRM 6.3.2], if implemented, generally requires the compiler to expand inline the subprogram body. The programmer can expect improved execution performance with this although there can be a space penalty for multiple copies. Pragma INTERFACE [LRM Annex B], if implemented, requires the compiler to interface Ada subprograms with non-Ada code. This has serious ramifications on the execution environment since the calling conventions of the non-Ada language are likely to be different from that of Ada. Hence, the compiler designer must define any limitations or restrictions on the code that can be interfaced. For example, the ART must be able to handle non-Ada code that may pass parameters by a method not available in Ada itself.

Because of fault-tolerance requirements, an embedded system needs to detect overflow of storage space, particularly the runtime stack. Subprogram call (and any other scope entry) can result in overflow of the stack space available. The ART can choose whether it will provide additional features for checking overflow on procedure/function invocation. Techniques involve adding extra checks to prologue code or to each push onto the runtime stack, or having a special marker to indicate end of stack. This can present considerable overhead. Calling depth can be effected by the amount of recursion and nesting level.

Embedded systems measurements would include: time to pass parameters; subprogram call and return overhead; time for intra- and inter-package subprogram calls; time for instantiation of generic code [Clapp 86]; overheads of constraint checks on subprogram call and return; limits such as maximum number of subprograms, level of nesting, number of parameters; and number of formals in a generic subprogram [Ruane 85]. Compilers may include optimizing facilities such as tail recursion for recursive subprograms to eliminate any subprogram calls, which reduces the necessity to save system status (registers). Each generic instantiation is allowed to have a different ordering of its generic actual parameters, which may affect any expressions with side effects [ARTEWG 86d].

4.5. Input/Output Management

Input/output is very implementation dependent. The four kinds of input/output for Ada, as defined by the LRM Chapter 14, must be implemented as library packages by the compiler. The compiler needs to map the language-defined I/O operations to those of the underlying target system's file support utilities, if any. External file facilities must provide for binary and ASCII information. Some operations and default parameter values (for example, DEFAULT_BASE) are specified by the language to aid formatting and constraints. For sequential I/O and direct I/O [LRM 14.2; binary, sequential, and random external file accessing] and text I/O [LRM 14.3; ASCII, human-readable input/output to devices such as a terminal or printer], Ada specifies operations and exceptions. For low-level I/O used to control physical devices, the compiler designer must define the syntax and semantics of this form's operations, exceptions, and device access protocol along with implementing the SEND_CONTROL and RECEIVE_CONTROL primitives for interfacing to the devices.

The low-level I/O is likely to be the main form of I/O used by embedded systems. This may require files for maintaining persistent data such as recording statistics, so it will also require direct or sequential I/O. Similarly, there may be a need for terminal I/O for monitoring purposes, in which case text I/O will be used.

The ART must enable the communication between the LOW_LEVEL_IO package and the actual device. These needs will extend to the high-level and text-level portions of Ada I/O if all I/O requests are channeled through the LOW_LEVEL_IO package. Alternatively, rather than using the low-level package as an intermediary, the high-level and text-level packages may interface to the device drivers directly by employing such Ada facilities as address specifications and machine code insertions. Anytime I/O is to be performed, the ART will probably be involved in the operation since access to the physical resources may be protected and privileged only to the ART. Utilization of the resource may require the ART to suspend the execution of other tasks. [NAVSEA 83].

I/O_EXCEPTIONS is a package which the compiler must implement. It defines all the exceptions that can result from any input/output operations. The ART must map any file accessing errors, device problems, or parameter anomalies into Ada exceptions. The runtime stack must be in a state which the ART's exception manager can use to raise exceptions.

Issues and questions which the compiler designer must resolve:

- The language does not define what happens to external files after the completion of the main program. For example, what if a file is not closed by the end of a program? It is up to the implementation to decide the ramifications of input/output for access types—what does it mean to read/write a pointer value? There are no language limitations concerning the number of files that a programmer may associate with any given external file, nor how many file modes may be associated with an external file.
- Effects of scheduling, synchronization, and communication of tasks with I/O operations are left to the compiler designer.
- Size of files, existence of temporary files, external file associations, effect of reading uninterpretable elements, terminators, buffering, and representation of nongraphic characters are decided by the compiler designer [ARTEWG 86d].
- Input/output for enumeration types [LRM 14.3.9] could be used for integers although that is not intended. The language does not define the consequences of such usage.
- Limits are a concern. How large can a file/data value be? What are the maximum lines/page, characters/line, pages/file? The maximum/minimum size for disk I/O? Maximum length of various I/O queues? Maximum number of I/O devices and buffer sizes?
- What restrictions apply to types that can be instantiated for I/O? Is binary I/O supported? Is asynchronous I/O supported for character and block-oriented devices? Is formatted I/O supported? Is real-time target system data collection by host computer supported? Are time-outs detected for I/O requests? Can one external file be referenced by more than one program unit concurrently? Can separate tasks write to the same screen without interfering? Can a task perform an asynchronous I/O operation [Ruane 85]?
- What are the effects of disconnecting peripherals? Must device drivers be written in Ada? What happens to files when the main program has ended? What capabilities exist for maintaining file security? What is the minimum disk access time?
- Ada defines no locking or safety measures for simultaneous access to files.

4.6. Arithmetic

Universal types [LRM Chapter 3] are a canonical form for representing all the possible arithmetic values. The arithmetic of embedded targets is bound by the precision of the hardware. Compilers are expected to do exact arithmetic at compile time on static universal expressions [LRM 4.10.4] such as numeric literals. The requirement for the accuracy of operations with real operands at runtime is defined in LRM 4.5.7. However, in practice, an implementation will probably provide all the accuracy that the underlying hardware allows. The runtime cost for arithmetic expression evaluation includes computation and constraint checking. Arithmetic is very implementation and machine dependent.

An integer type [LRM 3.5.4] has a set of values within a specified range. Real types [LRM 3.5.6] provide approximations to real numbers. Floating point types have relative bounds on errors, whereas fixed point types have absolute bounds on errors. Error bounds on the predefined operations are given in terms of the model numbers. Fixed point types have an error bound specified as an absolute value, known as the delta of the type.

SHORT_INTEGER, SHORT_FLOAT, and LONG_INTEGER/FLOAT are types which, if implemented, are designated by the compiler designer to represent shorter and longer ranges, respectively, than INTEGER and FLOAT. The ART must implement the attribute operations such as SMALL and LARGE, and their representations must be defined by the compiler designer. Apart from the basic operations such as addition, the ART must implement attributes in LRM Sections 3.5.5, 3.5.8, and 3.5.10, such as T'WIDTH, T'MANTISSA, and T'DELTA. A compiler designer may provide new attributes. A compiler purchaser must make sure that the attributes are suitable for that machine. Validation cannot check such features.

The ART must raise an exception (numeric error or constraint error) for any arithmetic operation that cannot deliver a valid result, unless it is part of a larger expression that will subsequently deliver a valid result. Rules for determining a valid result are defined by the LRM. Universal expressions [LRM 4.10] must have an accuracy as good as that of the most accurate predefined floating point type supported by the implementation. The LRM implies that some error conditions, such as when the result of a real operation has a model interval that is undefined [ARTEWG 86d], may go undetected.

Ada makes use of *model numbers* and *safe numbers* [LRM 3.5.6] to describe the accuracy of real numbers. Safe numbers are an implementation-defined set providing guaranteed error bounds for operations on an implementation-dependent range of numbers. An implementation must include at least the model numbers and represent them exactly. Implicit conversions, and some explicit conversions, can result in an implementation-defined value. For example, the compiler designer chooses how real numbers, such as 0.5, are rounded to integers (i.e., up or down).

The accuracy and range of data types must be completely defined by the compiler designer. Different accuracies (depending upon the various data types) can be used. An embedded system designer will need a good understanding of how accurate the arithmetic can be.

The LRM recognizes that various targets may not be able to detect overflow situations. The attribute MACHINE_OVERFLOWS indicates whether the target will raise the exception. The LRM does not define the actions when the result of a real operation (where MACHINE_OVERFLOWS is false) is not in the range of safe numbers [ARTEWG 86d].

4.7. Exception Management

An *exception* is an error or an exceptional situation which occurs during program execution. It can result from a range or domain error for an operation. The handler is a piece of code which is executed when the exception occurs. It represents a transfer of control from the normal flow of control within the program. An embedded system needs to cater to exceptional situations and recover from them or be fault tolerant enough operate at a certain level of reliability in the face of faults.

The ART is required to detect the exception and search for a handler. The compiler designer chooses the following: the implementation strategy for handlers; how to propagate exceptions; whether to report an exception or lack of a handler beyond the scope of the main program; what strategy to use for setting the status of the program if no handler is found; how to implement nested exception handlers; and how to handle an exception during elaboration of the runtime system itself (for example, during library elaboration).

The ART must cater to direct and indirect exceptions. Not only must the ART instigate an exception indirectly caused by an operation such as overflow, but also a direct one requested by the programmer via the raise statement. The ART must allow exceptions to be propagated implicitly (i.e., to an outer scope if no handler exists within the immediate scope) or explicitly (i.e., via a reraise with the raise statement within a handler). Rules exist concerning the range of an exception's propagation. For instance, during a rendezvous, an exception in the server task will propagate to the client task if it is unhandled during the rendezvous or if the served task aborts, but not vice versa. The ART maps any hardware faults (related to operations) to the Ada software exception-raising mechanism. It also prepares the status of the task's stack to conform to that which the exception manager can use.

Language-defined exceptions [LRM 11.1] that the ART must implement are: CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, TASKING_ERROR, and the "catch-all" exception choice, OTHERS. This last one applies to anonymous exceptions—that is, all other possible exceptions for which no handler exists in the current scope.

Ada semantics for exception handling require the ART to implement guarded regions as defined by LRM 11.4. These are the scope of a handler within the Ada program. The ART must detect that an exception has occurred, find the handler, and change the execution context from the normal flow of code to that of the handler. If no handler exists for that region, it must propagate the exception to the outer region in order to continue its search for a handler; before that current region/scope can be left, tidy-up actions must be performed, such as deallocating storage and waiting for tasks to terminate. (The propagation can be suspended due to preemptive scheduling; hence, mutual exclusion safeguards should be enforced.) Once a handler is found, its code can be executed and the task can continue at a point from within the handler.

The LRM requires no particular implementation strategy such as exception map or a stack unwind mechanism [Baker 86]. Each has its own overheads. Some performance issues are related to the overhead if a code sequence has an exception handler associated with it yet has no exceptions raised during execution of that code [ARTEWG 86d]. Other performance issues related to the overhead associated with the raise statement and finding the handler. There is no requirement for an ART implementation to notify the programmer when an unhandled exception has propagated to its limit, for example, when a task completes or the main program's execution is abandoned. Nor is the status of the program defined by language semantics when the latter happens. The representation of unique identifiers for exceptions can have an impact on the speed and size of the generated code [ARTEWG 86d]. The raising of NUMERIC_ERROR, supported as hardware, could be more efficient than the compiler generating code after every arithmetic operation [ARTEWG 86d]. The designer can implement additional exceptions.

One of the major runtime overheads for Ada concepts is the constraint checking which the ART is required to perform. The LRM 11.6 suggests that certain compiler optimizations can be performed, thus eliminating some exception raising situations, particularly constraint checks.

Pragma SUPPRESS should cause the compiler to omit the corresponding exception checking that would occur at runtime. The LRM does not define what happens when a execution error occurs and the check has been eliminated. The ART could ignore this pragma, though, and raise the exception anyway. Using pragma SUPPRESS involves a considerable risk factor for a programmer, but eliminating the checks can significantly reduce object code size and improve execution time.

4.8. Pragmas

There are 14 language-defined pragmas [LRM Annex B]:

- CONTROLLED
- ELABORATE
- INLINE
- INTERFACE
- LIST
- MEMORY_SIZE
- OPTIMIZE
- PACK
- PAGE
- PRIORITY
- SHARED
- STORAGE_UNIT
- SUPPRESS
- SYSTEM_NAME

Their main purpose is to select particular runtime features of the language or to override the compiler's default. The compiler designer can choose whether the compiler implements any of the pragmas and whether it gives a warning to the programmer that it has ignored the pragma. The designer can also define new pragmas.

Pragmas ELABORATE, LIST, SYSTEM_NAME, and PAGE have their primary effect before runtime. Pragma OPTIMIZE requires the compiler to make time versus space efficiencies. This implies that the

ART provides the capabilities to offer different algorithms based on runtime costs. Criteria need to be defined by the ART as to how such tradeoffs can be determined and implemented. Pragmas CONTROLLED, MEMORY_SIZE, and PACK relate to storage management and are discussed in Sections 4.1.2, 4.1.3, and 4.1.1, respectively. Pragma STORAGE_UNIT is a compile-time feature which replaces the value given in package SYSTEM for representing the size of a storage unit. Pragma INLINE and INTERFACE relate to subprogram management and are discussed in Section 4.4. Note that the LRM does not define the circumstances under which subprograms are expanded. For instance, the implementation may place restrictions on inline expansion of a subprogram body compiled in another compilation unit [ARTEWG 86d]. Pragma SUPPRESS relates to exception handling and is discussed in Section 4.7. Pragmas PRIORITY and SHARED are discussed in Sections 4.2.2 and 4.2.4.

The semantics of the 14 language-defined pragmas are broad enough for any implementation to choose its own approach. More pragmas can be defined by the compiler designer (for example, a pragma to invoke overlays) as long as they are documented. As with all pragmas, a compiler may ignore the programmer's request without notifying the programmer.

4.9. Chapter 13 Features

Chapter 13 of the LRM contains specific implementation-dependent issues which were designed into the Ada language, such as storage mappings, association of entities with hardware, implementation-defined package SYSTEM, attributes, machine code insertions, interface to other languages, and facilities for removing checks. A compiler need not provide any of these except for package SYSTEM. If it does, the compiler designer must define the semantics of these features.

4.9.1. Clauses

Length clauses [LRM 13.2] and record representation clauses [LRM 13.4] are discussed in Section 4.2.3, along with pragma PACK. Address clauses [LRM 13.5] are discussed in Section 4.1.4. The compiler designer can specify new attributes.

Currently, an ART implementation can limit its acceptance of representation clauses to those that can be handled simply by the underlying hardware. Except for address clauses, the compiler and ART must guarantee that the net effect of the program is not changed by the presence of clauses, either for parts of the program that interrogate representation attributes, or for length clauses of fixed point types. The compiler designer determines the interpretation of the expression that appears in the representation clauses. Similarly, it defines how literals and aggregates are represented in the execution program. This can affect space tradeoffs [ARTEWG 86d]. Use of enumeration representation clauses [LRM 13.3] may lead to inefficient implementation. Restrictions on record representation clauses are left to the implementation to be defined. The LRM defines no ordering of bits, nor does it determine whether a component can overlap storage units.

4.9.2. Package System

Every compiler must define a library package, SYSTEM, which includes the definitions of any target-dependent characteristics. A minimum set of features must include those such as types ADDRESS and NAME, and constants SYSTEM_NAME, STORAGE_UNIT, MEMORY_SIZE, MIN_INT, MAX_INT, MAX_DIGITS, FINE_DELTA, and TICK [LRM 13.7]. The ART may provide capabilities for the programmer to alter some of these features via pragma SYSTEM_NAME. The compiler designer is free to choose its target's most suitable values for the runtime constants and variables [LRM 13.7] of package SYSTEM.

4.9.3. Machine Code Insertion

Ada provides the ability for a programmer to insert assembly code statements directly into compiled Ada code [LRM 13.8(3)] and defines a number of limitations on the constructs permitted in the body of a code procedure. Further limitations can be defined by the compiler designer. The LRM may impose further restrictions on the record aggregates (which represent the machine instructions) than those which Ada normally requires for record aggregates. Programmers will most likely need some knowledge of the ART system in order to write assembly code inserts.

An implementation, if it supports machine code inserts, is free to provide pragmas for specifying register-calling conventions regarding machine code inserts. It can also decide what assembly statements are permitted within the inserts along with determining how the machine features are interfaced.

There is no LRM requirement for the ART to provide any safety regarding machine code inserts [LRM 13.8]. That is, an Ada compiler could perform optimizations across any programmer's assembly code inserts without the programmer's knowledge. In effect, the programmer may not have the control expected.

4.9.4. Interfacing Other Languages

Subprograms written in other languages can be called from an Ada program. Pragma INTERFACE, which indicates this, is discussed in Section 4.4. The compiler designer must define any calling conventions, parameter passing, exception handling, storage, and tasking ramifications.

Pragma INTERFACE, if provided, requires the ART to define the semantics of the conventions of interfacing. This is a tricky issue since the other language may not have similar runtime semantics or concepts such as exception handling, tasking, dynamic storage (de)allocation. For example, how should the ART recognize errors occurring in the non-Ada code, and does the non-Ada code need to know anything about the status of an Ada caller?

4.9.5. Eliminating Checks

Library subprograms, UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION, permit a way of bypassing Ada's checking conventions. UNCHECKED_DEALLOCATION is discussed in Section 4.1.2. It is entirely up to the programmer to ensure that there will not be subsequent use of the same item by another access variable, which could have dangerous and unpredictable results. UNCHECKED_CONVERSION coerces a value into that of another type without changing the bit pattern. This permits a way of bypassing the compiler's strong typing features but places the onus on the programmer to provide the type checking that the ART would have done. The LRM does not define whether the objects need to be of the same size for conversion.

4.10. Conclusion

This chapter has highlighted many of the compiler and runtime options that compiler builders have when developing an Ada compiler. Any compiler buyer, in particular an embedded systems designer, will need a thorough understanding of the ramifications, in terms of functionality and performance, of the options provided by the compiler. Apart from optional features of compilers such as those of Chapter 13 and pragmas, the storage, tasking, and exception management will provide runtime overheads in an embedded system.

Ada does have a complex runtime system as far as embedded systems developers are concerned. The *Ada Language Reference Manual* requires the compiler designer to document any of the implementation-defined features given in its Appendix F, such as:

- effect of pragmas
- name and type of attributes
- specification for package SYSTEM
- restrictions on representation clauses
- naming conventions
- interpretation of address clauses
- restrictions on conversions and characteristics of input/output packages

Unfortunately, many implementation-dependent aspects could go undocumented.

It is clear that an Ada runtime system implementation involves a complex interface between the compiler and the ART routines. Unlike other languages, such as C, a programmer cannot directly use ART routines without a clear understanding of the runtime system. It would be desirable to have a separation between the ART routines and code that the compiler generates. ARTEWG [ARTEWG 86c] recognizes the need for runtime systems to be interchangeable. This would be quite a complex design since the dividing line is so fine between the compiler and runtime system. Also, it may be possible for compiler vendors to build a generic ART that enables compiler buyers to completely tailor their ART, as suggested by SofTech [Grover 83]. But it is expected that such generic overheads would not be suitable for real-time performance requirements. Another important issue is when these options are bound, that is, at runtime, compile time, or load module generation time [Lomuto 83]. It may be that not every embedded system requires all the facilities of Ada since an embedded system generally makes tradeoffs for performance over functionality. Similarly, it may be that different processors within the embedded system use only certain parts of the ART and may not need, for instance, the tasking manager.

Due to embedded systems requirements, Ada compiler buyers prefer a customized ART [Bamberger 86, Grover 85a, Hood 86, Pepper 86, Rodriguez 86]. Tradeoffs need to be considered whenever options and tailoring are desired. ARTEWG [ARTEWG 86d] is developing a document which discusses all of the implementation-dependent issues in which compiler designers are interested. (The major issues were included in this chapter.) A compiler buyer must select an Ada compiler and runtime system with care. The demarcation between the compiler buyer's and compiler designer's responsibilities may be fuzzy. Buyers are starting to ask, "How much will it cost to meet performance requirements using such features?" [Lomuto 82].

For a compiler buyer, it is prudent to conduct test measurements by building tools and defining metrics before purchasing, to ascertain any restrictions and timing limitations of Ada runtime environments [Ruane 85]. A designer of a real-time system must understand the Ada runtime system along with the real-time system. Benchmarks are being used, such as those of Clapp [Clapp 86] and MITRE [Ruane 85] for determining some performance factors. Programming idioms can reduce runtime overheads, but this is not enough to significantly reduce performance problems. No single runtime environment can be satisfactory for every user [Grover 83]. Once performance overheads are known, policies for Ada programmers can be enforced. These policies can describe the best usage of features for giving the most suitable performance at execution time.

ARTEWG is providing assistance in increasing awareness of the problems of implementation-defined options by documenting many of the runtime issues. "The current generation of Ada runtime systems is sufficient to pass validation; but it lacks additional runtime capabilities, such as speed and compactness, needed for real-time applications" [ARTEWG 87b]. Customizations do not really affect the validation concept since a tailored compiler can pass validation—the compiler must include only the minimal facilities to pass validation. However, customized portions of the compiler cannot be validated. Validation exists to stop the proliferation of nonstandard Ada compilers, but differing compilers do result from customizations. ARTEWG is attempting to define a common runtime model and a set of features that Ada compiler implementors would implement.

The first generation of Ada runtime systems for embedded targets is not sufficiently compact, tailorable, or efficient for those embedded real-time applications that operate under stringent memory and throughput constraints. Ada implementors are compelled to make the execution function decisions that application developers previously made on their own when building the execution environments from scratch [ARTEWG 87b].

5. Ada Embedded System Development Environment

Inherent in an environment for embedded system development is the notion of distinct host and target machines. Typically, embedded software systems are developed and tested to the extent possible on the host and then downloaded and integrated with the target hardware for system-level testing and eventual deployment. Prior to acquiring any development tools, one is faced with the problems of deciding what tools are needed and ultimately selecting the vendor from which to purchase those tools. The requirements depend on a particular project's needs, whereas the selection involves systematically evaluating the quality of the applicable products currently available in the marketplace. This systematic evaluation can be conducted using the approach developed in *Evaluation of Ada Environments* [Weiderman 87]. The purpose of this chapter is to characterize the minimal functionality requirements for the support tools essential (from an real-time application developer's viewpoint) to developing and testing Ada embedded systems. Since there is an inherent host-target scenario when developing and testing embedded software systems, this chapter will be divided into two sections: target-independent tools and target-dependent tools.

5.1. Target-Independent Tools

In developing programs for real-time embedded systems, program design, development, maintenance, and management are performed on a host machine. Typically, automated support of these life-cycle activities is provided in the form of target-independent tools running on the host machine. This section characterizes some of the more commonly used target-independent tools used in real-time application development.

5.1.1. Pretty Printer

To ensure coding conventions (i.e. indentation, spacing, paging) throughout a project, a pretty printer is usually employed. This tool formats a file with standard, predefined coding conventions in preparation for printing.

A pretty printer should have a facility for changing the predefined conventions since not all projects use the same conventions. This tool should also be easily integrated with a language-sensitive editor where syntactic (and possibly semantic) checking can be performed as formating takes place.

5.1.2. Language-Sensitive Editor

In conjunction with a common text editor, a language-sensitive editor can be employed to facilitate code entry. A language-sensitive editor consists of source code templates that are usually invoked with short control commands. Use of this tool relieves some of the programmer's typing demands and may also eliminate syntax errors.

A language-sensitive editor should have facilities to support compilation, review of diagnostics, and interaction with a symbolic debugger. The degree of user friendliness, including a lucid command structure and a help facility, is an important consideration.

5.1.3. Static Analyzer

Characteristics of a compiled program are provided by this tool. Information (i.e., type, scope, size) regarding symbols used in the program may be given. Also, information about the code may be provided, such as total lines of code and the number of comments. Capabilities may be provided to give the user a measure of subprogram complexity and a list of unreachable statements. Compilers may provide some static analyzing functions, thus possibly eliminating the need for such a tool.

5.1.4. Source Code Cross-Referencer

This tool also describes attributes of a compiled program. A source code cross-referencer gives a cross-reference table or map of the symbol definition sites and their use sites. The option to have only a section of the code processed may be helpful. Certain compilers may also generate cross-reference information, thus possibly eliminating the need for this tool.

5.1.5. Test Manager

A test manager helps the user to organize and execute software tests. Some test managers may provide the capability to compare test results (i.e., if a module has been changed, how do the new test results differ from the previous results). This tool automates the software testing phase in the system development process.

When looking at this tool, some of the testing capabilities to consider are creating and debugging a test harness, developing a test plan and associated test data, and performing initial unit testing. Regression testing, if available, is also a useful capability.

5.1.6. Configuration Manager

This tool is used to clearly identify the components of a software configuration. Software changes are controlled, and a record is maintained of how a software system evolves and what its status is at any given time.

A good configuration manager can be a helpful tool. Some of the functions that should be provided by this tool are: create/modify a system element, build a system baseline, build a current system with variant version elements, and reserve/replace/delete a particular system element. One should also consider what information is recorded in the history log. In addition, a facility for system/element version comparison should be provided.

5.1.7. Module Manager

A module manager is used to build and test software modules; it may also be able to determine which modules in a system have been changed and which other modules are affected by the changes (and be able to update those modules appropriately).

This tool should provide capabilities to create, modify, and maintain modules, keep history records on modules (including module dependencies), perform automatic updates to dependent modules (when necessary), and perform automatic builds of a system. A configuration manager used in conjunction with this tool should provide all configuration management functions.

5.1.8. Browser

When trying to understand the complexities of large Ada programs, it is important to be able to browse rapidly through a program library. For example, when an object is defined in another unit, it is useful to find its definition. It may also be useful to find dependent units or all uses of a defined object. These and similar functions are important in an Ada programming environment.

5.2. Target-Dependent Tools

There comes a point in the software development life cycle when an application must be downloaded onto the target machine for hardware/software integration and system level testing. The normal downloading process involves:

1. Translating the source (Ada or assembler) code into target object code.
2. Linking the resultant target object code.
3. Building an executable load module from the linked object code.
4. Downloading the executable load module to the target machine.
5. Executing and debugging the application on the target machine.

Typically, automated support of this process is provided in the form of target-dependent tools running on the host machine. This section characterizes some of the more commonly used target-dependent tools as they pertain to this process.

5.2.1. Ada Cross-Compiler

The primary functional requirement for an Ada cross-compiler is translation of Ada source code into either target assembly or target object code. The overall performance of an Ada cross-compiler can be evaluated using existing benchmark suites [Clapp 86, Hook 85]. Functionally, an Ada cross-compiler should support the optional features listed in Chapter 13 of the LRM: representation clauses, length clauses, enumeration representation clauses, record representation clauses, address clauses (interrupts), change of representation, interface to other languages (e.g., assembler, FORTRAN), and unchecked type conversions. It also should cater to real-time timing requirements (e.g., DURATION'SMALL <= 100 microseconds and SYSTEM'TICK <= 1 milliseconds) and offer the real-time application developer the ability to select (preferably via a pragma) the scheduling paradigm to employ.

5.2.2. Cross-Assembler

The primary functional requirement for a cross-assembler is translation of target assembly into target object code. This object code must be suitable for linking with the object code produced by the Ada cross-compiler. The integration of the cross-assembler with the compiler is highly desirable.

5.2.3. Linker

The linker produces a load module from one or more independently translated object modules by resolving cross-references among those object modules. Depending on the nature of the target's execution environment, the load module produced by the linker may be suitable for immediate downloading to and execution on the target machine. However, if runtime routines, target-specific services, or a target kernel also need to be linked with the application object module(s), the load module produced by the linker is not suitable for execution on the target machine. Typically, another step involving building a system load module is necessary.

The linker should support a comprehensive mechanism (e.g., a map file containing an object module synopsis, a module relocatable reference synopsis, and a list of symbols by name) for summarizing the load modules that it creates. The linker should also generate a symbol table file containing the load module's global symbols and make those symbols available to the debugger and dynamic analyzer at program runtime. It should also support selective linking of the object code modules for runtime routines and target-specific services, as well as the ability to specify (in a text file) the names of all object code modules produced by the cross-assembler that are to be linked together with the Ada code.

5.2.4. System Builder

The system builder tool is normally provided in either of the following cases:

1. The target's execution environment has a small operating system or a real-time kernel that must be linked with the application object module(s).
2. More control over the mapping of code into target memory (e.g., loaded into ROM) is required.

A system builder utility must potentially combine one or more application load modules, a kernel image, and any necessary runtime or target-dependent service images into a single executable system load module suitable for downloading to and execution on the target machine. As with the linker, the system builder facility should provide a comprehensive mechanism (e.g., map file) for summarizing the system load modules that it creates; and it should also provide selective linking of the object code modules for runtime routines and target-specific services.

5.2.5. Load Module Downloader/Receiver

In order for an application to be executed on the target machine, it must first be downloaded into the target's main memory. Normally there is a download/receive tool (running on the host, target, or both machines) for loading an application's system load module into target memory across a communications line or through another medium. It must also support the process of triggering the execution (booting) of the application code on the target machine.

5.2.6. Symbolic, Source-Level Debugger

A target-dependent, symbolic, Ada source-level debugger provides a mechanism (via minimally single-stepping execution) for testing and detecting errors in an Ada application executing on the target machine.

The source-level debugger should support facilities including, but not limited to, setting/resetting breakpoints and tracepoints, single-stepping program execution in increments of single hardware instructions, setting breakpoints on exceptions, setting breakpoints on task context switches, controlling program execution path (e.g., execute 10 statements, enter a specified subprogram), querying the program's execution state (e.g., examine values of program variables, display runtime stack), and modifying the program's execution state (e.g., modify variable values). It should also support both a local and remote mode of operation and offer target-dependent facilities for monitoring and controlling the resources of and software running on the target machine.

5.2.7. Dynamic Analyzer

A dynamic analyzer is a tool that aids in the tuning evaluation of an application's performance by monitoring its execution behavior. A dynamic analyzer should collect and analyze application performance data such as: program counter sampling, control path coverage, statement execution frequency and timing, input/output statistics, and system service calls. It should provide the capability of analyzing this performance data and reporting it in an effective manner (e.g., histograms, tables).

5.2.8. Simulator

A target simulator is a computer program that simulates the behavior of the target machine by representing its physical characteristics. Target simulators are used in lieu of the actual hardware for testing purposes and thus simulate the execution of system load modules suitable for the target machine.

A target simulator facility must accurately emulate both the functional and temporal behavior of the target's instruction set architecture. It should provide access to all memory locations and registers. Furthermore, it should support typical features found in a symbolic debugger (e.g., single-stepping instruction execution, examination of variable values) augmented by the capability to perform timing analysis (e.g., how much time elapsed when executing the last 10 instructions). It should support simulated input/output interaction by providing access to I/O ports, device control and data registers, and emulation of the architecture's interrupt mechanism. Finally, it should facilitate the set-up and reuse of test sessions by allowing freezing of the current session's context, executing debugger commands from script files, and supplying I/O data from existing data files.

5.2.9. Real-Time Monitor

In order to monitor the target system in a non-intrusive way, it is useful to have hardware or a combination of hardware and software to help determine whether deadlines are being met and whether activities are being performed in the proper sequence. Such a device may also be used to monitor and timestamp message traffic within a system.

5.3. Summary

The purpose of this chapter was to identify and functionally characterize programming support tools that are, from the application developer's viewpoint, essential to the development and testing of Ada embedded software systems. The list of tools presented in this chapter is in no way comprehensive but is intended as a enumeration of some of the more commonly used tools in a host-target development environment. A list summarizing the typical functions and features of these commonly used tools follows.

Target-Independent Tools

Pretty Printer

- Reformats Ada code relative to a set of coding style conventions
- Supports a method for changing coding style conventions
- Operates in both interactive or batch mode
- Is integrated with text editor for interactive use

Language-Sensitive Editor

- Source code templates
- Language construct expansion
- Keyword abbreviation
- Integration with Ada compiler (syntax checking, compiling)
- Integration with debugger
- Facility for reviewing syntax errors in code
- Online help facility

Static Analyzer

- Data flow analysis
- Path analysis
- Interface analysis
- Call graph analysis
- Dependency analysis
- Code style analysis (LOC, number of comments, number of *if* statements)

Source Code Cross-Referencer

- Generates listing for each definition of program symbols
- Generates listing for each use of program symbols

Test Manager

- Creates test harness
- Creates test input data
- Performs initial test
    - creates expected output data
    - produces actual output data
    - compares actual and expected data
- Performs regression testing

Configuration Manager

- Has create/delete element
- Creates new version of existing element (3 classes of versions)
    - successive (e.g., bug fix)

    - parallel (e.g., implementation for different target)
    - derived (e.g., optimized module)

- Merges variants of an element
- Retrieves specific version of an element
    - explicit (e.g., use version 4)
    - dynamic (e.g., use most recent version)
    - referential (e.g., use same version as used in Rev 4.0)

- Compares different versions of an element
- Maintains/displays history attribute of an element

Module Manager

- Defines system model
    - source dependencies
    - translation rules
    - translation options
    - tools necessary for translation

- Builds system
    - current default
    - specific earlier release (rebuild)
    - hybrid (mixture of default versions and specific versions)

- Maintains/displays list of constituents of a built system
- Maintains/displays system build history
    - date built
    - name of builder
    - reason for building
    - options employed

- Baselines system as a product release
- Maintains/displays product release information
    - number of distributed versions
    - differences among versions
    - locations of each version
    - required hardware for each version
    - correlation between versions and error reports
    - correlation between versions and components
    - errors reported/fixed by version

- Maintains/displays system release history
    - What was built, when, why, and by whom

- Reverts back to previous release environment using old binaries, source, and dependent modules

- Automatically deletes unused binaries

Browser

- Finds an object's definition
- Inspects a body from a specification
- Inspects a *withed* package
- Inspects a called subprogram
- Finds all uses of an object

Target-Dependent Tools

Ada Cross-Compiler

- Translate Ada source code into either target assembly or target object code
- Desirable implementation features
    - prints warning message for unrecognized pragmas [LRM, Section 2.8]
    - provides short and long integers [LRM, Subsection 3.5.4]
    - provides short and long reals [LRM, Subsection 3.5.7]
    - reclaims storage automatically when object becomes inaccessible [LRM, Section 4.8]
    - supports pragma INLINE (The compiler should detect and flag any situations where the pragma cannot be followed, e.g., recursive subprograms.) [LRM, Subsection 6.3.2]
    - supports pragmas SUPPRESS, ELABORATE, LIST, and PAGE
    - has minimum ranges of 0...15 for pre-defined type PRIORITY [LRM, Section 9.8]
    - provides low-level I/O packages to support real-time device drivers [LRM, Section 14.6]
- Support for majority of Chapter 13 features
    - required features
        - representation clauses [LRM, Section 13.1]
        - enumeration representation clauses [LRM, Section 13.3]
        - record representation clauses [LRM, Section 13.4]
        - address clauses (interrupts) [LRM, Section 13.5]
        - change of representation [LRM, Section 13.6]
        - interface to other languages (Assembler, HOL) [LRM, Section 13.9]
        - unchecked type conversions [LRM, Section 13.10.2]
    - desirable features
        - length clause [LRM, Section 13.2]
        - unchecked storage deallocation [LRM, Section 13.10.1]
- Comprehensive documentation
- Informative diagnostic (error) messages
- Clearly documented restrictions
- Clearly documented implementation-dependent characteristics (Appendix F)

    - the form, allowed places, and effect of every implementation-dependent pragma [LRM, Section 2.8]
    - the name and type of every implementation-dependent attribute [LRM, Section 4.1.4]
    - the specification of the package SYSTEM [LRM, Section 13.7]
    - a list of all restrictions on representation clauses [LRM, Section 13.1]
    - the conventions used for any implementation-generated names denoting implementation-dependent components [LRM, Section 13.4]
    - the interpretation of expressions that appear in address clauses, including those for interrupts [LRM, Section 13.5]
    - any restrictions on unchecked conversions [LRM, Section 13.10.2]
    - any implementation-dependent characteristics of the input/output packages [LRM, Sections 14.1, 14.2.1, 14.4]

- Clearly documented known bugs
- Ability to produce at the user's option
    - source listing with line numbers
    - cross-reference listing
    - variable map
    - assembly listing of generated object code containing references to source line level and source procedure
    - compilation summary including
        - date and time of compilation
        - compilation options in effect
        - size of the generated object
        - count of source lines
        - name of the object file created
        - presence of implementor-defined pragmas
        - names of packages referenced
        - compiler version number

- Ability to cater to real-time timing requirements
    - DURATION'SMALL <= 100 microsecond
    - SYSTEM'TICK <= 1 millisecond
    - Selection (preferably via a pragma) by the real-time application developer of the scheduling paradigm to employ from various options

Cross-Assembler

- Translates target assembly code into target object code
- Generates assembly language listings
- Integrates with compiler

Linker

- Produces a load module from one or more independently translated object modules by resolving cross-references among those object modules

- Generates a map file containing an object module synopsis, a module relocatable reference synopsis, and a list of symbols by name for summarizing the load modules
- Generates a symbol table file containing the load module's global symbols and makes those symbols available to the debugger and dynamic analyzer at program runtime
- Supports selective linking of the object code modules
- Supports multiple languages

System Builder

- Potentially combines one or more application load modules, a kernel image, and any necessary runtime or target-dependent service images into a single, executable, system load module suitable for downloading to and execution on the target machine
- Generates a map file containing an system load module synopsis, a module relocatable reference synopsis, and a list of symbols by name for summarizing the load modules
- Generates a symbol table file containing the load module's global symbols and makes those symbols available to the debugger and dynamic analyzer at program runtime
- Supports selective linking of the object code modules
- Configures kernel for different hardware configurations
- Supports locating code in different parts of memory space

Load Module Downloader/Receiver

- Loads an application's system load module into target memory across a communications line or through another medium
- Controls application execution
    - triggering
    - halting
    - suspending
    - resuming

Symbolic, Source-Level Debugger

- Sets/resets breakpoints and tracepoints
- Single-steps program execution in increments of single hardware instructions
- Sets breakpoints on exceptions
- Sets breakpoints on task context switches
- Controls program execution path (e.g., executes 10 statements, enters a specified subprogram)
- Queries the program's execution state (e.g., examines values of program variables, displays runtime stack)
- Modifies the program's execution state (e.g., modifies variable values)
- Supports both a local and remote mode of operation
- Offers target-dependent facilities for monitoring and controlling the resources and software running on the target machine

Dynamic Analyzer

- Collects and analyzes application performance data
    - program counter sampling
    - control path coverage statement execution

  - frequency and timing
  - input/output statistics
  - system service calls

- Provides the capability of analyzing application performance data and reporting it in an effective manner
  - histograms
  - tables

Simulator

- Accurately emulates both the functional and temporal behavior of the target's instruction set architecture
- Provides access to all memory locations and registers
- Supports typical features found in a symbolic debugger
  - single-step instruction execution
  - examines variable values
  - start/stop program execution

- Performs timing analysis (e.g., how much time elapsed when executing the last 10 instructions)
- Supports simulated input/output interaction
  - provides access to I/O ports
  - provides access to device control and data registers
  - emulates the architecture's interrupt mechanism

- Facilitates the set-up and reuse of test sessions
  - freezes the current session's context
  - executes debugger commands from script files
  - supplies I/O data from existing data files

Real-Time Monitor

- Checks deadlines
- Checks sequencing
- Checks and timestamps message traffic

6. Summary and Preliminary Recommendations

This chapter categorizes and summarizes the various kinds of issues which accompany the use of Ada in embedded systems. Whereas Chapters 3, 4, and 5 deal with the details of the issues, Chapter 6 attempts to identify and clarify the global issues. In this chapter we also make some preliminary global recommendations on the basis of our reading and initial investigation. What has become clear is that there must be cooperation and communication among the various groups concerned with Ada in order to make Ada a viable option for embedded systems. If any of the groups (applications developers, program managers, implementors, or policy makers) act independently, suboptimal solutions will result.

Chapter 2 establishes the requirements of the embedded systems problem domain, which are quite different from those of traditional information systems. These systems monitor and control their environment, exhibit logical and physical parallelism, have severe timing and resource constraints, and require special tools for software development. It is clear from these requirements that implementors cannot provide their products without considering the unique requirements that face the application developers.

Chapter 3 deals with language issues particularly pertinent to embedded systems, such as tasking, inter-task communication, time control, input/output, internal representation, error handling, and numerical computation. Since there are so many features in the Ada language, there are many options for programming in Ada. Programs can be written in the style of FORTRAN, but this is clearly not the best way to exploit the software engineering benefits of Ada. At the other extreme, if the full power of Ada is used, there may be performance penalties. The tradeoff is to make maximum use of the functionality provided while still meeting performance requirements. Thus the application developer must have a "bag of tricks," or programming idioms, which provide the optimal solution for the problem at hand.

Chapter 4 deals with implementation issues. Here, because of the latitude provided by the Ada reference manual, the implementor must decide how to provide capabilities. Because of the economic pressures of validation, as well as the difficulties of implementing the more novel features of Ada, the implementors have often chosen the simple solutions rather than those which make the most sense for embedded applications. As a result we find few compilers that do garbage collection, for example. Issues that are critical to developers, such as scheduling algorithms, timing control, and interrupt and exception handling, have been solved in a variety of ways. Until recently, the problems of the Ada runtime environment have received little attention compared to that given the syntax and semantics of the language.

It must also be noted that implementors are governed not only by the LRM, but also by interpretations recommended by the Language Maintenance Panel and approved by the Ada Board. These so-called Ada Issues (AIs) are identified by a sequential numbering system and resolve questions and ambiguities in the language. They now number in the hundreds and, when finally approved, are binding on implementors.

Chapter 5 gives a brief overview of the tools necessary to support a host-target development system. These tools were enumerated and described in general terms. Future activity of the AEST Project must concentrate on evaluation criteria for these tools so that their state of readiness for software development can be determined.

6.1. Preliminary Recommendations

It is too soon to make definitive recommendations on a wide variety of issues concerning the use of Ada in embedded systems. Only experimentation and experience with the Ada tools becoming available will permit us to draw meaningful conclusions about the state of the technology. However, based on a literature review, a review of DoD Ada policy, and an evaluation of the product literature available as of December 1986, it is possible to make a few preliminary recommendations. These preliminary recommendations will be expanded as more experience is gained with the testbed and as more experiments are conducted.

6.1.1. Recommendations for Application Developers

- Application developers need to understand the various programming paradigms in which particular language features can be used.
- Application developers need to have a detailed knowledge of the programming alternatives and corresponding tradeoffs associated with using or not using particular Ada features.
- Application developers must understand the machine dependencies and the features dependent on their particular implementation of the Ada language.
- Programmers must understand how the runtime system works so that they can evaluate predictability and performance and bolster their confidence in controlling the underlying resources.
- Programmers must understand the tradeoffs between absolute control of every resource in an embedded system and good software engineering practices that promote maintainability and reuse.
- Application developers must recognize that design decisions may be governed by performance and features of the particular Ada implementation they choose.

6.1.2. Recommendations for Program Managers

- Program managers should be aware of the metrics and test suites available and should be able to apply them and evaluate the results before procurement decisions are made for Ada implementations.
- Program managers must become more knowledgeable about the nature and impact of Ada runtime environments. They must be aware that many functions previously under control of their application programmers will now be under control of the compiler implementor.
- If Ada is mandated for a particular embedded system project, cost and schedule risks must be carefully evaluated because of the immaturity of the technology.
- Funding of Ada tools and, particularly, Ada runtime systems should be expected for several more years.
- Validation of an Ada compiler should be only the first of many steps in the evaluation of a compiler and its associated tools. Program managers must recognize that performance and usability are not covered by validation.

- Program managers should generate detailed requirements for development environment tools specific to the application, then evaluate and select the tools based on these requirements.

6.1.3. Recommendations for Compilation System Implementors

- All Ada compiler vendors should answer the questions posed in Ada-Europe's "Guidelines for Ada Compiler Specification and Selection," by Nissen and Wichmann [Nissen 82]. In general, they should provide extensive information about how the runtime system works.
- Vendors should provide the user with the capability to configure the Ada runtime system to suit the application program. They must foster a high degree of interaction with the customers.
- Vendors should recognize that in many time-critical applications, there is a need to sacrifice generality for efficiency. Vendors need to provide implementation-dependent pragmas for this purpose.
- Implementors must be sensitive to the requirements of the mission-critical application domain. In particular, they should identify metrics for performance and strive to achieve a minimum level of performance for those features that are important to the application.

6.1.4. Recommendations for Ada Policy Makers

- The current emphasis for the Ada program should be in the areas of performance and functionality. After there have been some successes in using Ada in embedded systems, the policy can place more emphasis on portability, reusability, and productivity.
- Instruction set architectures such as the MIL-STD 1750A must be evaluated with respect to their suitability for Ada; there may be mismatches between architectures and the language. Policies that promote newer architectures for Ada should be considered.
- More emphasis should be placed on the Ada runtime system as opposed to the syntax, semantics, and validation of the language. Little funding or attention has been paid to ongoing work, much of which is of high quality.
- Policy makers should address the implications of the proliferation of different runtime systems due to the implementation dependencies of the language.
- The ramifications of options for addressing language problems need to be explored. One option is to change the language, and the other is to add support packages that circumvent those problems. The tradeoffs include compatibility, portability, performance, and validation issues.

6.2. Future Work

This report suggests that there is a great deal of work to be done in investigating the use of Ada for embedded systems. Some of this work can take the form of introspection and study, but much of it requires experimentation in a laboratory setting. The Ada Embedded Systems Testbed will provide the vehicle for experimentation. We propose that three levels of experimentation take place. They are:

1. Stand-alone benchmarks and testing of language features.
2. Experiments using several features of the language which test commonly used algorithms and programming idioms.
3. The implementation of a complete embedded system application with real-time constraints.

6.2.1. Stand-Alone Tests

The University of Michigan, Performance Issues Working Group (PIWG), and the ACEC test suites are (overlapping) attempts at creating short Ada test programs to evaluate the performance of certain Ada language features. These programs provide timings for fine-grained features such as task creation, rendezvous, and termination; subroutine invocation times as a function of the number and type of parameters; and exception and interrupt handling. These tests should be executed on several target processors; and after evaluation of the coverage of these tests, new tests should be added to provide additional coverage. To date, little data is available on the performance of Ada on bare target machines.

6.2.2. Algorithmic Experiments

At the next level, it is necessary to conduct experiments using the algorithms and programming idioms typically found in embedded systems. These algorithms use several language features and test the interaction of these features. Algorithmic experiments address the choices which must be made by the application developer. For example, the system designer must decided early in the design process whether to use event-driven scheduling and tasking or a cyclic executive. Experiments at this level would provide insights into the tradeoffs resulting from these decisions.

6.2.3. Real Application

Finally, there needs to be a demonstration that Ada can handle a real-time application typical of many different applications in the mission-critical arena. An application needs to be complex enough to be credible to the MCCR community but small enough to be tractable for a small group to implement in six months or less. The application must handle real-time inputs with time constraints on the order of a small number of milliseconds. The testbed must faithfully simulate the actual environment in which the target processor will eventually reside. Once implemented, the application will serve as a gross test of the runtime environments of different compilers on different processors.

6.3. Conclusion

This report has attempted to raise many of the issues which must be addressed if the Ada program is to be successful. There are issues for the application developers, the managers of programs, compilation system implementors, and Ada policy makers. As the Ada Embedded Systems Testbed Project at the SEI progresses, these issues will come into greater focus. As more experiments are conducted on a variety of Ada tools and target processors, our recommendations will become more detailed and complete.

References

[Allworth 81] Allworth, S.T.
Introduction to Real-Time Software Design.
The Macmillian Press, Ltd., London, 1981.

[Ardo 83] Ardo, A.
Considerations for Full Ada Implementation on an Experimental Multiprocessor Computer.
Technical Report, University of Lund, December, 1983.

[ARTEWG 86a] Special Interest Group on Ada, Ada Runtime Environments Working Group.
A White Paper on Ada Runtime Environment Research and Development.
Technical Report Working Paper, ACM, November 13, 1986.

[ARTEWG 86b] Special Interest Group on Ada, Ada Runtime Environments Working Group.
A Canonical Model and Taxonomy of Ada Runtime Environments.
Technical Report Working Paper, ACM, November 13, 1986.

[ARTEWG 86c] Special Interest Group on Ada, Ada Runtime Environments Working Group.
A Catalog of Interface Features and Options for the Ada Runtime Environment.
Technical Report Working Paper, Release 1.1, ACM, November 23, 1986.

[ARTEWG 86d] Special Interest Group on Ada, Ada Runtime Environments Working Group.
Catalogue of Ada Runtime Implementation Dependencies.
Technical Report Working Paper, ACM, November 5, 1986.

[ARTEWG 86e] Special Interest Group on Ada, Ada Runtime Environments Working Group.
First Annual Survey of Mission Critical Application Requirements.
Technical Report Release 1.0, ACM, November, 1986.

[ARTEWG 87a] Special Interest Group on Ada, Ada Runtime Environments Working Group.
A Framework for Describing Ada Runtime Environments.
Ada Letters 7(5), September-October, 1987.

[ARTEWG 87b] Special Interest Group on Ada, Ada Runtime Environments Working Group.
The Challenge of Ada Runtime Environments.
Technical Report Working Paper, SIGAda, June, 1987.

[Baker 85] Baker, T.P.
An Ada Runtime System Interface.
Technical Report TR 85-06-05, Department of Computer Science, University of Washington, June, 1985.

[Baker 86] Baker, T.P. and Riccardi, G.A.
Implementing Ada Exceptions.
IEEE Software 3(5):42-51, September, 1986.

[Bamberger 86] Bamberger, J., Ritter, P., and Wilson, J.
Tactical Database Management System - An Ada Technology Project for the U.S. Army.
In *Fourth Annual National Conference on Ada Technology*, pages 132-141. U.S. Army Communications-Electronics Command, March, 1986.

[Barnes 84] Barnes, J.G.P.
Programming in Ada, 2nd Ed.
Addison-Wesley Publishers Ltd., 1984.

[Borger 86] Borger, Mark W.
Ada Task Sets: Building Blocks for Concurrent Software Systems.
In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*. Miami Beach, FL, April, 1986.

[Clapp 86] Clapp, R.M., Duchesneau, L., Volz, R.A., Mudge, T.N., and Schultze, T.
Toward Real-Time Performance Benchmarks for Ada.
Communications of the ACM 29, #8(RSD-TR-12-86), August, 1986.
Pages 760 - 778.

[Cornhill 87] Cornhill, D. and Sha, L.
Priority Inversion in Ada.
Ada Letters (7), November - December, 1987.
Pages 30-32.

[Downes 82] Downes, V.A. and Goldsack, S.J.
Programming Embedded Systems with Ada.
Prentice-Hall International Inc., 1982.

[E&V Team 84a] E&V Team.
Evaluation and Validation Plan.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, December, 1984.

[E&V Team 84b] E&V Team.
Requirements for Evaluation and Validation of Ada Programming Support Environments, Version 1.0.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, October, 1984.

[E&V Team 84c] E&V Team.
APSE Analysis Document.
Technical Report, Air Force Wright Aeronautical Laboratories, September, 1984.

[E&V Team 85] E&V Team.
Technical Coordination Strategy Document, Version 2.0.
Technical Report, Air Force Wright Aeronautical Laboratories, August, 1985.

[Fairley 85] Fairley, R.E.
Software Engineering Concepts.
McGraw-Hill, 1985.

[Foreman 87] Foreman, J.G. and Goodenough, J.B.
Ada Adoption Handbook: A Program Manager's Guide.
Technical Report CMU/SEI-87-TR-9, Software Engineering Institute, May, 1987.

[Graumann 86a] Graumann, D.M.
Minutes of the Ada-JOVIAL Users Group.
Technical Report, Language Control Facility, Wright-Patterson AFB, Ohio, July 7-10, 1986.
Los Angeles, CA.

[Graumann 86b] .
Minutes of the Ada-JOVIAL Users Group.
Technical Report, Language Control Facility, Wright-Patterson AFB, Ohio, November 17-21, 1986.
Charleston, WV.

[Grover 83] Grover, V. and Rajeev, S.
Notes on the Ada Runtime Kit (ARK).
Technical Report 9074-7, SofTech, May, 1983.

[Grover 85a] Grover, V.
Guidelines for a Minimal Ada Runtime Environment.
Technical Report ESD-TR-85-139, Electronic Systems Division, Hanscom Air Force Base, January, 1985.

[Grover 85b] Grover, V. and Vanderminden, C.
Designing Control Systems in Ada.
Technical Report TP 216, SofTech, 1985.

[Hood 86] Hood, P. and Grover, V.
Designing Real Time Systems in Ada.
Technical Report DAAB07-85-C-K506, US Army Communication and Electronics Command, January, 1986.

[Hook 85] Hook, A.A., Riccardi, G.A., Vilot, M., and Welke, S.
User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC), Version 1.
IDA Paper P-1879, Institute for Defense Analysis, October, 1985.

[IEEE 83] Institute of Electrical and Electronics Engineers.
IEEE Standard Glossary of Software Engineering Terminology .
ANSI/IEEE Std 729-1983, IEEE, 1983.

[Laird 86] Laird, J.D., Burton, B.A., and Koppes, M.R.
Implementation of an Ada Real-Time Executive - A Case Study.
In *Fourth Annual National Conference on Ada Technology*, pages 114-124. U.S. Army Communications-Electronics Command, March, 1986.

[Lomuto 82] Lomuto, N.
How Fast Is X in Ada? or Does Ada Support X?.
Technical Report 9074-1, SofTech, December, 1982.

[Lomuto 83] Lomuto, N.
Options in Ada Implementations.
Technical Report 9074-4, SofTech, April, 1983.

[LRM 83] U.S. Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD 1815A, DoD, January, 1983.

[MacLaren 80] MacLaren, L.
Evolving Toward Ada in Real-Time Systems.
In *Proceedings of the ACM-SIGplan Symposium on the Ada Programmng Language*. November, 1980.

[NAVSEA 83] NAVSEA.
Ada Runtime Support Environment Requirements Analysis Study.
Technical Report 0967-LP-598-9770, NAVSEA, August, 1983.

[Nissen 82] Nissen, J.C.P. and Wichmann, B.A.
Ada-Europe Guidelines for Ada Compiler Specification and Selection.
ACM Ada Letters 3(5):50-62, March, 1982.

[Pepper 86] Pepper, W.S., IV.
An Experimental Utilization of Ada in Real-Time Interactive Avionics Communication Application.
In *Fourth Annual National Conference on Ada Technology*, pages 8-12. March, 1986.

[Pierce 86] Pierce, R.H., Marshall, I., and Bluck, S.D.
An Introduction to the MoD Ada Evaluation System.
Technical Report Report Number 5485, Software Sciences Ltd., June, 1986.

[Pressman 82] Pressman, R.S.
Software Engineering - A Practitioners Approach.
McGraw-Hill International, 1982.

[Reino 86] Kurki-Suonio, R.
An Operational Model for Ada Tasking.
Technical Report 1/1986, Tampere University of Technology, 1986.

[Rodriguez 86] Rodriguez, T. and Griffin, L.
An Ada Tracker - Experiences and Lessons Learned.
In *Fourth Annual National Conference on Ada Technology*, pages 1-7. U.S. Army Communications-Electronics Command, March, 1986.

[Ruane 85] Ruane, M.F., Cheikes, B.A., and Galia, J.H.
Ada Runtime Environment Characterization for JAMPS.
Technical Report MTR-9614, MITRE, September, 1985.

[SofTech 84] SofTech.
Real-Time Ada.
Technical Report DAAB07-83-C-K514, US Army Communication and Electronics Command, July, 1984.

[Sonicraft 86] Sonicraft.
Sonicraft Experience with Ada in Weapons Systems.
Technical Report, Presented at the E&V Team Meeting, Dayton, Ohio, June 4, 1986.

[Squire 85] Squire, J.
Performance Issues Working Group Workshop.
Technical Report, ACM SIGAda Users Committee, Baltimore, MD, July 15 and 16, 1985.

[Squire 86] Squire, J.
Performance Issues Working Group Workshop.
Technical Report, ACM SIGAda Users Committee, Fort Lauderdale, FL, March, 1986.

[TASC 86] The Analytical Sciences Corporation.
E&V Classification Schema Report, Draft Version 2.0.
Technical Report TR-5234-2, TASC, July, 1986.

[TIAIM 85] Texas Instruments.
APSE Interactive Monitor -- Final Report on Interface Analysis and Software Engineering Techniques.
Naval Ocean Systems Center Contract No. N66001-82-C-0440, Equipment Group - ACSL, July, 1985.

[Weiderman 87] Weiderman, N.H., Haberman, A.N., et al.
Evaluation of Ada Environments.
Technical Report CMU/SEI-87-TR-1, Software Engineering Institute, March, 1987.

Table of Contents

List of Figures