Technical Report

CMU/SEI-89-TR-022
ESD-TR-89-030

Real-Time Software Engineering in Ada: Observations and Guidelines

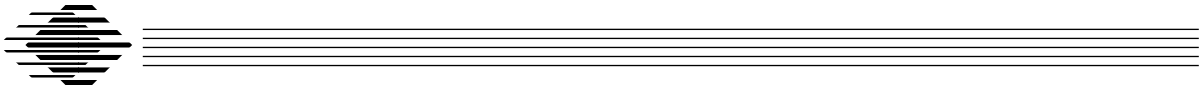
Mark W. Borger
Mark H. Klein
Robert A. Veltre

September 1989

Technical Report

CMU/SEI-89-TR-022
ESD-TR-89-030
September 1989

Real-Time Software Engineering in Ada: Observations and Guidelines



Mark W. Borger
Mark H. Klein
Robert A. Veltre

Real-Time Embedded Systems Testbed (REST) Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

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

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Real-Time Software Engineering in Ada: Observations and Guidelines

Abstract: Two important aspects of developing a real-time system are controlling devices and managing concurrency. In this report, we present several techniques for controlling devices with Ada and several Ada tasking paradigms for managing concurrency. The material presented in this report is taken from our experiences in developing a real-time embedded system in Ada, and we use examples from this system to illustrate the various methods we present. We begin by describing our experiences using Ada to control devices. Specifically, we identify issues related to accessing device registers and handling interrupts, and present techniques for dealing with such issues. We then recount our experiences using Ada to manage concurrency. Specifically, we present coding paradigms for implementing periodicity and constructing synchronization mechanisms. We illustrate analytical methods for determining the schedulability of a task set. We then discuss the effect of aperiodic processing requirements on the schedulability of a task set.

1. Introduction

A fundamental goal of the Real-Time Embedded Systems Testbed (REST) Project at the SEI is to examine Ada technology from a real-time systems perspective. Our basic strategy has been to define a representative real-time embedded problem, develop the system in Ada, and make observations along the way. This strategy has resulted in a real-time Ada artifact that has provided us with many insights into real-time programming issues and the state of Ada technology.

Our development effort was atypical for various reasons. First, the purpose of the effort was to experiment with real-time systems, not to deliver a real-time system. Second, certain Ada features are sometimes not used to avoid the risks associated with them. We chose instead to exercise these features to explore their efficacy in developing real-time systems. For example, we believe the use of Ada tasking separates this effort from other early real-time Ada efforts. Finally, a close association with the Advanced Real-Time Technology (ART) Project within Carnegie Mellon University's School of Computer Science and with the Real-Time Scheduling in Ada (RTSIA) Project at the SEI has allowed us to use the latest in analytical methods whenever possible.

This report[1] presents a set of observations and guidelines drawn from our experiences in developing the Ada artifact. Specific emphases are placed on controlling devices, managing concurrency through the use of Ada tasking, and applying analytical methods to predict and

[1]The following reviewers provided many valuable comments and suggestions: Neal Altman, Nancy Belz, Rich D'Ippolito, Pat Donohoe, Ken Fowler, Michael Gagliardi, John Goodenough, Tim McCardle, Ragunathan Rajkumar, Lui Sha, Roger Van Scoy, and Nelson Weiderman.

understand the runtime behavior of Ada software in a deadline-driven environment. Three classes of tasks are introduced: periodics, servers, and aperiodics. Analytical methods applicable to periodics and servers are also presented. An extended example drawn from the artifact is used to illustrate the various analytical methods. The conclusion summarizes the major lessons we have learned.

2. Controlling Devices Using Ada

A common characteristic of real-time systems is a strong reliance on time-critical interactions (e.g., input and output) with a set of potentially heterogeneous devices. To control devices through application code, one must be able to:

1. Map data types efficiently and accurately onto an underlying machine's architecture to access I/O memory (e.g., device registers).

2. Enable, disable, and/or handle device interrupts.

The following subsections discuss several of our experiences and offer recommendations about how to use the Ada programming language to implement these two types of required capabilities. Although we realize that using assembly language to control devices is always a viable alternative to using Ada, our intent is to illustrate how Ada *could* be used to achieve the same functionality.

2.1. Accessing Device Registers

To control a hardware device, you must be able to access its register set. Ada provides representation clauses that allow you to specify the layout of data types to represent device registers. In particular, Ada representation clauses allow you to specify the storage layout of a record type, including the alignment, order, storage location, bit positions within the storage location, and size of the components. The record representation clause, in combination with a length clause, allows you to specify precisely defined data structures for interfacing with hardware devices. For example, Figure 2-1 demonstrates the use of a record representation clause, length clause, and address clause to define the structure, size, and memory location of a receiver buffer register for a serial communications device.

The length clause used in Figure 2-1 guarantees that exactly 16 bits will be used for representing the buffer register. Additionally, this code segment defines RX_Buffer, a variable to store the contents of the device's receiver buffer register. An Ada address clause immediately follows the declaration of the RX_Buffer variable in Figure 2-1. This address clause specifies an address in memory to which the RX_Buffer variable is to be mapped. In this example, Memory_Address is an instantiation of Unchecked_Conversion that converts an unsigned word into an address.

Accessing device registers using Ada involves issues other than just correct data representation. The following list, which is summarized from our experiences in developing an application for an inertial navigation system (INS) simulator [9, 8], contains some of the most important observations we made when using Ada to read and write device registers.

1. If the memory location of a device register is known at compile time then the example code shown in Figure 2-1 is applicable. However, if the device register address is not known until runtime (e.g., it is returned from a kernel service call), a different approach must be used to access the register. In

```
type Byte is range 0..16#FF#;

type Line_Number_Type is range 0..16#3#;

type Receiver_Buffer is record
  Receive_Char         : Byte;
  Receiver_Line_Number : Line_Number_Type;
  Parity_Error         : BOOLEAN;
  Framing_Error        : BOOLEAN;
  Overrun_Error        : BOOLEAN;
  Data_Valid           : BOOLEAN;
end record;

for Receiver_Buffer use record at mod 2;
  Receive_Char         at 0 range 0..7;
  Receiver_Line_Number at 0 range 8..9;
  Parity_Error         at 0 range 12..12;
  Framing_Error        at 0 range 13..13;
  Overrun_Error        at 0 range 14..14;
  Data_Valid           at 0 range 15..15;
end record;

for Receiver_Buffer'SIZE use 16;

RX_Buffer : Receiver_Buffer;
for RX_Buffer use at Memory_Address(16#00FF_0000#);
```

Figure 2-1: Receiver Buffer Representation Clause

such a case, an Ada subprogram (or a declare block) with local variables, whose memory locations are specified via dynamic address clauses, provides an effective way of accessing data stored in particular locations of memory. The example code shown in Figure 2-2 illustrates this technique, using a function and a dynamic address clause for the RX_Buffer variable shown earlier in Figure 2-1. In Figure 2-2, the address of the receiver buffer register is passed as a parameter and is used in a dynamic address clause to map the local variable RX_Buffer to the memory location corresponding to the device's data register. An alternative approach to using a dynamic address clause is to use an access variable (i.e., a pointer to an object of type Receiver_Buffer) and assign the pointer a value (address) at runtime.[2]

2. Some hardware devices have registers that serve a dual (read/write) purpose. For such registers, different register layouts are used for reading and writing. These special dual-purpose device registers can be modeled in Ada as two different device registers. Using the approach outlined in Figure 2-1, you can define two local variables and use two address clauses to specify the same memory location to represent the read and write view of the register for read-

[2]This approach assumes that you can perform an unchecked conversion from an address to the appropriate access type.

ing and writing the register's contents.[3] One alternative to this overlaying of variables with different types is to use one additional variable declared as an unsigned integer type (having the same number of bits as the device register) through which I/O operations can be performed. However, using this approach introduces the need to do unchecked conversions from the unsigned type to the read register type (for reading) and from the write register type to the unsigned type (for writing) to access the device register.

3. Some hardware architectures restrict the kinds of machine instructions that can be used for accessing device registers. For example, some devices are only word addressable and will produce erroneous results if accessed through bit field instructions [1]. You should be aware of any such device access restrictions and inspect your generated code to check for any such instructions that might violate those restrictions.

4. You should be aware of any global code optimizations that might eliminate important assignment statements being used to control a device. For example, many hardware devices require explicit initialization sequences that can involve a sequence of assignments to the same device control register. To a global optimizer, some of these assignment statements may appear to be redundant (through variable definition or use analysis) and therefore would be eliminated. But eliminating such assignment statements can effect the operational correctness of your code.

```
function Get_Character (RX_Buffer_Address : in ADDRESS)
        return CHARACTER is
-----------
--   Map variable RX_Buffer to serial I/O device's receive buffer
--   using a dynamic address clause.
-----------
  Read_Character : CHARACTER := ASCII.NUL;
  RX_Buffer      : Receiver_Buffer;
  for RX_Buffer use at RX_Buffer_Address;
begin
  if RX_Buffer.Data_Valid then
    ----------
    --   Assumes an 8 bit CHARACTER type representation; otherwise
    --   a potential data size mismatch could occur for the conversion.
    ----------
    Read_Character :=
          Convert_To_Char(RX_Buffer.Receive_Char);
  end if;
  return Read_Character;
end Get_Character;
```

Figure 2-2: Using a Dynamic Address Clause to Access a Device Register

[3] In effect, this approach overlays two separate data objects onto the same memory address. Although the *Ada Language Reference Manual* (LRM) [20] states that overlaying objects is erroneous, some Ada implementations do support it. Here, erroneous means that the runtime behavior of this code cannot be guaranteed across Ada implementations.

2.2. Interrupt Handling

To handle hardware interrupts in application software, Ada defines task interrupt entries, which enable binding, via an address clause, of a hardware interrupt to a task entry. The semantics of the interrupt handling model described in the *Ada Language Reference Manual* (LRM) [20] state that 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. However, the LRM does not specify the type of entry call which must be made to interrupt entries, but rather gives implementors the freedom to implement any calling mechanism (e.g., normal entry call, conditional entry call) provided that it preserves the above semantics. Figure 2-3 illustrates a generic interrupt handler task written in the spirit of Ada that builds on the example code shown in Figures 2-1 and 2-2.

```
task Keyboard_Watcher is
  entry Character_Entered;
  for Character_Entered use at RX_Interrupt_Address;
end Keyboard_Watcher;

task body Keyboard_Watcher is
  My_Character : CHARACTER;
begin
  loop
    accept Character_Entered do
      My_Character := Get_Character(RX_Buffer_Address);
      Place_Character_In_Buffer;
      Reset_RX_Interrupt;
    end Character_Entered;
  end loop;
end Keyboard_Watcher;
```

Figure 2-3: Interrupt Handling Ada Task

In this example, the Keyboard_Watcher task has an entry named Character_Entered, which is bound to the receiver interrupt of a serial I/O device. When a receiver interrupt occurs, an entry call to Character_Entered will be made. Within the accept body, a call to the Get_Character function of Figure 2-2 is made to fetch a character from the receiver buffer register. Since the language definition does not specify how the entry call to Character_Entered is to be implemented, Ada implementations can vary in how they achieve this call.

It has been our experience that the techniques employed for implementing Ada interrupt entry calls vary along at least two dimensions:

1. When the entry call is made relative to the time of the interrupt event. In general, the interrupt entry call can be made immediately upon receipt of the interrupt or it can be deferred.

2. How the entry call is achieved, i.e., what kind of synchronization mechanism is used. For example, the interrupt entry call can be either a normal, conditional, or timed entry call.

Some of the Ada interrupt handling implementation schemes we have encountered are summarized below.

1. Interrupt Service Routine (ISR): The ISR interrupt handling scheme involves using an Ada procedure for handling hardware interrupts. Under this approach, the address of an Ada subprogram (ISR) is stored as the interrupt vector for the hardware device. When the device interrupts, the interrupt handling logic of the CPU sets the program counter to the starting address of the ISR, and thus effects a subroutine call to the ISR. Typically, this approach has the least amount of runtime overhead, but this minimal overhead may come at the expense of flexibility within the ISR procedure. For example, common restrictions on ISR code include: no task entry calls, no calls to Text_IO subprograms, and limited access to the user's data space. Of the Ada interrupt handling models discussed in this paper, the ISR approach is the only one that isn't patterned after the LRM model in some way, since an Ada task is not used to handle the device interrupt. Normally, however, the ISR code can call an implementation-dependent signaling primitive to synchronize with application tasks, and therefore achieve the effect of an Ada interrupt task. Note that this task synchronization is not immediate, but rather occurs from within the runtime kernel to the application task some time after the interrupt has been completely handled.

2. Deferred Runtime Signal Mechanism: The deferred runtime signal approach implements the semantics of an Ada interrupt entry by using a runtime signal primitive, rather than a full Ada rendezvous, to effect a call to the interrupt handling task. When an interrupt occurs under this scheme, a runtime system ISR posts a signal and completes its execution. The runtime system subsequently maps that signal into a call to the interrupt task's entry. The main advantage of this approach is the minimal runtime overhead needed to service the interrupt. Notice that in this model, servicing the interrupt does not include executing the interrupt entry's accept body. Generally, this model does not limit the operations that can be performed in the interrupt task. However, tasking optimizations often can be performed when certain guidelines are followed within the task body.

3. Ada Rendezvous Mechanism: This approach is suggested in the LRM and uses an Ada rendezvous to call the interrupt handling task. When an interrupt occurs, the runtime makes an Ada entry call to the interrupt task as part of dispatching the interrupt. Since a normal Ada entry call can potentially block, some Ada implementations use a conditional entry call for calling interrupt task entries. For example, the TeleGen2 3.23 VAX/VMS to MC68020 cross-compiler provides the user with a capability for defining a "failure" task to be called if the interrupt task cannot immediately accept the interrupt entry call. The main advantages of this approach are portability and the lack of any restrictions on code in the interrupt task.

   Nevertheless, because this scheme uses an Ada rendezvous for synchronizing with the interrupt task, it has more runtime overhead than the other approaches. However, optimized interrupt entry calls are possible when the application task code follows certain coding restrictions. For example, the VADS

5.7 Sun/UNIX to MC68020 cross-compiler has a *passive task* scheme that implements the semantics of an Ada interrupt entry by using compiler optimizations that treat the code inside the accept body of the interrupt entry as a subprogram protected by a semaphore.

2.3. Recommendations

Although representation clauses are part of the language definition, only recently has the Ada Compiler Validation Capability (ACVC) (suite 1.10) begun to test Ada implementations for compliance to Chapter 13 of the *Ada Language Reference Manual*. Unfortunately, most of these added ACVC tests check to ensure that syntactically illegal clauses are rejected; only a few of the tests actually check for the presence of representation clause functionality. As a consequence of this inadequate testing in the ACVC suite, the extent and reliability to which compilers implement the features defined in Chapter 13 of the LRM varies considerably. Our recommendations for controlling devices in Ada are summarized below.

1. **Determine representation specification restrictions.** Determine if your compiler provides sufficient support to handle the range of external interface specifications expected for your application. In particular, determine whether or not representation clauses are supported, and if so, what restrictions on their use are dictated by your Ada implementation.[4] For instance, if your Ada implementation does not support representation specifications at the bit level, then writing device controlling code in Ada will be difficult. Also, know if your Ada implementation imposes any restrictions with respect to aligning data records in memory. Some hardware devices are strictly odd- or even-word addressable; furthermore, across various devices, register sizes often vary between 8, 16, or 32 bits.

2. **Verify data structure representation.** Verify that your representation clauses are implemented as expected. Having a technique (e.g., inspecting the generated code) for verifying that your data structures are being represented correctly is essential. Know how your Ada implementation's scheme for numbering bytes, words, and long words maps onto the target's data organization in memory. Also, an important implementation detail is whether your Ada compiler uses a left-to-right or right-to-left bit field ordering scheme for its record representation clauses. Additionally, unsigned types can be very important in developing code to control a hardware device. The critical issue here is to ensure that an exact number of bits will be used for mapping onto a device register defined in the I/O address space.

3. **Assess the efficiency of generated code.** Besides verifying the correctness of the generated code, consider its efficiency. For example, does the code generator use the least expensive instructions for accessing bit level components of device register records?

---

[4]The *Ada Language Reference Manual* requires that a list of all restrictions on representation clauses be provided in Appendix F.

4. **Understand the syntax and semantics of interrupt entries.** Determine if your Ada implementation supports interrupt entries, and if so, what implementation scheme(s) they use. Be aware of the following issues when using Ada interrupt entries.

    a. Performance: What is the latency from the time the hardware interrupts until the code in the Ada interrupt task begins executing? How much runtime overhead is incurred on a call to an interrupt entry? How long are interrupts disabled by the runtime system during interrupt processing?

    b. Implementation restrictions: What restrictions have been placed on interrupt tasks? Can they be called from the application code? Can they perform I/O operations? Can they share data with other pieces of applications code? Can they make entry calls to other tasks? What type of interrupt entry call is made when a interrupt occurs: normal Ada entry call, timed entry call, conditional entry call, signal? Are interrupt entry calls queued or can they be lost?

5. **Consider alternatives to Ada interrupt entries.** If your Ada implementation does not support interrupt entries or you choose not to use them, you have some other options as summarized below.

    a. You can code the entire interrupt handler (e.g., subprogram) in Ada and associate its starting memory location with the interrupt either through an address clause or by calling a runtime kernel service.

    b. You can perform the above interrupt linkage, but implement the declared Ada interrupt service routine in another language (probably assembler) and use either pragma INTERFACE or machine code insert statements for the subprogram body.

    c. You can rely on an existing device driver and interrupt service routine to handle device interrupts.

    Of course, to fully support the semantics of an Ada interrupt entry call these ISRs would need to have a signaling capability in order to synchronize with application level tasks.

3. Managing Concurrency in Ada

There are two fundamental methods for managing concurrency on a single processor:

1. Through the use of time-division multiplexing (e.g., a cyclic executive).

2. Through the use of a preemptive scheduling mechanism.

The basic tradeoff between these two approaches is complexity and maintainability of application design versus nondeterminism of execution timing behavior. The manual dissection and multiplexing of execution threads, which is necessary for designing and maintaining a cyclic executive, may be a difficult task; however, the resulting execution timing behavior is basically deterministic. The reverse is true for a preemptive scheduler, which allows for the separation of scheduling and functional aspects of the program but introduces nondeterminism into the timing behavior. Ada tasking provides a vehicle for separating scheduling and functional concerns. The use of Ada tasking in conjunction with analytical methods based on scheduling theory allows us to maintain this separation and at the same time predict and understand an Ada program's timing behavior.

This section offers guidelines for using Ada tasks to construct real-time software that will execute in an understandable and predictable manner. Following the spirit of Sha [17], we have grouped tasks into three major categories: periodics, servers, and aperiodics. We compare and contrast several design paradigms for periodics and servers, and illustrate associated techniques for performing schedulability analysis. Aperiodics are then briefly discussed with an emphasis on how they affect the schedulability of a task set. Finally, we offer a set of recommendations based on our experiences.

3.1. Periodic Tasks

Periodic processing requirements with hard deadlines[5] are common in real-time systems (e.g., reading a device and sending messages at a specified frequency). Paradigms that are amenable to schedulability analysis are needed for implementing periodicity in order to guarantee that rigorous timing requirements will always be satisfied. The following subsections discuss alternatives for performing periodic processing in Ada and illustrate the practical application of analytical methods to a periodic task set extracted from an INS simulator [9, 8].

[5] A task's deadline is considered to be hard if meeting the deadline is critical to the system's operation. On the other hand, it is desirable but not necessary to meet a soft deadline [19].

3.1.1. Implementing Periodic Tasks

For this discussion, we define a *periodic task* as one that is ready to execute at fixed intervals in time (or periods) and must complete execution before the end of the interval (which is the start of the next period). We outline four models for designing periodic tasks in Ada, summarizing the design approach of each model and commenting on them.

Delay Model

The most widely known model for implementing periodic tasks in Ada, and the one that is offered in the Ada LRM, involves using Ada's delay statement as shown in Figure 3-1. In this approach, the periodic tasks operate autonomously in an endless loop by performing their computations, computing their next activation time, and delaying until the start of their next period.

```
task body Periodic_Task is
  Next_Time : Calendar.Time := Task_Start_Time;
begin
  delay (Task_Start_Time - Calendar.Clock);
  loop
    Do_Work;
    Next_Time := Next_Time + Task_Period;
    delay (Next_Time - Calendar.Clock);
  end loop;
end Periodic_Task;
```

Figure 3-1: Autonomous Periodic Task Using Ada Delay

A major drawback with this approach is that it suffers from a phenomenon that we call *delay jitter*, which occurs when a task is preempted between the time the clock is read and the time at which the delay begins. If such preemption occurs, the preempted task will start its delay later than it should and effectively postpone the starting point of its next period.

The problem caused by delay jitter can be significant since the preemption time can be long. A low-priority periodic task could be preempted for the entire execution time of a higher priority task. There are circumstances under which this problem will result in the task missing deadlines. Moreover, this phenomenon alters the periodicity of the task and thus violates a premise used in the analytical techniques described later in this report. For these reasons, we cannot recommend this approach. However, there may be circumstances where this approach is viable. One example is a system in which it can be shown that deadlines will be met even in the presence of delay jitter. Additionally, if one can prevent delay jitter by ensuring that a periodic task cannot be preempted from the time the clock is read until the delay is started, then this alternative is more attractive. However, even if delay jitter is eliminated in this manner, the model still incurs the overhead of using Calendar.Clock. The need to read the clock is inherent in using delay intervals to achieve periodicity [21]. Delaying until an absolute time eliminates this problem.

Supervisor/Worker Model

A slight variation of the previous model uses pairs of tasks to achieve periodic processing as shown in Figure 3-2. In this approach, each periodic (worker) task has a higher priority supervisor task associated with it. In fact, to reduce the delay jitter inherent in the previous model, each supervisor task has a priority which is higher than all worker tasks; therefore a supervisor cannot be preempted by a worker. The priorities of the supervisors are ordered based on the priority of the associated worker task. Each supervisor task is responsible for achieving the desired periodic execution. The worker task embodies a simple endless loop that accepts the periodic activation calls from its supervisor and then performs its computations.

```
task body Periodic_Supervisor is
  Next_Time : Calendar.Time := Task_Start_Time;
begin
  delay (Task_Start_Time - Calendar.Clock);

  loop
    select
      Periodic_Task.Activate;
    else
      Handle_Missed_Deadline;
    end select;

    Next_Time := Next_Time + Task_Period;
    delay (Next_Time - Calendar.Clock);
  end loop;
end Periodic_Supervisor;

task body Periodic_Task is
begin
  loop
    accept Activate;
    Do_Work;
  end loop;
end Periodic_Task;
```

Figure 3-2: Supervisor/Worker Model

The preemption time, which can potentially cause delay jitter, is bounded in this model because the supervisor tasks perform the delaying (on behalf of their workers) and because a supervisor's only work is to periodically activate a worker task via an entry call. Nonetheless, this approach still suffers from the delay jitter problem because the supervisors can preempt each other. Additionally, the cost of an extra rendezvous (to activate the worker task) is introduced. This method can be recommended only for situations in which the execution of a conditional entry call is an insignificant fraction of the worker's period. If this is the case, then the supervisor can detect missed deadlines by using a conditional entry call to activate its associated worker.

Task Dispatcher Model

The previous model requires multiple supervisor tasks, one for each periodic task. This model uses only one supervisor task which we call a task dispatcher. The task dispatcher is responsible for periodically activating all worker tasks. One method for implementing a task dispatcher is to use a real-time clock. The real-time clock serves as a source of interrupts that signal the task dispatcher; the task dispatcher, in turn, activates the appropriate periodic task(s). An example demonstrating this scheme is shown in Figure 3-3.

```
task body Periodic_Dispatcher is
begin
  loop
    accept Clock_Interrupt;

    loop
      Determine_Which_Periodic_Task_To_Activate;
      exit(When_No_Periodic_Task_To_Activate);

      select
        Periodic_Task.Activate;
      else
        Handle_Missed_Deadline;
      end select;
    end loop;

  end loop;
end Periodic_Dispatcher;

task body Periodic_Task is
begin
  loop
    accept Activate;
    Do_Work;
  end loop;
end Periodic_Task;
```

Figure 3-3: Real-Time Ada Task Dispatcher

In this approach, the task dispatcher receives an interrupt from a real-time clock only when a periodic task (or a set of tasks) should be activated. The dispatcher task iteratively determines which periodic task to activate and makes a conditional entry call to that task's Activate entry. If this conditional entry call fails, the periodic task has missed a deadline and appropriate action can be taken. Details on how to implement this periodic tasking scheme in Ada are explained by Borger [2].

There are several advantages to this approach. First, it does not suffer from delay jitter since the Periodic_Dispatcher executes at the highest priority and consequently will not suffer from preemption.[6] Second, it allows for a finer notion of time than is readily available

[6]We assume that the task dispatcher will not be preempted by other tasks or by other interrupts; i.e., interrupts from the real-time clock never will be masked. We also assume that the clock automatically rolls over. See [21] for a discussion of drift due to resetting a real-time clock.

from most Ada systems. (For example, using this model we were able to effectively reduce our resolution of time from 10 ms to 2.56 ms for the INS.) Finally, this is a solution that can be used today, because modifications to the Ada language are not necessary; it only requires that an implementation supports interrupts and a real-time clock. We demonstrate the application of scheduling theory to this technique later in this section.

Nevertheless, there are three main drawbacks to this approach. First, the task dispatcher introduces the runtime overhead for handling the clock interrupts and activating tasks via a conditional entry call. Second, the application programmer must play a role (i.e., implementing the task dispatcher) in managing concurrency; ideally, this function should be abstracted out of the application program. Finally, a real-time clock is required. The last model to be discussed offers a relatively simple, efficient, and easy to use approach for implementing periodic tasks in Ada.

Delay_Until Model

An improvement over all the previous models, which is discussed by Volz [21], is to code autonomous periodic tasks using a Delay_Until absolute time mechanism as shown in Figure 3-4.

```
task body Periodic_Task is
  Next_Time : Calendar.Time := Task_Start_Time;
begin
  delay_until(Task_Start_Time);
  loop
    Do_Work;
    Next_Time := Next_Time + Task_Period;
    begin
      delay_until(Next_Time);
    exception
      when CALENDAR.TIME_ERROR => Handle_Missed_Deadline;
    end;
  end loop;
end Periodic_Task;
```

Figure 3-4: Autonomous Periodic Task Using Delay_Until Mechanism

This model is similar to the delay model in that the periodic tasks operate autonomously in an endless loop. The task delays until its initial start time and then becomes ready to execute. When it is the highest priority task that is ready to execute, it enters a loop and repeatedly performs its computations, computes the next activation time based on the previous schedule time and the task's period, and delays until the start of its next period. The Delay_Until mechanism illustrated above has built-in semantics stating that if the specified absolute time has already passed at the time of the Delay_Until subprogram call, the Time_Error exception in package Calendar will be raised. Given such a runtime mechanism, missed deadline detection is straightforward to implement in the application code.

The Delay_Until model does not suffer from delay jitter. For the delay model to work, the calculation of the delay duration followed by the start of the delay must be an atomic action. If it is not atomic, delay jitter results. The Delay_Until model does not have an equivalent atomicity requirement since the Delay_Until is based on absolute time. The only requirement is for the Delay_Until statement to be executed before that absolute time. Although it is not commonly available, we have been working with an Ada system that supports this mechanism.

Several techniques for implementing periodicity through the use of Ada tasking have been discussed, yet we still need a guarantee that our timing requirements will be met. Analytical methods can offer this guarantee. The next section introduces analytical methods that allow us to reason about the timing characteristics and predict the timing behavior of a collection of periodic tasks.

3.1.2. Analyzing the Schedulability of Periodic Tasks

The rate-monotonic algorithm (RMA) [12] provides a foundation for analyzing the schedulability of a set of periodic tasks. This algorithm states that for priority-based preemptive scheduling, the static priority assigned to a periodic task is determined solely by its frequency; higher frequency tasks are assigned higher priorities than lower frequency tasks. If this rule is followed then all tasks are guaranteed to meet their deadlines, provided that CPU utilization for the periodic task set is below $n(2^{1/n}-1)$, where n is the number of tasks. [7] Moreover, this algorithm is optimal; that is, any task set that is schedulable using another fixed priority algorithm is also schedulable using the RMA. Since Ada uses a fixed priority, preemptive scheduling discipline, the RMA applies.

Consider the periodic tasks from the INS task set whose task periods and execution times are shown in Table 3-1.[8] Task priorities are assigned to the six periodic tasks in decreasing order, with P_1 being assigned the highest priority and P_6 being assigned the lowest priority.

[7] $n(2^{1/n}-1)$ converges to 0.69 as the number of tasks is increased.

[8] In Table 3-1, the task periods are exact whereas the execution times are estimates that illustrate the various types of analyses that can be performed. Since this is ongoing work, the actual execution times are still changing.

Task (P_i)	Description	Period (T_i)	Exec. (C_i)	Util. (C_i / T_i)
P_1	Update Ship Attitude	2.56 ms	0.5 ms	19.53 %
P_2	Update Ship Velocity	40.96 ms	5.0 ms	12.21 %
P_3	Send Attitude Message	61.44 ms	15.0 ms	24.41 %
P_4	Send Navigation Message	983.04 ms	30.0 ms	3.05 %
P_5	Update Console Display	1024.0 ms	50.0 ms	4.88 %
P_6	Update Ship Position	1280.0 ms	1.0 ms	0.08 %

Table 3-1: Periodic Tasks from the INS

The total utilization for this task set is 64.16%, so the task set is schedulable according to the RMA.[9] This is a very easy schedulability test to apply. However, there are several practical concerns that are not treated by the algorithm. These limitations include:

1. Context switching overhead is assumed to be negligible.

2. The overhead required to activate periodic tasks is assumed to be negligible.

3. The RMA assumes that tasks do not interact.

The next two sections discuss techniques for including context switching and task dispatching overheads in the schedulability analysis. Section 3.2 discusses the effects of task synchronization on the schedulability of the periodic tasks.

Including Context Switching Overhead in the Analysis

Task switching time can be introduced into the utilization calculation by adding it to task execution time [18]. When a task preempts a lower priority task, a context switch saves the state of the lower priority task and establishes the execution state (e.g., register values, program counter and stack pointer) of the higher priority task. After the higher priority task completes execution, a context switch restores the state of the lower priority task. If C_s represents this context switching overhead and C_i is the execution time of task P_i, then C_i+2C_s is the new execution time to be used in the analysis.[10] This allows us to make the

[9] Utilization is calculated by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

where C_i and T_i represent the execution time and the period of task P_i, respectively.

[10] Note that this treatment of context switching overhead assumes perfect preemptability (i.e., that the runtime may itself be preempted during a context switch). Perfect preemptability is an assumption of rate-monotonic theory. In the event that the runtime system does not exhibit perfect preemptability, we can use the techniques for handling blocking time that are discussed later.

observation that the example task set in Table 3-1 will continue to be schedulable provided that the following inequality is satisfied.

$$\frac{(0.5+2C_s)}{2.56}+\frac{(5.0+2C_s)}{40.96}+\frac{(15.0+2C_s)}{61.44}+\frac{(30.0+2C_s)}{983.04}+\frac{(50.0+2C_s)}{1024.00}+\frac{(0.8+2C_s)}{1280.00} \leq 6(2^{1/6}-1)=0.734$$

Using the above inequality and solving for C_s we find that the periodic tasks can sustain 0.106 ms of context switching overhead and remain schedulable. Note that the RMA uses a pessimistic utilization bound which is based upon a worst-case analysis. It has been shown in [10] that on the average the RMA can schedule task sets with 88% periodic utilization. In fact, this task set can sustain significantly more overhead and remain schedulable as is shown below.

A necessary and sufficient condition for the schedulability of a periodic task set (allowing for any possible task phasing) is also provided in [10]. In essence, this schedulability test requires that each task be examined to ensure that it can meet its first deadline when all tasks are started at the same time (which has been shown to be the worst task phasing [12]). For each particular task, this is accomplished by numerically determining if there is sufficient time to allow the task to meet its first deadline [18]. In the following example we will use the same technique to illustrate a method for determining the absolute maximum amount of overhead sustainable by the INS periodic task set.

Consider task P_1. The following inequality tests if task P_1 can sustain the $2C_s$ units of overhead and still meet its deadline.

$$C_1+2C_s \leq T_1 \ \text{^} \ 0.5+2C_s \leq 2.56 \ \text{^} \ C_s=1.03\,ms$$

Solving for C_s, we determine that it will remain schedulable provided that context switching overhead does not exceed 1.03 ms.

Consider task P_2. The strategy for this task is to determine if there is a point in time between time $t=0$ and time $t=T_2$, the end of the period of P_2, such that P_2 has completed its execution. For each scheduling point (i.e., the starting point of each task's period), we must determine if the execution of task P_1 to that point and the execution of all of P_2 (including overhead) have completed. For example, the first inequality below tests whether the execution of P_1 and P_2 have completed by time $t=T_1$. The second inequality tests whether P_1 has executed twice and P_2 has completed by time $t=2T_1$. We continue in this fashion until all scheduling points from time $t=0$ to the end of the period of P_2 have been examined. The goal is to determine the maximum C_s such that at least one of the following inequalities holds:

$$
\begin{array}{rcl}
(C_1 + 2C_s) + (C_2 + 2C_s) \leq T_1 \ \wedge & (0.5 + 2C_s) + (5.0 + 2C_s) \leq 2.56 \ \wedge & C_s < 0, \ or \\
2(C_1 + 2C_s) + (C_2 + 2C_s) \leq 2T_1 \ \wedge & 2(0.5 + 2C_s) + (5.0 + 2C_s) \leq 5.12 \ \wedge & C_s < 0, \ or \\
3(C_1 + 2C_s) + (C_2 + 2C_s) \leq 3T_1 \ \wedge & 3(0.5 + 2C_s) + (5.0 + 2C_s) \leq 7.68 \ \wedge & C_s = 0.148\,ms, \ or \\
\ldots & \ldots & \ldots \\
16(C_1 + 2C_s) + (C_2 + 2C_s) \leq 16T_1 \ \wedge & & 16(0.5 + 2C_s) + (5.0 + 2C_s) \\
16(C_1 + 2C_s) + (C_2 + 2C_s) \leq T_2 \ \wedge & 16(0.5 + 2C_s) + (5.0 + 2C_s) \leq & 40.96
\end{array}
$$

C_s would have to be negative for the first and second inequality to be true. The third inequality can sustain a positive C_s. The last two inequalities can sustain the largest C_s. Notice that since the first two tasks are harmonic, the last two inequalities are identical. P_2 will remain schedulable provided that context switching overhead does not exceed 0.822 ms. For the first two tasks to remain schedulable, the maximum amount of sustainable overhead will be the minimum of the overheads calculated for P_1 and P_2.

Table 3-2 lists the the maximum overhead sustainable by each task. To determine the maximum sustainable overhead by the periodic task set, we take the minimum of all of the listed numbers, which is 0.41 ms. This is a significant improvement over the previous calculation of 0.106 ms.

Task	Maximum C_s
P_1	1.03 ms
P_2	0.82 ms
P_3	0.45 ms
P_4	0.47 ms
P_5	0.41 ms
P_6	0.41 ms

Table 3-2: Maximum Sustainable Context Switching for Each Task

Analyzing the Task Dispatching Mechanism

The INS uses a real-time clock and associated task dispatcher to precisely schedule the six periodic tasks in the task set. This raises two questions:

- How can the task dispatcher be modeled analytically?
- How does the task dispatcher affect the schedulability of the periodic tasks?

Assume that the task dispatcher has an Ada priority greater than that of P_1, the highest priority task in the task set. Since the task dispatcher executes once each time a periodic task is to be activated (and no more often than that), we can treat the execution time of the task dispatcher as additional execution time on behalf of each periodic task. However, it is important to emphasize that this additional execution time occurs at the highest priority.

Thus, the dispatching of a lower priority task can temporarily block the execution of a higher priority task. We must be careful to account for these blocking effects in addition to the preemption effects.

Preemption occurs naturally when a low-priority task is delayed by the execution of a higher priority task. In contrast, *blocking* occurs when a high-priority task is delayed by the execution of a lower priority task. In the context of the task dispatching analysis, blocking may occur when tasks are activated. For example, since the execution time of the dispatcher when activating P_3 is considered additional execution time on behalf of P_3, then P_3 can effectively block both P_1 and P_2 because P_3 is a lower priority task. Therefore, the time taken from P_1 and P_2 when P_3 is activated must be considered blocking time. To treat the task dispatching mechanism analytically, we must also account for the blocking time resulting from the dispatching of periodic tasks.

Now consider the case of P_1. When the task dispatcher activates P_1, the execution time of the task dispatcher (referred to as D_e) can be modeled as additional execution time on behalf of P_1. But in the worst possible case, where all six periodic tasks must be activated at the same time, the task dispatcher must execute five additional times to activate the remaining tasks. Since the five additional iterations of the task dispatcher are on behalf of lower priority tasks, P_1 may be blocked for as long as $5D_b$, where D_b represents the time taken by the dispatcher to activate a lower priority task. Continuing the analysis in this manner, we find that each periodic task must be able to accommodate additional execution time due to being dispatched (D_e)[11] and blocking time caused by the dispatching of all lower priority periodic tasks (a multiple of D_b).

The analytical techniques for independent periodic tasks do not take into account blocking time. However, it has been shown in [16] that the analytical techniques for determining the schedulability of a set of independent periodic tasks can be generalized to account for blocking time. When we apply these analytical techniques to the problem of determining the schedulability of the periodic tasks in the presence of the task dispatcher, we find that the series of inequalities shown in Figure 3-5 express sufficient conditions for the periodic tasks to be schedulable. There are six inequalities, one for each task. Each inequality determines whether or not a periodic task can accommodate its own execution time (which includes additional execution time caused by the task dispatcher), preemption time caused by all higher priority tasks, and blocking time caused by the dispatching of all lower priority tasks. Note in Figure 3-5 that the last inequality does not contain a D_b term. Since P_6 is the lowest priority periodic task, it cannot be blocked by the dispatching of any lower priority periodic

[11] D_e includes the time that it takes to: switch from the currently executing task to the task dispatcher, determine which task(s) to activate, rendezvous with the task(s) being activated, and switch back to the highest priority task ready to execute.

tasks. Also note that since $D_b = D_e$ in the worst case,[12] the distinction between D_b and D_e is to emphasize that execution time and blocking time are treated differently.

With $C_i' = C_i + D_e$,

$$\frac{C_1'}{T_1} + \frac{5D_b}{T_1} \leq 1(2^{1/1} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{4D_b}{T_2} \leq 2(2^{1/2} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{3D_b}{T_3} \leq 3(2^{1/3} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{2D_b}{T_4} \leq 4(2^{1/4} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3}{T_3'} + \frac{C_4'}{T_4} + \frac{C_5'}{T_5} + \frac{1D_b}{T_5} \leq 5(2^{1/5} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{C_5'}{T_5} + \frac{C_6'}{T_6} \leq 6(2^{1/6} - 1)$$

Figure 3-5: Sufficient Conditions for Periodic Tasks to Be Schedulable in the Presence of the Task Dispatching Mechanism

If all of the inequalities of Figure 3-5 are satisfied, then the task set is schedulable. Table 3-3 contains a summary of the various parameters that describe the periodic tasks and the task dispatcher. Substituting values from Table 3-3 into the inequalities of Figure 3-5, we find that all of the inequalities evaluate to true. Therefore, the periodic tasks are still schedulable, even with the overhead of the task dispatcher taken into account. Note that it is our intention to use the INS to verify these predicted results empirically.

[12] When more than one periodic task is activated at the same time, higher priority tasks are activated first. Thus, when a low-priority task is activated at the same time as a high-priority task, $D_b < D_e$, because the task dispatcher incurs the runtime overhead associated with handling the clock interrupt only for the first task activated. When a low-priority task is activated while a high-priority task is already running, the high-priority task will be blocked for as long as D_e. Therefore, $D_b = D_e$ in the worst case.

$D_e = D_b$ = 0.2 ms

Task (P_i)	Period (T_i)	Exec. ($C_i + D_e$)	Figure 3-5 Inequality (Left Side)	(Right Side)
P_1	2.56 ms	0.7 ms	0.47	1.00
P_2	40.96 ms	5.2 ms	0.41	0.82
P_3	61.44 ms	15.2 ms	0.65	0.77
P_4	983.04 ms	30.2 ms	0.68	0.75
P_5	1024.0 ms	50.2 ms	0.73	0.74
P_6	1280.0 ms	1.2 ms	0.73	0.73

Table 3-3: Runtime Characteristics of Periodics with the Task Dispatcher

While the inequalities shown in Figure 3-5 describe sufficient conditions for the task set to be schedulable, the conditions are not necessary. Therefore, should any one of the inequalities evaluate to false, we cannot immediately conclude that the task set is unschedulable. Rather, we would proceed to apply the technique shown in [10] to determine whether or not a timeline exists such that all of the tasks will meet their deadlines.

3.2. Server Tasks

In addition to periodicity requirements, real-time systems often have synchronization requirements since (periodic) tasks must interact to share logical and physical resources. Task synchronization can be achieved in Ada by using server tasks. For our purposes, we will adopt the definition of an Ada server task as presented in [16]: "A *server task* is a task whose accept statements are all contained in a single select statement that is the only statement in the body of an endless loop."

When tasks interact, *priority inversion* [5] occurs if a high-priority task is blocked by a lower priority task. Because such a high-priority task must accommodate blocking from a lower priority task, priority inversion can have a negative effect on the overall schedulability of a task set. Although priority inversion cannot be eliminated, it can be bounded [16].

In this section we will discuss some of our relevant experiences in using Ada server tasks to solve a typical data access and consistency problem. Our design problem concerns providing a synchronization mechanism by which all six of the INS periodic tasks can access a shared table of data. Some of the INS periodic tasks read this shared data, some modify it, and some do both. Since we do not know exactly when a task is going to read or write this data (as we would with a cyclic executive), the INS tasks must synchronize their access of the shared data. We will first illustrate how an ill-conceived Ada tasking design for this problem can lead to uncontrolled priority inversion. Then we will discuss another design alternative which uses Ada tasking in a way that will minimize the blocking time caused by priority inversion. Finally, we will present some analytical techniques which are useful for better understanding the runtime behavior of interacting tasks.

3.2.1. Implementing Server Tasks

In general, a server task's entries correspond to the various services that it provides and we consider a called server task to be executing on behalf of (or as an extension to) its corresponding client task. In this section, we will analyze two different task designs for solving the INS synchronization problem introduced above. The first design uses Ada tasking to implement a binary semaphore for protecting the shared data, whereas the second design alternative implements an Ada server task based on the priority ceiling protocol guidelines discussed in [7, 16].

Semaphore Tasks

A binary semaphore is a widely accepted synchronization primitive that can be used for providing mutually exclusive access to a shared resource such as the INS results table. As illustrated in Figure 3-6, an Ada task can be used to implement such a binary semaphore. Figure 3-6 also illustrates a possible implementation of a periodic client task that uses this semaphore task for gaining access to some shared data.

```
task body Semaphore is
begin
  loop
    accept Enter_Critical_Region;
    accept Exit_Critical_Region;
  end loop;
end Semaphore;

task body Periodic_Client is
begin
  loop
    accept Activate;

    Semaphore.Enter_Critical_Region;
    Do_Critical_Section_Read;
    Semaphore.Exit_Critical_Region;

    Do_Some_More_Work;
  end loop;
end Periodic_Client;
```

Figure 3-6: Simple Ada Binary Semaphore Task and a Periodic Client Task

Unfortunately, using a semaphore task for solving the INS synchronization problem can lead to undesirable runtime behavior, such as missed deadlines. For example, assume that the semaphore task in Figure 3-6 (call it S) controls access to a block of shared data that is used by all of the INS periodic tasks. In our example, further assume that the priority of S is set to be higher than P_1, our highest priority periodic task, to ensure that none of the periodic client tasks can preempt S.[13] We will also assume that each periodic task's critical section

[13]The priority of S is set high, because if it were not, priority inversion would be possible if S did not loop back around (after the Exit_Critical_Region accept) prior to another client call. Of course, this only removes one potential source of priority inversion.

consumes 0.5 ms of execution time. Under these conditions, our higher priority periodic tasks can be blocked by lower priority tasks for an unnecessarily long time. Suppose, for instance, that the phasing of all our periodic clients is such that immediately after one client attempts to lock the semaphore, the next higher priority client becomes ready to execute. In such a situation, the periodic clients will be activated and thus make their entry calls to Enter_Critical_Region in reverse priority order, i.e., a calling order of P_6, P_5, P_4, P_3, P_2, P_1, and the following could happen:

1. P_6 is granted the lock on the shared data (i.e., the call by P_6 to Enter_Critical_Region is accepted) through S at time $t = 0\,ms$.

2. Immediately after P_6 completes its rendezvous with S and enters its critical section, P_5 preempts the critical section of P_6 and attempts to lock the resource; S is not ready to accept another Enter_Critical_Region call since it must first accept a call to Exit_Critical_Region, so P_5 is placed in the corresponding entry queue.

3. For $i = 4,3,2,1$, P_i preempts the critical section of P_6 immediately after the attempted rendezvous of P_{i+1} with S; P_i is denied the lock and is placed at the end of the FIFO entry queue.

In this example, the entry queue for Enter_Critical_Region will contain P_1 through P_5 in reverse priority order. Sometime after time $t = 0.5\,ms$ when P_6 completes its critical section, the next client task in the entry queue, namely P_5, will be granted access to the resource. The service requests will be handled in FIFO order and consequently P_1 will have to wait for P_5 through P_2 to complete their critical sections before it will be granted the lock on the resource. The earliest P_1 could execute its critical section would be sometime shortly after time $t = 2.5\,ms$ since it would be blocked for a total of 2.5 ms waiting for the other periodic clients to complete their critical sections. Clearly, with a period of 2.56 ms and 0.70 ms of compute time (see Table 3-3), P_1 will miss its first deadline as a result of this blocking time caused by this tasking design.

The blocking time of P_1 could be even longer if we introduce another periodic task, P_{new}, that does not use S and whose priority is between that of P_1 and P_2. Assuming that the execution time of P_{new} is 3 ms and the same phasing exists for this new task set, P_1 will be blocked an additional 3 ms because of the execution of P_{new}. The three steps discussed above still apply, but now P_{new} will preempt P_6, and thus, P_6 will not complete its critical section until shortly after time $t = 3.5\,ms$. Consequently, the earliest that P_1 can now execute its critical section would be sometime shortly after time $t = 5.5\,ms$, since it is being blocked by P_{new} for an additional 3 ms.

On the surface our semaphore (synchronization) task, S, looks innocuous. But as we have illustrated, this type of design can lead to arbitrarily long blocking time and undesirable runtime behavior. The next subsection discusses a design approach that bounds and minimizes this blocking time caused by priority inversion.

Server Tasks That Emulate the Priority Ceiling Protocol

The essence of our second design approach for solving the INS synchronization problem is to use Ada server tasks to model semaphores as discussed in [16], and, further, to design and implement these server tasks so that their runtime behavior emulates the priority ceiling protocol (PCP)[14] [7, 16]. The PCP guarantees that:

1. The execution of a high-priority client task can be delayed by at most one lower priority client task per server call.

2. A deadlock cannot occur as long as a task does not call itself.

3. Blocked calls to the same server task are serviced in order of priority [16].

For convenience, we will denote these properties as PCP1, PCP2, and PCP3.

```
task body Server is
begin
  loop
    select
      accept Service_1 do
        Perform_Service_1;      -- Critical region #1
      end Service_1;
    or
      accept Service_2 do
        Perform_Service_2;      -- Critical region #2
      end Service_2;
    or
           . . .

      accept Service_n do
        Perform_Service_n;      -- Critical region #n
      end Service_n;
    end select;
  end loop;
end Server;
```

Figure 3-7: General Structure of an Ada Server Task

Figure 3-7 illustrates the canonical form of a priority ceiling server task written in Ada. Notice the difference between the semaphore task of Figure 3-6 and the server task of Figure 3-7. In the latter model, the critical section code is within the accept body for each respective service and not within the client's body.

The priority ceiling of a server task is defined as the highest priority of its client tasks, i.e., the highest priority of tasks that can call the server directly or indirectly. To properly emulate the priority ceiling protocol, our approach must guarantee that the following condition always holds:

[14] PCP emulation is necessary for the cases in which the runtime system does not implement the protocol directly.

Any entry call from a client task T to a server task S can be accepted if and only if the priority of T is higher than the highest priority ceiling of *all* other servers executing on behalf on clients other than T [16].

By following the rules listed below, one can emulate the PCP protocol in Ada applications:[15]

1. Set the priority of server tasks to be equal to their priority ceiling plus 1 by using pragma PRIORITY.[16]

2. Implement server tasks using the structure shown in Figure 3-7.

Rule 1 guarantees that all client requests are serviced without the possibility of being preempted by any other of the server's clients. Consequently, once a client enters the rendezvous with the server, that task will be the highest priority client eligible to execute; thus, no other client can make a call to that server. This runtime behavior also guarantees that no clients will be blocked on a server call, so property PCP3 is always preserved. Rule 2 imposes a task coding style which yields a structure that models the notion of critical regions guarded by a semaphore, thus allowing us to apply real-time scheduling theory based on semaphores to our system of Ada tasks [7, 16].

Assuming non-suspending server tasks, following Rules 1 and 2 in the context of Ada's scheduling rules will ensure that the fundamental condition of PCP stated above is always met.[17] For example, suppose we have three client tasks (from highest to lowest priority: P_1, P_2, P_3) and two server tasks (S_1, S_2). Assume P_1 calls S_1 and P_2 and P_3 call S_2. Rule 1 implies the following priority ordering: S_1, P_1, S_2, P_2, P_3. Notice that since the priority of P_1 is greater than the priority ceiling of S_2, the priority of P_1 is higher than the priority of S_2. Suppose that S_2 is executing on behalf of P_3 and then P_1 and P_2 both become ready to run. In this case, P_1 will preempt the execution of S_2 and be able to complete all of its execution, including its call to S_1. Under these circumstances, as would be the case for an actual PCP runtime, P_1 can preempt the execution of S_2 because its priority is higher than the highest priority ceiling of all other servers (i.e., S_2) executing on behalf of clients (i.e., P_3) other than P_1. After P_1 completes its execution, S_2 will resume its execution on behalf of client P_3. As soon as this service has completed, P_2 will begin executing by preempting P_3, showing how our Rule 1 works in enforcing property PCP1. Finally, knowing that the underlying condition of the PCP is upheld by following our guidelines, it follows that property PCP2 must also hold.

---

[15] Assuming that the server task(s) do not execute any statements which would cause them to suspend (e.g., synchronous I/O operation).

[16] In practice, Ada implementations vary with respect to their scheduling behavior for tasks with equal priority. Therefore, to ensure that runtime behavior is consistent with the expectations of rate-monotonic theory across differing Ada implementations, server tasks should be assigned priorities which do not conflict with other client tasks' priorities. Because the server's priority must be at least that of its highest priority client T_H, we recommend assigning the server task a priority one larger than the priority of T_H.

[17] A third rule could be added for dealing with server tasks which suspend: implement a prioritized server task; i.e., a server task that services client requests based on their assigned priority rather than in FIFO order. Since the server tasks can potentially suspend during a service call, resulting in client service requests queuing in their order of arrival, prioritized entry queues are necessary. There are numerous approaches to implementing a prioritized server in Ada using server tasks with entry families [3, 6]. Unfortunately, this approach will not emulate the PCP for all task sets. In particular, in the presence of nested server calls it is possible to get a deadlock using this approach.

Let's now reexamine the INS synchronization problem. For the sake of the example, assume that the periodic client tasks either read or write the shared data during their respective critical sections. Assuming that the access to the shared data cannot cause any task suspension, we want to construct a PCP server task which will service both read and write requests from the periodic clients using the guidelines outlined above. As before, we will set the server's priority to be higher than the priority of P_1; in particular, we will set its priority to 1 more than the priority of P_1 (Rule 1). We can construct a PCP server task and corresponding periodic client as illustrated in Figure 3-8 (Rule 2). Notice that this server task is really a monitor that protects the shared Results_Table defined in the task body.

```
task body Results_Table_Monitor is
  Results_Table : Results_Table_Type;
begin
  loop
    select
      accept Read_Service (...) do
        Do_Critical_Section_Read;
      end Read_Service;
    or
      accept Write_Service (...) do
        Do_Critical_Section_Write;
      end Write_Service;
    end select;
  end loop;
end Results_Table_Monitor;

task body Periodic_Client is
begin
  loop
    accept Activate;

    Results_Table_Monitor.Read_Service(...);
    Do_Some_More_Work;

  end loop;
end Periodic_Client;
```

Figure 3-8: PCP Server Task and a Periodic Client Task

The runtime behavior of our task set example will now change when compared with the semaphore task of the previous section. Assuming the original INS task set without P_{new}, P_6 will still be granted the lock at time $t = 0\,ms$. However, the other periodics—although they will have become ready to execute while P_6 is executing its critical section—will not run, since the Results_Table_Monitor task has higher priority. P_6 will complete its critical section at time $t = 0.5\,ms$ as before and P_5 through P_1 will be ready to run. Shortly after time $t = 0.5\,ms$, the next highest priority, ready to run task, namely P_1, will execute (remember, using the tasking structure of Figure 3-6, P_1 did not execute until after time $t = 2.5\,ms$). In this case, P_1 can meet its first deadline at $t = 2.56$; previously it could not.

Analyzing the INS task set including P_{new} shows that the above runtime behavior with respect to the effects of blocking on P_1 still applies. P_1 will start executing its critical section

sometime after time $t = 0.5\,ms$. P_{new} is assumed to be ready to run prior to time $t = 0.5\,ms$ based on our phasing, but it will be blocked by the Results_Table_Monitor executing on behalf of P_6.

As these two examples illustrate, using the PCP emulation approach, the INS client tasks in our example will start their execution in priority rather than FIFO order, thereby minimizing the blocking time caused by priority inversion caused by task synchronization. Furthermore, this approach reduces the number of rendezvous (from two to one) for accomplishing task synchronization in comparison with the Semaphore task model.

3.2.2. Analyzing the Schedulability Effects of Server Tasks

The previous section presented specific guidelines for constructing server tasks in Ada such that the properties of the priority ceiling protocol would be preserved. This section presents analytical techniques that may be used to understand and predict the behavior of systems in which periodic and server tasks interact.

We will begin by demonstrating techniques for analyzing the effect of server tasks on the schedulability of a task set. Then we will briefly discuss the additional complexity associated with analyzing the effects of server tasks which may suspend themselves.

Including Blocking Time in the Analysis

Server tasks introduce blocking time. To analyze the effect of server tasks on schedulability, the analytical techniques must be able to treat blocking time as well as execution time. General analytical techniques that treat both blocking time and execution time are available. In this section, we will apply the generalized techniques to the case in the INS where the six periodic tasks obtain mutually exclusive access to shared data through the Results_Table_Monitor server. Furthermore, we will build on our earlier analysis of the task dispatcher overhead.

When we apply the techniques of [16] to the problem of determining whether or not the six periodic tasks in the INS task set are schedulable in the presence of the Results_Table_Monitor server, we end up with the series of inequalities shown in Figure 3-9. The inequalities express sufficient conditions for the periodic tasks to be schedulable. There are six inequalities, one for each periodic task. Each inequality is to determine whether or not a periodic task can accommodate its own execution time, preemption time caused by all higher priority periodic tasks, and blocking time due to all lower priority tasks. We assume that each C_i includes the execution time of the server on behalf of the periodic client. In each inequality except the last one, a B_i term represents the worst-case blocking time for P_i due to lower priority tasks using the server. Since P_6 is the lowest priority periodic task, it cannot be blocked by a lower priority periodic task. For the periodic tasks to be schedulable, all of the inequalities shown in Figure 3-9 must be satisfied.

With $C_i' = C_i + D_e$,

$$\frac{C_1'}{T_1} + \frac{5D_b}{T_1} + \frac{B_1}{T_1} \leq 1(2^{1/1} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{4D_b}{T_2} + \frac{B_2}{T_2} \leq 2(2^{1/2} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{3D_b}{T_3} + \frac{B_3}{T_3} \leq 3(2^{1/3} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{2D_b}{T_4} + \frac{B_4}{T_4} \leq 4(2^{1/4} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{C_5'}{T_5} + \frac{1D_b}{T_5} + \frac{B_5}{T_5} \leq 5(2^{1/5} - 1) \text{ and}$$

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \frac{C_3'}{T_3} + \frac{C_4'}{T_4} + \frac{C_5'}{T_5} + \frac{C_6'}{T_6} \leq 6(2^{1/6} - 1)$$

Figure 3-9: Sufficient Conditions for Periodic Tasks to Be Schedulable in the Presence of the Results_Table_Monitor Server

To evaluate the inequalities, we must determine the value of each B_i in the equation. P_5 can be blocked when the server is executing on behalf of P_6 and when the task dispatcher activates P_6. Thus, if S_i represents the execution time of the Results_Table_Monitor server on behalf of periodic client P_i, then $B_5 = S_6$. Note that the blocking time caused by the task dispatcher has already been accounted for in the inequalities. P_4 can be blocked when the server is executing on behalf of P_5 or P_6 and when the task dispatcher is activating P_5 and P_6. Therefore, $B_4 = max(S_5, S_6)$. Values for B_3, B_2, and B_1 can be determined in the same manner.

Table 3-4 contains a summary of the various parameters that describe the periodic tasks and the Results_Table_Monitor server. Substituting values from Table 3-4 into the inequalities of Figure 3-9, we find that all of the inequalities evaluate to true. Therefore, the periodic tasks are still schedulable even in the presence of the blocking time introduced by the Results_Table_Monitor server. Had any one of the inequalities evaluated to false, we would not have concluded that the task set was unschedulable. Then we would have proceeded to apply the technique shown in [10] that uses numerical methods to determine whether a timeline exists such that all of the tasks will meet their deadlines.

$D_e = D_b$ = 0.2 ms

Task (P_i)	Period (T_i)	Exec. ($C_i + D_e$)	Server (S_i)	Block (B_i)	Figure 3-9 (Left Side)	Inequality (Right Side)
P_1	2.56 ms	0.7 ms	0.45 ms	0.60 ms	0.70	1.00
P_2	40.96 ms	5.2 ms	0.60 ms	0.52 ms	0.42	0.82
P_3	61.44 ms	15.2 ms	0.52 ms	0.44 ms	0.66	0.77
P_4	983.04 ms	30.2 ms	0.44 ms	0.34 ms	0.68	0.75
P_5	1024.0 ms	50.2 ms	0.25 ms	0.34 ms	0.73	0.74
P_6	1280.0 ms	1.0 ms	0.34 ms	0.00 ms	0.73	0.73

Table 3-4: Runtime Characteristics of Periodic Tasks Including Task Dispatcher and Results_Table_Monitor Server

Note that the following inequality is a sufficient condition for all of the inequalities shown in Figure 3-9 to be true [16]:

$$\frac{C_1'}{T_1} + \frac{C_2'}{T_2} + \cdots + \frac{C_6'}{T_6} + max\left(\frac{(B_1 + 5D_b)}{T_1}, \frac{(B_2 + 4D_b)}{T_2}, \frac{(B_3 + 3D_b)}{T_3}, \frac{(B_4 + 2D_b)}{T_4}, \frac{(B_5 + 1D_b)}{T_5}\right) \leq 6(2^{1/6} - 1)$$

Thus, the inequality above could have been used as a quick test before applying the inequalities in Figure 3-9. Had the quick test evaluated to true, then the six inequalities in Figure 3-9 would also have been true. But if the quick test had failed, then the six inequalities in Figure 3-9 would have to have been evaluated in turn. In other words, the single inequality above is a sufficient condition for the inequalities in Figure 3-9 to be satisfied, but it is not a necessary condition. The purpose of the single inequality is to try to save time when testing for schedulability.

Illustrating an Effect of Task Suspension

There are many cases in which a server task may suspend. For example, tasks P_3 and P_4 in the INS task set send messages to an external computer system (ECS). Message sending services that ensure mutually exclusive access to the communications port are provided by a server called the Communications_Controller. Since the communications protocol between the INS and the ECS involves handshaking and the I/O is interrupt-driven, the Communications_Controller must suspend routinely while waiting to hear back from the ECS.[18]

Rate-monotonic scheduling theory assumes that a task which is eligible to run will indeed execute on the CPU until either the task completes its designated duty or the task is

[18]Nominal message transmission sequence is as follows: (1) INS sends start-of-message to ECS, (2) ECS replies by returning ready-to-receive to INS, (3) INS sends packet of data words followed by end-of-message to ECS, (4) ECS validates the data packet and sends acknowledgement back to the INS.

preempted by a higher priority task which is also eligible to run. In the case described earlier, synchronous I/O with the ECS leads to a violation of this basic assumption since the Communications_Controller gives up the CPU before completing its designated duty. In general, when a periodic task (or a server task executing on behalf of a periodic client) suspends in this manner, we refer to the period of time that the task is suspended as *idle time*.

When analyzing the schedulability of a periodic task that may suspend, idle time must be accounted for in the same manner as execution time, even though the task that suspends does not use the CPU during the idle time. Thus, the C_i term for the task should include both execution time and idle time. But the analytic treatment of idle time does not stop there. Idle time may also have an adverse effect on lower priority tasks. This adverse effect is counter-intuitive since it would seem that lower priority tasks should be able to take advantage of a higher priority task's idle time. We will illustrate this adverse effect by way of an example.

Consider the three periodic tasks shown in Figure 3-10a. The three tasks are schedulable for all possible phasings, given their execution times and periods.[19] This particular phasing is used because it will point out the effect of idle time on lower priority tasks. In Figure 3-10a, none of the tasks can suspend. Note that P_2 requires 3 units of execution time and that P_3 finishes well before its deadline at time $t = 28$. Now we will introduce idle time and try to predict the results. Let us hypothesize that P_2 still requires just 3 units of execution time, but that it exhibits the following runtime behavior: P_2 executes for 2 units, suspends for 1 unit, and then executes for 1 more unit. Our intuition is that the three tasks should still be schedulable. After all, we have not increased C_2; we have only deferred part of it until later in T_2. Our intuition also suggests that P_2 will complete later in T_2 than it had before and that P_3 will complete sooner in T_3 than it had before. Without idle time, P_3 had to wait until both P_1 and P_2 had finished before it could use the CPU. With idle time, P_3 has to wait only until P_1 finishes and P_2 executes two units before it can use the CPU.

[19] To verify this claim, draw a timeline in which all three tasks are activated at the same time. The timeline will show that all three tasks will meet their first deadlines. Therefore, they will meet all of their deadlines [12].

Figure 3-10: An Example of One Effect of Idle Time on Lower Priority Tasks

Figure 3-10b illustrates the effect of the idle time of P_2 on P_3. Obviously, our intuition was wrong. Not only does P_3 fail to finish early, it fails to meet its deadline at time $t = 28$; the three tasks are no longer schedulable. Essentially, P_3 is penalized because part of C_2 is deferred until later in T_2. This effect is known as the *deferred execution penalty* [11, 15]. In our example, this can be seen by observing in Figures 3-10a and 3-10b that T_3 is 24 units long and that one of the three tasks is executing during each of the 24 units. In Figure 3-10a, P_1 executes for 5 of the 24 units, P_2 executes for 12 of the 24 units, and P_3 executes for 7 of the 24 units. In Figure 3-10b, P_1 still executes for 5 of the 24 units but P_2 executes for 13 of the 24 units, leaving only the remaining 6 units for P_3.

The complete effect of idle time on the schedulability of a task set is not yet fully understood. Therefore, only pessimistic worst-case schedulability bounds are available [15]. A complete understanding of the effects of idle time is very important in order for the scheduling theory to be generally applicable.

The problems associated with idle time, while not new, have been brought to the forefront by our work on the INS. In fact, one of the main objectives of the INS development was to expose problem areas for which treatment was lacking or ad hoc. The SEI REST and RTSIA projects and CMU ART project will collectively address the idle time problem in the near term.

3.3. Aperiodic Tasks

Many real-time systems are driven by the need to respond to aperiodic events. Considerable theoretical work has been done in this area [11]. This work provides a conceptual framework for analyzing the schedulability of a system with aperiodic processing requirements. However, many questions remain unanswered regarding the construction of analyzable models for handling aperiodic tasks in Ada. This section summarizes several important concepts for understanding how to incorporate aperiodicity into a schedulability model.

Server tasks execute only on behalf of other tasks. Periodic tasks are tasks with a periodic timing requirement that execute on their own behalf. *Aperiodic tasks* also execute on their own behalf, but do not have periodic timing requirements; however they may have response time requirements. Typically, aperiodic tasks execute in response to an event or perform a low-priority background function.

Given a set of periodic tasks and a response time requirement for an aperiodic task, we must ensure that aperiodic response time requirements are met, and at the same time, understand the effect of aperiodic tasks on the schedulability of the remainder of the task set. The concept of an aperiodic execution manager [11], also called an aperiodic server, helps to address both of these issues. An aperiodic execution manager is conceptually a periodic task[20] that doles out execution time to aperiodic tasks. This allows aperiodic utilization to be

[20] An aperiodic execution manager may manifest itself as an Ada task or an algorithm in the runtime system. From the point of view of the schedulability analysis, it is viewed as a periodic task.

analyzed in a periodic framework, assuming that there is a known, minimum separation of aperiodic events. See [11, 18, 19] for a discussion of the various strategies for managing aperiodic execution. Given this conceptual framework, let us address three types of aperiodic tasks:

- An aperiodic task that has a hard deadline for its response to an event.
- An aperiodic task that has either a soft deadline or no deadline for its response to an event.
- An interrupt handler, which executes at a very high priority regardless of its response time requirement.

The basic strategy for meeting a hard deadline requirement is to give the aperiodic execution manager (that serves such a task) a high enough priority. Clearly, if the manager has the highest execution priority in the system (assuming that there are no interrupts) then the aperiodic task will meet its deadline if its execution time is less than its response time. To analyze the schedulability of the entire system, the aperiodic execution manager is factored in as one of the periodic tasks. Preserving schedulability of the periodics may not permit the aperiodic manager to execute at the highest priority in the system. Thus, part of the schedulability analysis of a system of tasks that has both periodics and aperiodics will involve determining a priority for the aperiodic manager that meets the response time requirement and still allows the periodic tasks to meet their deadlines. If this is impossible, the requirements are untenable; the requirements must be relaxed or a system overload could occur. If it is possible, then periodic deadlines and response time requirements are guaranteed. In both cases we have predictable results.

If an aperiodic task does not have a hard deadline but it is still advantageous to service aperiodic activity in a timely fashion, the aperiodic execution manager is still a viable approach. The philosophy in this case is slightly different. For aperiodic tasks with hard deadlines, one may be willing to sacrifice low-priority periodic deadlines. However, in the absence of a hard deadline, the goal is to take advantage of spare CPU capacity in a manner that is advantageous to aperiodic tasks. Using the same strategy described above, the aim is to find the highest priority level in the rate-monotonic pecking order that achieves all periodic deadlines and minimizes aperiodic response time.

The first two cases required that we find a suitable priority for the aperiodic manager. However, interrupts are handled at a priority higher than all Ada task priorities; thus, a high priority is imposed on this type of aperiodic processing independent of its response time requirement. The approach for analyzing this case is slightly different. Once again we assume that aperiodic interrupts are separated by a minimum time interval. In the worst case, interrupts occur periodically, with that minimum separation interval as the period. There is a rate-monotonic priority associated with this period. For all tasks with a priority higher than this priority, the interrupt execution is treated as blocking time. For all lower priority tasks, this interrupt is treated as a higher priority periodic task. Assuming that this interrupt is neither masked nor interrupted, response time is not an issue in this case.

3.4. Recommendations

Recent scrutiny of Ada by the real-time software community has identified inadequacies in the language with respect to expressing and handling real-time behavior [4, 5, 6, 13, 14]. These identified limitations include: nondeterministic selection of open alternatives, FIFO entry call queues, lack of an adequate time abstraction for implementing real-time requirements, and problems with using the delay statement to achieve periodic processing. In this section, we offer several guidelines for engineering real-time systems in Ada based on our own experiences with the INS application. These guidelines are intended to demonstrate how to design Ada real-time systems in ways that circumvent the problems mentioned above and enhance the predictability of runtime behavior. Our recommendations for managing concurrency in Ada are summarized below.

1. **Design and implement analyzable software systems.** To fully understand and be able to reliably predict the runtime behavior of a real-time Ada application, you must design and implement your system in a manner which is amenable to the application of real-time scheduling theory. Generally, this means that you should design and implement your application by using language features in well understood and analyzable ways. Without this type of engineering approach, there are no guarantees regarding the execution timing behavior of your real-time Ada application.

2. **Understand how to implement analyzable periodicity.** Implementing analyzable periodicity depends on task priority assignment and the design paradigm used for achieving periodic processing. The RMA should be used as a basis for task priority assignment, since it is an optimal fixed priority algorithm, because a solid body of scheduling theory is based on this algorithm, and because it is applicable to Ada. We recommend the Delay_Until model for achieving periodicity, provided that the runtime system's timing resolution meets the application's requirements. If the functionality that is needed to support the Delay_Until model is not available or does not provide the required timing resolution, the Dispatcher model is a suitable alternative.

   Also, be aware of effects that can alter periodicity. For the Delay_Until and Dispatcher models, it is very important to understand the system's interrupt structure. In particular it is important to know if, when, and for how long the underlying real-time clock's interrupt is masked. If models other than these two are used (for example, using the delay statement in the manner suggested in section 9.6 of the LRM), then awareness of the potential for delay jitter is important.

3.

4. **Minimize priority inversion by using the priority ceiling protocol.** Be aware of priority inversions that may be caused by task interactions. Since high-priority, high-frequency tasks must be able to accommodate additional blocking time that results from priority inversions, the overall schedulability of a task set can be greatly affected by task interactions. While the priority inversions cannot be completely eliminated, they can be minimized by creating

server tasks that emulate the priority ceiling protocol (assuming, of course, that the runtime system does not directly support the priority ceiling protocol).

5. **Understand the interactions between your application and the Ada run-time system.** In order to perform a comprehensive schedulability analysis of your application, its interaction with the Ada runtime system needs to be well understood. Generally, this means that you should identify all potential sources of execution time, preemption time, and blocking time in both your application and the Ada runtime system. In particular, be aware of the effects of context switching and idle time.

4. Conclusion

Experimenting with design and implementation alternatives in the context of an actual real-time application (the INS) has led us to several conclusions concerning the development of real-time systems in Ada:

- Ada allows the real-time programmer to handle devices and concurrency at a relatively high level of abstraction; however, these language features do not relieve the programmer of low-level data representation concerns and schedulability concerns.

- Applying analytical methods in concert with experimentation and prototyping can provide important insights about the system's runtime behavior throughout the life cycle of a real-time system.

- Given analytical methods and coding guidelines, Ada has the potential to be successfully used in real-time systems.

The Ada language has specific features that address the need to control devices and manage concurrency. Representation specifications allow the programmer to interact with a device by using record types to model the memory map of its registers. However, the responsibility for ensuring proper data representation is still in the hands of the programmer. Compiler optimization may thwart attempts to control a device by eliminating important steps in a sequence of device control instructions. Awareness of the relationship between the bit and byte numbering schemes of the underlying hardware and the model used by the Ada implementation is also important.

Ada tasking provides a conceptual model for managing concurrency. This model allows implementation of concurrency at a relatively high level of abstraction. However, raising the level of abstraction for real-time programming is beneficial only if the resulting behavior is predictable. Using tasking in a manner that results in predictable runtime behavior requires careful analysis. The analytical methods presented in this report offer a framework for performing such an analysis.

Early exploration of high risk areas such as device interfaces and system timing behavior exposes problem areas early: uncovering inconsistent or unreasonable requirements, detecting potential performance problems, and making hardware/software tradeoffs. Continual iteration through cycles of analysis, implementation (prototyping), and experimental verification are important activities that need to be done throughout the life cycle of a real-time system. For example, these activities can provide feedback during requirements analysis, help to determine the feasibility of design alternatives, and facilitate maintenance. As the design progresses, system behavior can be modeled with increasing accuracy. A system may be incrementally developed with schedulability analysis applied at each step. Therefore, a skeleton system can be operating early in the development phase, and the system runtime behavior will always be predictable. To satisfy the need for continual analysis, the system must be instrumented to collect data about itself (e.g., performance

data and scheduling data). This requires that instrumentation be designed into the system from the beginning.

Finally, when complementary analytical and experimental methods are used in concert with implementation guidelines for Ada real-time features, Ada has the potential to be employed successfully in the real-time domain. The analytical methods and associated Ada techniques presented in this article provide a foundation for engineering real-time systems in Ada today. The REST, RTSIA, and ART Projects intend to continue extending the scheduling theory to ensure its general applicability and to promulgate the resulting methods and guidelines.

References

[1] Borger, M.W.
VAXELN Experimentation: Programming a Real-Time Clock and Interrupt Handling Using VAXELN Ada 1.1.
Technical Report CMU/SEI-87-TR-29, ADA188100, Software Engineering Institute, October 1987.

[2] Borger, M.W.
VAXELN Experimentation: Programming a Real-Time Periodic Task Dispatcher Using VAXELN Ada 1.1.
Technical Report CMU/SEI-87-TR-32, ADA200612, Software Engineering Institute, October 1987.

[3] Burns, A.
Using Large Families for Handling Priority Requests.
Ada Letters 7(1):97-104, January 1987.

[4] Burns, A., Wellings, A.J.
Real-Time Ada Issues.
In *Proceedings of the International Workshop of Real-Time Ada Issues*. November 1987.

[5] Cornhill, D., Sha, L.
Priority Inversion in Ada.
Ada Letters 7(7):30-32, November 1987.

[6] Cornhill, D., Sha, L., Lehoczky, J.P., Rajkumar, R., and Tokuda, H.
Limitations of Ada for Real-Time Scheduling.
In *Proceedings of the International Workshop of Real-Time Ada Issues*. November 1987.

[7] Goodenough, J. B., Sha, L.
The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks.
In *Proceedings of the International Workshop of Real-Time Ada Issues*. June 1988.

[8] Landherr, S.F., Klein, M.H.
INS Behavioral Specification.
Technical Report CMU/SEI-87-TR-33, ADA200604, Software Engineering Institute, June 1987.

[9] Klein, M.H., Landherr, S.F.
INS Simulator Program: Top-Level Design.
Technical Report CMU/SEI-87-TR-34, ADA200605, Software Engineering Institute, July 1987.

[10] Lehoczky, J. P., Sha, L., and Ding, Y.
The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior.
Technical Report, Department of Statistics, Carnegie Mellon University, 1987.

[11] Lehoczky, J. P., Sha, L., Strosnider, J.
Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment.
In *IEEE Real-Time System Symposium*. December 1987.

[12] Liu, C.L., Layland, J.W.
Scheduling Algorithms for Multi-Programming in a Hard-Real-Time.
JACM 20(1):46-61, January 1973.

[13] Locke, C.D., Vogel, D.R.
Problems is Ada Runtime Task Scheduling.
In *Proceedings of the International Workshop of Real-Time Ada Issues*. November 1987.

[14] McCormick, F.
Scheduling Difficulties of Ada in the Hard Real-Time Environment.
In *Proceedings of the International Workshop of Real-Time Ada Issues*. November 1987.

[15] Rajkumar, R., Sha, L., and Lehoczky, J.P.
Real-Time Synchronization Protocols for Multiprocessors.
In *In Proceedings of the IEEE Real-Time Systems Symposium*. December 1988.

[16] Sha, L., Rajkumar, R., and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-time Synchronization.
Technical Report (CMU-CS-87-181), Department of Computer Science, Carnegie Mellon University, 1987.

[17] Sha, L., Goodenough, J. B.
Real-Time Scheduling Theory and Ada.
Technical Report CMU/SEI-89-TR-14, Software Engineering Institute, April 1989.

[18] Sha, L., Ralya, T., and Goodenough, J. B.
An Analytical Approach to Real-Time Software Engineering.
In *Software Engineering Institute Annual Technical Review*. 1989.

[19] Sprunt, B., Sha, L., and Lehoczky, J. P.
Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System.
Technical Report CMU/SEI-89-TR-11, Software Engineering Institute, April 1989.

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

[21] Volz, R. A., Mudge, T. N.
Instruction-Level Timing Mechanisms for Accurate Real-Time Task Scheduling.
IEEE Transactions on Computers C-36(8), August 1987.

Table of Contents

List of Figures

List of Tables