ADA in Control Applications - A Case Study

Mattsson, Sven Erik

FÖRSVARETS FORSKNINGSANSTALT
Huvudavdelning 2

FOA-rapport
C 20 -D8, E3
December 1981

ADA IN CONTROL APPLICATIONS - A CASE STUDY

Sven Erik Mattsson

Department of Automatic Control
Lund Institute of Technology
Box 725
S-220 07 Lund 7
Sweden

Antal blad 30

ABSTRACT

This report considers the Ada programming language in a control system application. A wind turbine system has been selected as an example.

The main part of this study is devoted to ascertaining how the design and system structuring are influenced and supported by Ada.

SAMMANFATTNING

Rapporten behandlar egenskaper hos programmeringsspråket Ada i en relerteknisk tillämpning. Som exempel har ett vindkraftverk valts.

Huvuddelen av arbetet ägnas åt att studera hur Ada i en reglerteknisk tillämpning påverkar och stöder konstruktion och strukturering av systemet.

Uppdragsnr: DR09

TABLE OF CONTENTS

1. INTRODUCTION

This report considers the properties of the Ada programming language [1] when implementing control systems. The study is carried through as a project to develop the software for a control system.

1.1. The plant

The plant to be controlled is a 3 MW wind energy conversion system (WTS-3) [2], to be built by Karlskronavarvet AB and Hamilton Standard. The wind turbine system consists of a horizontal axis downwind turbine with two blades mounted on a teetered hub. This wind turbine drives a 3-phase synchronous generator through a gear. The system is designed to supply power in parallel with other electrical generators to a large power utility grid. Power is controlled by changing the blade pitch angle. Actuation is by means of a hydraulic positioning system.

1.2. The control system

The control system contains three processors, called the supervisory processor, rotor processor and interface processor. The supervisory processor supervises the system and decides the mode of operation. Depending on the mode of operation, the rotor processor regularly calculates a reference value to the hydraulic positioning system. The interface processor handles operator communication. The three processors can communicate with each other via serial lines. This serial communication is handled by special I/O-processors. An I/O-processor communicates with its master via buffers in shared memory and interrupts.

The control system is representative as it contains the following components:

1) Several processors
2) Communication between plant and computer
3) Communication between computers
4) Communication between computer and operator
5) Time synchronization
6) Common data bases of limited sizes

The main control system requirement is of course that it controls energy production manner reliable and optimal. Since the system is in the developing stage and is subject to research, it must be possible to easily test and evaluate different control strategies. The need to easily exchange parts of the software implies that it should be extensively modularized with well-defined interfaces between the modules. To facilitate the test and evaluation of a control strategy, the software should provide good operator communication and logging facilities.

1.3. The aim of this study

The main part of this project is devoted to design software for the rotor processor, and to studying how the design and design procedure are influenced and supported by Ada. References to the Ada Manual [1] are given throughout the text. [1:2.3] means section 2.3 in the Ada Manual.

2. ROTOR PROCESSOR SPECIFICATION

Requirement analysis and specification are important stages in software development. The manufacturers of the wind turbine system have worked out detailed specifications for the rotor processor. The rotor processor should:

1. Handle the control loop; At predefined instants in time

    1. Sample measurements from the plant.
    2. Calculate a control signal.
    3. Output the control signal to the plant.

2. Regularly send a status flag word, the value of the control signal and error codes to the supervisory processor.

3. Regularly send measurements and states to the interface processor.

4. Change the mode of operation at the request of the supervisory processor.

5. Update parameters at the request of the interface processor.

This list extracts the essence of the problem and details are given when needed.

3. PROGRAM STRUCTURE

A program can show several forms of structure. In this case, interactions with external events and demands on response times are important factors which influence the program structure, therefore design starts by identifying those activities which have to be handled in parallel, and their interactions. Ada's tasks can be used to implement parallel activities, but also to implement facilities like mailboxes, buffers and monitors. In this discussion it is desirable to distinguish between these uses of tasks.

3.1. Processes

3.1.1. The control loop

The calculation of control signals is the main object of the rotor processor. In order to achieve good wind turbine performance, it is important that calculations are done on schedule. To avoid unnecessary delays, a high-priority process called Regul is introduced to sample measurements from the plant, calculate a control signal and output it to the plant.

3.1.2. Input/Output

The supervisory processor, the rotor processor and the interface processor communicate by sending messages to each other on serial lines. Each processor has an I/O-processor that handles the low-level part of the protocol. The rotor processor and its I/O-processor communicate via shared memory areas and interrupts.

It is natural to introduce one task to receive the incoming messages from the supervisory processor and one for those from the interface processor. However, they are not independent of each other, since the interrupts and a register in the shared memory are common. This can be solved by introducing a third task which acts as a multiplexer and forwards the interrupts to the proper task. Here, another approach is chosen. The task In_handler is set to receive all incomming messages from the I/O-processor and make them available for further handling in the rotor processor.

Sending is similar to reception. However, no interrupts are generated by the I/O-processor, but the status bits describing the output lines must be polled. The task Out_handler is introduced to forward messages to the I/O-processor from the rotor processor.

3.1.3. Communication with the supervisory processor

The orders from the supervisory processor to change the mode of operation and the sending of data to the supervisory processor form an essential part of the control and security systems and are time critical. The handling of them should have the same priority as the calculation of a new control signal. They are easy to handle. Changes in the mode of operation demand only a recalculation of the regulator states to achieve a bumpless transfer. This recalculation can be performed by Regul, since in practice it must be so short that it can be performed between two sampling instants. Consequently, it will be assumed that Regul itself can handle the messages from and to the supervisory processor.

3.1.4. Messages from the interface processor

The interface processor is introduced to make the system more flexible and easier to test, evaluate and maintain. Therefore messages from the interface processor are not time critical. It is not advisable to let Regul handle the messages, because it may delay the calculations of new control signals. A message containing new parameters takes time to handle. The parameters must be validated in some way or other. Checks against lower and upper limits are simple forms of validation. The operator and regulator may for convenience use different collections of parameters and the new parameters must then be converted before they can be used by Regul. Consequently, a new process must be introduced to handle the incoming messages. The order in which the messages should be handled depends both on their importance and their time of arrival. However, the problem should not be exaggerated. If many messages are queued up, the system cannot be said to work well. Here, a low-priority process called Opcom is introduced to handle the messages in order of arrival.

3.1.5. Messages to the interface processor

The interface processor collects and logs information about the system. The rotor processor should contribute by sending measurements and state. These data originate from Regul. They must be made available in some way or other and it is assumed that Regul should generate the messages to the interface processor.

3.1.6. Introduced processes

So far the following processes have been introduced:

Regul	Executes the control cycle; samples measurements from the plant, calculates a control signal, outputs it to the plant, generates messages to the supervisory processor and the interface processor and handles the messages from the supervisory processor and Opcom.

In_handler	Receives the messages from the I/O-processor and makes them available for further handling in the rotor processor.
Out_handler	Forwards the messages from the rotor processor to the I/O-processor.
Opcom	Handles and prepares the messages from the interface processor for Regul.

3.2. Inter Process Communication

Entry call and accept statements are the primary means of inter process communication in Ada [1:9.5]. Communication using global, unprotected variables is very unreliable and should not be used. The rendezvous concept has two asymmetric properties, which may influence the design. First, the caller of an entry must know the name of the task owning that entry, while the owner of the entry just needs to know that a caller expects some external interaction. However, there are no identification problems in this application, since the set of activities is small and fixed. Second, it is possible to wait for calls of several entries simultaneously [1:9.7.1], but only one entry call can be issued at a time. Consequently, the design procedure must consider which processes that have to wait for several events simultaneously. Notice, that this is the first influence from Ada on the design.

In_handler should make the incoming messages available for Regul and Opcom. Serial lines should be used efficiently, since they are often bottlenecks. Consequently, it is not suitable to let In_handler forward the messages by calling entries in Regul or Opcom, because they can only accept messages at certain times and In_handler may then be delayed. Two buffers (one for messages from the supervisory processor and one for messages from the interface processor) are needed. The buffers are made local to In_handler, since there is no reason for introducing separate buffer tasks. The time to serve an entry call and to return a message is so short that it does not influence the utilization of the serial lines. In the same way, the time to serve an interrupt is so short that it does not delay fetching of messages.

Out_handler can have local buffers in the same way as In_handler. Regul can forward the messages to the supervisory processor and interface processor by calling entries in Out_handler.

The handling of the controller parameters is a decision point. The parameters are used by Regul when calculating a new control signal, and Opcom must be able to update them when it receives new values from the interface processor.

One approach is to let the parameters be local to Regul and to let Opcom send messages to Regul when the parameters should be updated. Another approach is to handle them as a database in a monitor [11], where they can be accessed both by Regul and Opcom. These two approaches are discussed below.

3.2.1. The message passing approach

Consider the approach of having the parameters in Regul.

Opcom can fetch the messages from In_handler by itself. Opcom should wait for a message from the interface handler, handle it and make the result available to Regul. The suitability of letting Opcom itself hand over the new parameters by calling an entry in Regul depends on the frequency of messages from the interface processor, and on how often Regul accepts new parameters. It seems better to introduce a buffer to make communication between Regul and Opcom more asynchronous.

Regul should fetch messages both from In_handler and Opcom. There are two alternatives.

First, Regul can issue two conditional entry calls [1:9.7.2] every control cycle to see if any messages have arrived. The conditional entry call statement must be used with care and the use in cases like this can hardly be recommended. A negative result can mean two different things: there is no message available at the moment or the task is busy in a rendezvous with another task. (The second case is impossible, if the caller has lower priority than the called and if the program is running on a single processor, since, if two tasks have different priorities, the rendezvous is executed with the highest one [1:9.8].)

Second, tasks (messengers) can be introduced, which only fetch a message from one given source and forward it by calling a given entry of Regul. Regul can then wait for messages by using a select statement with a delay statement to the next sampling instant. These tasks are easy to implement and can be made generic with one parameter defining the type of messages and two subprogram parameters defining the source and destination. A generic package implementing a messenger can be found in Appendix 1.

The program structure is shown in Figure 1.

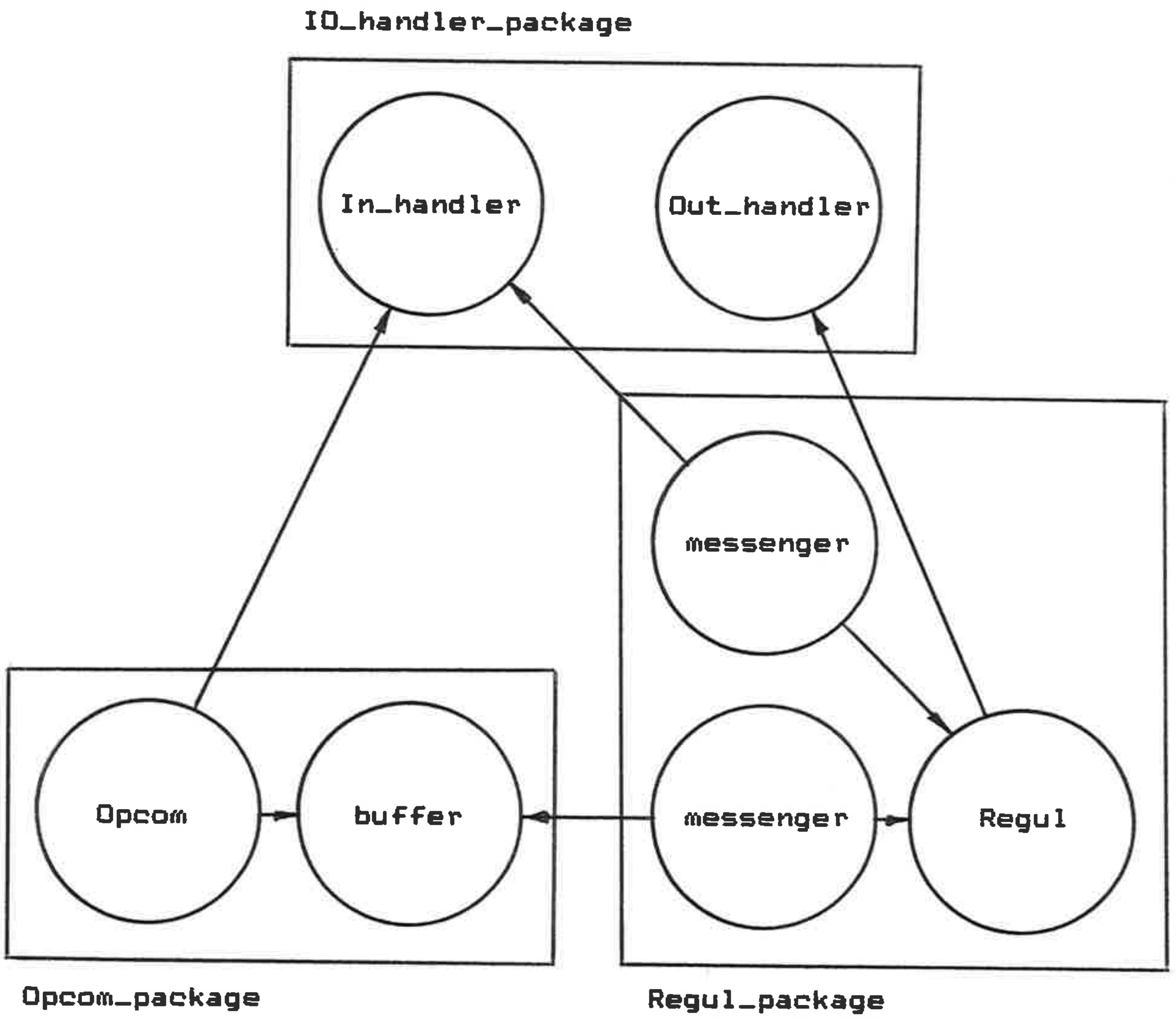

Figure 1: Program structure when using message passing. The squares denote packages, the circles denote tasks and the arrows indicate entry calls.

3.2.2. The data base approach

Let us now consider the approach of having the parameters in a monitor. A structure is given in Figure 2.

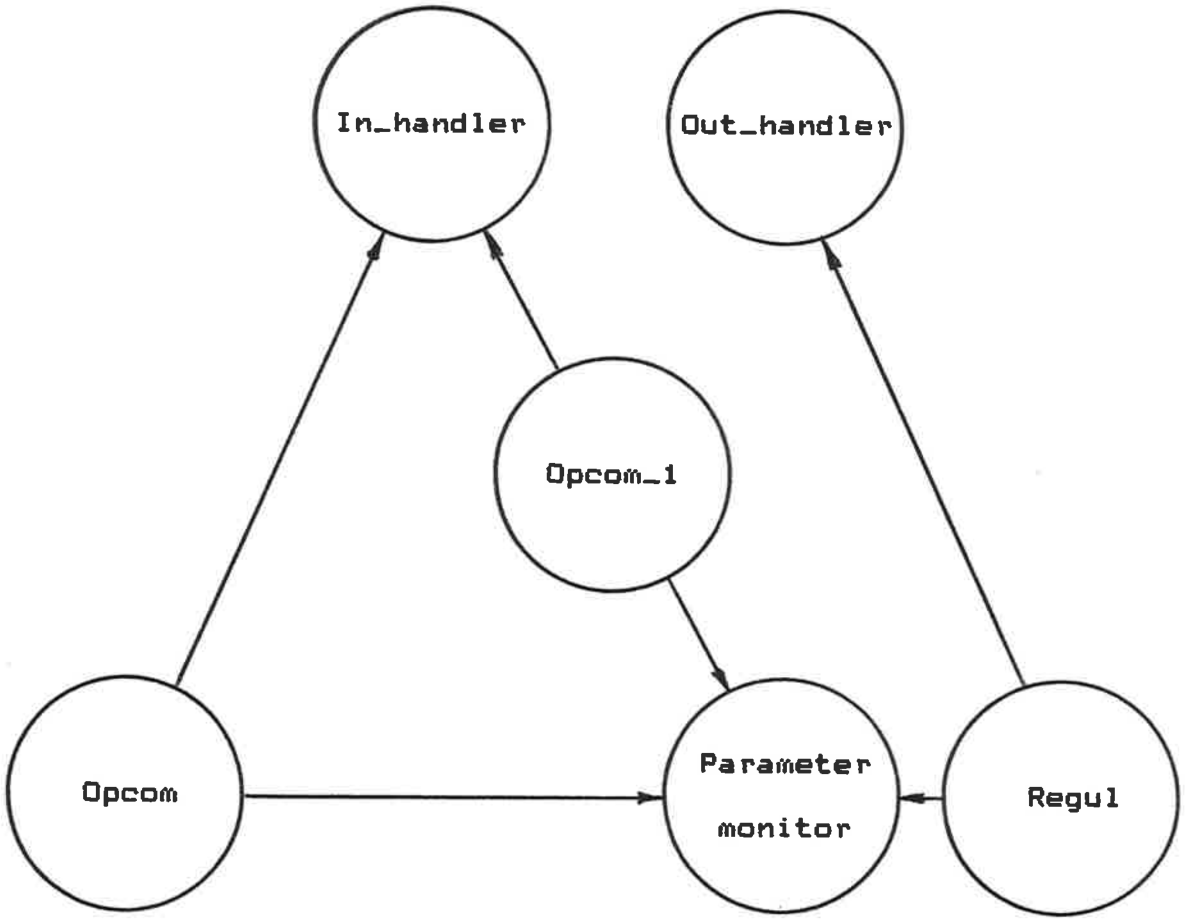

Figure 2: Program structure when using a data base. The circles denote tasks and the arrows indicate entry calls.

Opcom_1 should handle the messages from the supervisory processor and update the parameter which indicates the mode of operation. Parameter_monitor should contain an infinite loop with a selective wait statement [1:9.7.1] accepting entry calls from Opcom, Opcom_1 and Regul.

The synchronization between Regul and Parameter_monitor can be handled in a number ways. One simple way is to let Regul start the control cycle by fetching the parameters from Parameter_monitor. It is somewhat inefficient as the parameters must be copied every control cycle. This can be avoided by letting Regul execute the control cycle inside Parameter_monitor. Opcom and Opcom_1 will not be delayed unnecessarily since Regul should have higher priority. It is also possible to let Regul have its own set of parameters. Regul must then regularly check if the parameters in the monitor have been updated. If Opcom and Opcom_1 indicate an updating of the parameters by setting a flag in the monitor, this flag can be used as a guard to Regul's entry to the monitor. Regul can then call the entry using a timed entry call statement [1:9.7.3] with a delay to the next sampling instant. This means that Parameter_monitor should accept entry calls from Regul only when the parameters have been updated.

3.2.3. Discussion

The properties of the rendezvous concept forced us to introduce three extra tasks in Figure 1. Incidentally, the structure given in Figure 1 is a possible solution to the data base approach, if the buffer is interpreted as the monitor with the parameters.

The solution in Figure 1 has some minor advantages over that in Figure 2. First, the stategy of using messages for inter process communication is in harmony with the philosophy of sending messages between the processors. Second, messengers and buffers are standard elements. They are independent of the control algorithm. A parameter monitor is not a standard element in the same way, and is somewhat more difficult to make independent of the control algorithm. However, the choice between these two approaches in this case is mainly a question of taste. The structure in Figure 1 is chosen.

3.3. Organization of the Code

IO_handler, Opcom and Regul are formed as packages. See Figure 1. IO_handler_package is discussed in Section 5 and the package specification can be found in Appendix 6. The types of messages are assumed to be declared in a package called Message_types. Opcom_package should contain the task Opcom, and a buffer where the messages from Opcom could be fetched. Regul_package should of course contain the task Regul, and messengers which fetch messages from the I/O-handler and Opcom and forward them to Regul. Furthermore, a simple procedure, which is acting as the main program, is needed to put the pieces together. The programming of Opcom and Regul will not be discussed here.

To handle the buffering problems, generic buffer packages are provided. They are discussed in Section 4.

4. BOUNDED BUFFERS

Bounded buffers are common data structures and should be available in libraries. Use of standard elements should be encouraged, since it simplifies maintainance and increases portability and reliability. Here it is assumed that such packages are not already available, but have to be written. Below, two kinds of buffers are discussed; buffers for local use inside tasks, and buffers for use between tasks.

4.1. Buffers for Local Use Inside a Task

A generic package Local_buffer_package providing a buffer type local_buffer for use inside a task is given in Appendix 2 and is meant to be self-documenting. The body can be found in Appendix 3. The design is discussed below.

Two methods can be used for encapsulation of a buffer implementation: the package can form the buffer or it can define a buffer type. These two methods have different properties with respect to flexibility and reliability.

If the packages forms the buffer, it is possible to let the compiler guarantee the properties of the buffer, since the user can only operate on the buffer as specified in the visible part of the package. However, the method also implies that one package is needed for each buffer. This can force the user to duplicate subrograms, since packages cannot be passed as parameters to subprograms.

If the package defines a buffer type with the buffer size as a discriminant, this can be used to declare several objects of the same type and with different buffer sizes. If the buffer type is also specified as limited private, it is impossible for the user to operate on the buffer objects in other ways than those given in the visible part of the package specification. The compiler can then guarantee that the logical properties of an object expressed by the package specification is preserved. This assumes that the object is properly initialized. Unfortunately, it is not possible to define initialization routines that are invoked automatically, when an object of a limited private type is created. Simple cases can be solved by using explicit initialization of record components for record types, as is done in Appendix 2 for local_buffer, where the buffer is represented as an array. The problem can be solved in a general and reliable way by enclosing the needed data structure in a record together with a boolean variable that indicates if the record has been initialized. Explicit initialization of record components for record types [1:3.7] can guarantee that this variable has the initial value false. Every routine declared in the visible part of the package must then test this boolean before using the object, and initialize the object if the boolean variable is false. This method is inefficient and clutters the program text.

In Appendix 2, the package Local_buffer_package defines a buffer type. The initialization is left to the user. Before any use of a buffer variable he must initialize it by a call of Init_buffer. To preserve the property that the internal representation can be changed without influencing the user's program, all implementations should provide Init_buffer even if it is not needed.

The specification of the buffer types as limited private [1:7.4.2] means that neither assignment nor comparison for equality is available. This is important, since the semantics of these operations are dependent on the representation of the buffers. Consider the two types local_buffer_1 and local_buffer_2 (For simplicity, the buffer size is fixed.):

```
type local_buffer_1 is
  record
    in_p, out_p: natural := 1;
    count:       integer range 0..10 := 0;
    buff_area:   array (1..10) of elem_type;
  end record;

type local_buffer_2 is access local_buffer_1;
```

An assignment of a variable of type local_buffer_1 to another gives two different buffers with the same elements, while an assignment of a variable of type local_buffer_2 would mean that the two variables denote the same buffer. If an application needs to store and exchange buffer identities, it can do so by defining an access type designating the corresponding buffer objects and by using access values for identification purposes.

The current implementation of the buffers as arrays need no final processing when the scope containing the buffer variable is left. However, if the elements were stored in explicitly allocated storage, via new, it is not certain that the area is reclaimed automatically. It depends on the implementation. Destroy_buffer is introduced to handle this on the user-level.

4.2. Buffers for Use Between Tasks

A generic package Buffer_task_package providing a buffer type buffer_task for use between tasks is given in Appendix 4, and the body can be found in Appendix 5.

To guarantee mutual exclusion, it is implemented as a task monitoring an object of the type local_buffer. The task is given the highest priority to guarantee that it is not blocked by a low-priority process (if the two tasks have different priorities, the rendezvous is executed with the highest one [1:9.8]).

Tasks cannot be parameterized [1:9.1] and the buffer size is passed by Init_buffer in a rendezvous.

The task type is encapsulated in a package, because a task type can be neither generic [1:12] nor separately compiled [1:10]. There are several reasons for not hiding the representation of buffer_task. First, a task type can be viewed as a limited private type [1:9.2] and constitutes by itself an encapsulated type. Second, each entry constitutes exactly one operation on the buffer, so there is no need for interface procedures which call the entries in the correct order. Interface procedures, which just call one procedure, increase neither the readability nor the efficiency. Third, since the representation is known to the user, he can use conditional entry call [1:9.7.2] and timed entry call statements [1:9.7.3] directly. If the representation was hidden, the visible part had to provide procedures for this and they are not straightforward to implement. For example, even if the buffer is empty a procedure Get, specified as

```
procedure Get(e: out elem_type; out done: boolean);
```

must return a proper value of e to avoid a constraint_error exception at the return of the procedure [1:6.4.1]. Neither the compiler nor the run-time system is aware of the connection between e and done. Since elem_type is private it is not always possible to return a proper value (it is of course possible to have such a value as a generic parameter). There are three major ways of solving this problem. First, let the user provide a value by declaring e as a parameter of in out mode. Second, leave the procedure via a user-defined exception if the buffer is empty. Third, make the connection between e and done explicit like this:

```
type result_type(done: boolean := false) is
  record
    case done is
      when true  => e: elem_type;
      when false => null;
    end case;
  end record;

procedure Get(r: out result_type);
```

Now it is possible to assign a value to r inside Get even if the buffer is empty.

5. I/O-HANDLER

The IO_handler_package, which is given in Appendix 6, should handle the communication with the I/O-processor. It is made generic with respect to the message types, since it ought to be independent of them and make separate compilation [1:10] possible without knowing them. It is further assumed that the formats of the messages are properly specified by means of Ada's type representation specification concept [1:13], so that the same representation of a message is used in both ends. The reception of messages is discussed first and then the sending of messages.

5.1. Reception of Messages

The rotor processor and its I/O-processor communicate via shared memory areas and interrupts. The shared areas are described by the package Hardware_interface in Appendix 7. The I/O-processor puts incoming messages from the supervisory processor into f_S_area and those from the interface processor into f_I_area. The status of these areas is described by bits in S_status and I_status. One bit (mess_in) indicates that there is a message available. The rotor processor should clear this bit when the message is read to indicate to the I/O-processor that the area is free, and a new message could be input. The procedures Ack_S_mess and Ack_I_mess should be used to do this. When the I/O-processor has input a new message, it issues an interrupt in the rotor processor. The origin of the message is indicated by cmd_in. The procedure Get_interrupt reads cmd_in, and the status, and acknowledges the interrupt by clearing cmd_in. The shared memory areas are initialized by the I/O-processor.

5.1.1. Handling of interrupts in Ada

In Ada, interrupts are associated with entries [1:13.5]. An address specification links a hardware interrupt to an entry. The occurrence of an interrupt acts as an entry call issued by a task whose priority is higher than that of any user-defined task. The entry call may be an ordinary entry call, a timed entry call or a conditional entry call, depending on the type of interrupt and on the implementation. If data are explicitly supplied by the interrupt, they are passed to the associated entry as one or more in parameters. As pointed out in [3], translations of interrupts into ordinary entry calls move the buffering decisions for interrupts from the interrupt service task to the run-time system. Furthermore [3], if data supplied by the interrupt are passed as parameters to the entry, it would require an elaborate explanation in the Language Reference Manual and change of the compiler each time a new type of device is added. To avoid these problems it is suggested [3] that an entry call caused by an interrupt should be conditional or timed depending on the hardware and that entries associated with interrupts should have no parameters, but the passing of data should be handled in other ways, for example by the use of Ada's Low_Level_IO procedures [1:14.6]. This approach allows the compiler to translate all entry calls caused by interrupts into the same style of code. It gives the programmer more freedom, but also the burden of handling the buffering.

It is assumed that the run-time system translates the interrupts from the I/O-processor into entry calls with no parameters. Here, there are no problems with lost messages, since each interrupt should be acknowledged by clearing of cmd_in before the I/O-processor issues a new one, and each message should be acknowledged by clearing a bit in the status word before the I/O-processor could input a new message.

5.1.2. Handling_of_incorrect_messages

Unfortunately, all incoming messages are not correct. In order to handle them in a transparent manner the types from_S and from_I are introduced. This makes it possible to use standard buffers (S_buff and I_buff) of the type local_buffer to store messages inside In_handler.

5.1.3. The_user_interface

The interface to the user can be designed in different ways. Here, the entries of the In_handler task are given. The reasons are the same as those given in the discussion of buffers for use between tasks. The messages returned are of the types from_S and from_I and not of the types f_S_mess and f_I_mess. Another way of returning erroneous messages is to raise an exception. But, since there are no interface procedures, the exception must be raised inside an accept statement. If the exception is not handled locally it is propagated to the calling task as wanted, but also to the unit containing the accept statement. Consequently, In_handler must also handle the exception. If a messenger is used to forward the message to Regul, it must also handle the exception and that complicates the implementation of such a task.

5.2. Sending_of_Messages

The procedure of sending is similar to the reception. No interrupts are generated by the I/O-processor, but the status bits describing the output areas must be polled. The generic function procedures are supposed to return the size of the messages. The I/O-processor needs this information in order to use the lines efficiently. The body of Out_handler is not given, since it does not provide anything new.

6. DISCUSSION

The impressions of Ada from this study are mixed. It is possible to solve our problem in Ada. Ada provides rather powerful concepts for both logical and physical modularization. However, although our problem is simple, the solution cannot be said to be so. The inter process communication is not straightforward. It is discussed further below. However, the design and coding of a program are only parts in the development of software. Testing and maintenance are important aspects that must be taken into account when forming an overall opinion of Ada.

As seen from the discussion on the bounded buffer, it is a weakness that it is not possible to specify initialization routines which are invoked automatically when an object of (limited) private type is created.

6.1. Ada's Rendezvous Concept

Ada's concepts for synchronization and communication are subject to controversy [4,5], and the mechanisms of interaction between processes are far from being well understood. Consequently, it is motivated to compare Ada's concepts for synchronization and communication with other commonly used high-level concepts like message passing schemes and monitors.

As seen from Figure 1, it was in many cases not possible to use the rendezvous concept directly. The asymmetric properties of the rendezvous concept forced us to introduce two messengers. Furthermore, the tight synchronization implied by the rendezvouz concept forced us to introduce a buffer between Opcom and Regul to make the Opcom more independent of Regul.

The structure could easily be implemented with mailboxes and primitives for sending and receiving messages. The operating system RMX/80 [6,7], which together with the PL/M language [8] is used by the manufacturers of WTS-3 to implement the control system, provides a message passing scheme with mailboxes and primitives for sending and for conditional and timed receptions. There is no explicit synchronization between the sender and receiver, but, if a reply is wanted, the problem of returning an answer to the sender is easily solved by including a reference to an answer box in the message.

The buffers can also be viewed as monitors. A monitor [9,10,11] is a program module that defines a shared data structure, an initial operation and a set of procedures, called monitor procedures. The initial operation is executed when the data structure is created. Processes cannot operate directly on the data structure, but they can only call monitor procedures. A process has exclusive access to the data structure when executing a monitor procedure. To handle the cases where a monitor procedure finds itself unable to proceed to completion, there are mechanisms which allow it to release the monitor temporarily and wait for the condition to be fulfilled. The monitor concept is used in Concurrent Pascal [11], Modula [12] and Portal [13].

All monitors are not as simple to implement in Ada as the buffers. The problems are of two kinds.

First, Ada's process queues are associated with entries and only entries, which implies that more than one rendezvous is needed to handle a request, if processing is necessary before the monitor can decide whether it can satisfy the request immediately, or if the caller has to wait. For example, a file allocator must process a request before it knows whether the requesting task has to wait. To make it reliable, the user interface to the monitor cannot be the entries themselves, but it must be a procedure which calls the entries in a proper way. Furthermore, the conditional entry call statement and the timed entry call statement cannot then be used by the caller, but this information has to be provided as parameters.

Second, Ada's process queues are handled in a strictly first in first out manner. The concept of family-of-entries and the when clause allow simulation of static priority-ordered queues. This approach is only suitable for a small number of levels. If the number of levels is large or infinite then other, more complex approaches are required. The Ada design team [14] (Page 11.23-11.24) suggests a general technique, but it could not be said to be straightforward. Silberschatz [5] suggests a new language construct to handle priority scheduling in Ada.

7. ACKNOWLEDGEMENTS

It is a pleasure for me to thank my supervisor, Professor Karl Johan Åström for introducing me to this area and for initiating this study.

This work was supported by a research contract from Försvarets Forskninsanstalt (The National Defense Research Institute) Stockholm, Sweden. The valuable collaboration with Försvarets Forskningsanstalt is gratefully acknowledged. Special thanks to Svante Jahnberg for his assistance and to Jonas Agerberg for interesting discussions at the Ada Training Workshop on July 23-27, 1979 at the National Physical Laboratory in London.

8. REFERENCES

1. 'Reference Manual for the Ada Programming Language', United States Department of Defence, July 1980.

2. 'Swedish wind power', Karlskronavarvet AB, P.O. Box 1008, S-371 24 Karlskrona, Sweden, 1980.

3. Hibbard, P., Hisgen, A., Rosenberg, J., Sherman, M., 'Programming In Ada: Examples', Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, October 1980.

4. van den Bos, J., 'Comments on Ada Process Communication', ACM Sigplan Notices, Vol. 15, No. 6 (June 1980), pp. 77-81.

5. Silberschatz, A., 'On the Synchronization Mechanism of the Ada Language', ACM Sigplan Notices, Vol 16, No. 2 (February 1981), pp. 96-103.

6. 'RMX/80 System Users Guide', Santa Clara, CA: Intel Corp., 1977.

7. Kahn, K.C., 'A Small-Scale Operating System Foundation for Microprocessor Applications', Proceedings of the IEEE, Vol. 66, No. 2 (February 1978), pp. 209-216.

8. Mc Cracken, D.D., 'A Guide to PL/M Programming', Addison-Wesley, 1978.

9. Hoare, C.A.R, 'Monitors: an operating system structuring concept', Comm. ACM, Vol. 17, No. 10 (October 1974), pp. 539-547.

10. Brinch Hansen, P., 'Operating System Principles', Prentice Hall, 1973.

11. Brinch Hansen P., 'The Architecture of Concurrent Programs', Prentice Hall, 1977.

12. Wirth, N., 'Modula: a language for modular multiprogramming', Software - practice & experience, Vol. 7, No. 1 (1977), pp. 3-35.

13. Naegeli, H.H., 'Programming in Portal - An introduction', LGZ Landis & Gyr Zug Corporation, 1979.

14. 'Rationale for the Design of the Ada program language', ACM Sigplan Notices, Vol. 14, No. 6 (June 1979), Part B.

Appendix 1
Messenger package

```
generic
  type elem_type is private;
  with procedure get(e: out elem_type);
  with procedure put(e: in  elem_type);

package Messenger_package;

-- This package is intended to be used for moving elements
-- from one task to another task. The procedure get should
-- define the source (typically an entry of the producer).
-- The procedure put should in the same way define the
-- destination. The package contains a task which
-- infinitely first calls get and then put.



package body Messenger_package is

  task messenger;

  task body messenger is
    e: elem_type;
  begin
    loop
      get(e);
      put(e);
    end loop;
  end messenger;

end Messenger_package;
```

Appendix 1
Messenger package

```
^genericO
  ^typeO elem_type ^isO ^privateO;
  ^withO ^procedureO get(e: ^outO elem_type);
  ^withO ^procedureO put(e: ^inO  elem_type);

^packageO Messenger_package;

-- This package is intended to be used for moving elements
-- from one task to another task. The procedure get should
-- define the source (typically an entry of the producer).
-- The procedure put should in the same way define the
-- destination. The package contains a task which
-- infinitely first calls get and then put.



^packageO ^bodyO Messenger_package ^isO

  ^taskO messenger;

  ^taskO ^bodyO messenger ^isO
    e: elem_type;
  ^beginO
    ^loopO
      get(e);
      put(e);
    ^endO ^loopO;
  ^endO messenger;

^endO Messenger_package;
```

Appendix 2
Local buffer package

```
generic
  type elem_type is private;
    -- type of elements to be buffered;

package Local_buffer_package is

  -- This package provides a buffer type for local use inside
  -- a task. The elements are handled in a first-in-first-out
  -- manner. The buffers are bounded and the discriminant
  -- buff_size represents this bound.
  -- Before any use of a buffer variable it must be
  -- initialized by a call of Init_buffer.
  -- When it is no longer needed, Destroy_buffer should be
  -- invoked.

  type local_buffer(buff_size: natural) is limited private;

  procedure Init_buffer(B: in out local_buffer);

  procedure Destroy_buffer(B: in out local_buffer);

  procedure Put(B: in out local_buffer; e: in  elem_type);
  -- If the buffer B is full the exception full_buffer is
  -- raised, otherwise the element e is put into B.

  procedure Get(B: in out local_buffer; e: out elem_type);
  -- If the buffer B is empty the exception empty_buffer is
  -- raised, otherwise an element is returned.

  function  Is_empty(B: in local_buffer) return boolean;

  function  Is_full (B: in local_buffer) return boolean;

  full_buffer, empty_buffer: exception;


private

  type local_buffer(buff_size: natural) is
    record
      in_p, out_p: natural := 1;
      count:       integer range 0..integer'last := 0;
      buff_area:   array (1..buff_size) of elem_type;
    end record;

end Local_buffer_package;
```

Appendix 3
Local buffer package body

```
package body Local_buffer_package is

  procedure Init_buffer(B: in out local_buffer) is
  begin
    null;
  end Init_buffer;

  procedure Destroy_buffer(B: in out local_buffer) is
  begin
    null;
  end Destroy_buffer;

  procedure Put(B: in out local_buffer;
                e: in  elem_type) is
  begin
    if B.count = B.buff_size then
      raise full_buffer;
    end if;
    B.buff_area(in_p) := e;
    if B.in_p = B.buff_size then
      B.in_p := 1;
    else
      B.in_p := B.in_p + 1;
    end if;
    B.count := B.count + 1;
  end Put;

  procedure Get(B: in out local_buffer;
                e: out elem_type) is
  begin
    if B.count = 0 then
      raise empty_buffer;
    end if;
    e := B.buff_area(out_p);
    if B.out_p = B.buff_size then
      B.out_p := 1;
    else
      B.out_p := B.out_p + 1;
    end if;
    B.count := B.count - 1;
  end Get;

  function  Is_empty(B: in local_buffer) return boolean is
  begin
    return B.count = 0;
  end Is_empty;

  function  Is_full (B: in local_buffer) return boolean is
  begin
    return B.count = B.buff_size;
  end Is_full;

end Local_buffer_package;
```

Appendix 4
Buffer task package

```
generic
  type elem_type is private;
    -- type of elements to be buffered;

package Buffer_task_package is

  -- This package provides a buffer type for use between
  -- tasks. The elements are handled in a first-in-
  -- first-out manner. The buffers are bounded and
  -- the parameter buff_size of Init_buffer sets this bound.
  -- Before any use of a buffer variable it must be
  -- initialized by a call of Init_buffer.
  -- When it is no longer needed, Destroy_buffer should be
  -- invoked.

  task type buffer_task is
    entry Init_buffer(buff_size: in natural);
    entry Put(e: in  elem_type);
    entry Get(e: out elem_type);
    entry Return_empty(empty: out boolean);
    entry Return_full (full:  out boolean);
    entry Destroy_buffer;
    pragma priority(priority'last);
  end buffer_task;

end Buffer_task_package;
```

Appendix 5
Buffer task package body

```
with Local_buffer_package;
package body Buffer_task_package is

  package LB_package is new Local_buffer_package(elem_type);
  use LB_package;

  task body buffer_task is
    max_size: natural;
  begin
    accept Init_buffer(buff_size: in natural);
      max_size := buff_size;
    end Init_buffer;

    declare
      LB: local_buffer(max_size);
    begin
      Init_buffer(LB);
    buffer_task_operations:
      loop
        select
          when not Is_full(LB) =>
            accept Put(e: in elem_type) do
              Put(LB, e);
            end Put;
        or
          when not Is_empty(LB) =>
            accept Get(e: out elem_type) do
              Get(LB, e);
            end Get;
        or
          accept Return_empty(empty: out boolean) do
            empty := Is_empty(LB);
          end Return_empty;
        or
          accept Return_full(full: out boolean) do
            full := Is_full(LB);
          end Return_full;
        or
          accept Destroy_buffer;
          exit buffer_task_operations;
        or
          terminate;
        end select;
      end loop buffer_task_operations;
      Destroy_buffer(LB);
    end;
  end buffer_task;

end Buffer_task_package;
```

Appendix 6
I/O handler package

```
generic
  type f_S_mess is private;
  type t_S_mess is private;
  type f_I_mess is private;
  type t_I_mess is private;
  with function t_S_mess_size(x: t_S_mess) return integer
    is t_S_mess'size; -- in bits
  with function t_I_mess_size(x: t_I_mess) return integer
    is t_I_mess'size; -- in bits
package IO_handler is

  -- This package handles the communication with the
  -- supervisory processor (S) and the interface
  -- processor (I).

  type mess_status is (OK, bad_device, bad_mess);

  type from_S(s: mess_status := bad_mess) is
    record
      case s is
        when OK     => m: f_S_mess;
        when others => null;
      end case;
    end record;

  type from_I(s: mess_status := bad_mess) is
    record
      case s is
        when OK     => m: f_I_mess;
        when others => null;
      end case;
    end record;

  task In_handler is
    entry In_interrupt;
    entry Get(m: out from_S);
    entry Get(m: out from_I);
    pragma priority(priority'last);
    for In_interrupt use at 16#*****#;
  end In_handler;


  task Out_handler is
    entry Put(m: in t_S_mess; out s: mess_status);
    entry Put(m: in t_I_mess; out s: mess_status);
    pragma priority(priority'last);
  end Out_handler;

end IO_handler;
```

Appendix 7
I/O handler package body

```
package body IO_handler is

  f_S_buff_size: constant := 5;
  f_I_buff_size: constant := 5;
  t_S_buff_size: constant := 5;
  t_I_buff_size: constant := 5;
  S_index: constant := 1;
  I_index: constant := 4;

  package Hardware_interface is

    type channel is (no, SC, IC);

    type mess_size is
      record
        size: integer range 0..255; -- in bytes
      end record;

    t_S_size: mess_size;
    t_I_size: mess_size;
    f_S_area: f_S_mess;
    t_S_area: t_S_mess;
    f_I_area: f_I_mess;
    t_I_area: t_I_mess;

    procedure Get_interrupt(chan: out channel;
                            s: out mess_status);
    procedure Ack_S_mess;
    procedure Ack_I_mess;

    function t_S_area_free return boolean;
    function t_I_area_free return boolean;
    procedure Send_t_S(done: out boolean);
    procedure Send_t_I(done: out boolean);

  private
    for mess_size use
      record at mod 8;
        size at 0 range 7..0;
      end record;
    for mess_size'size use 16;

    for f_S_area use at 16#****#;
    for f_I_area use at 16#****#;
    for t_S_size use at 16#****#;
    for t_S_area use at 16#****#;
    for t_I_size use at 16#****#;
    for t_I_area use at 16#****#;
  end Hardware_interface;
```

Appendix 7
I/O handler package body

```
  package body Hardware_interface is

    type cmd_register is
      record
        cmd: 0..7;
      end record;

    type status_bits is
      (mess_in, bad_device, bad_mess, dum1,
       dum2, dum3, dum4, mess_out);


    type bit_type is (low, high);
    type status_word is array(status_bits) of bit_type;

    cmd_in:  cmd_register;
    cmd_out: cmd_register;
    S_status: status_word;
    I_status: status_word;

    procedure Get_interrupt(chan: out channel;
                            s: out mess_status) is
    begin
      if cmd_in.cmd = S_index then
        chan := SC;
        if S_status(mess_in) = low or
           S_status(bad_device) = high then
          s := bad_device;
        else if S_status(bad_mess) = high then
          s := bad_mess;
        else
          s := OK;
        end if;
      else if cmd_in.cmd = I_index then
        chan := IC;
        if I_status(mess_in) = low or
           I_status(bad_device) = high then
          s := bad_device;
        else if I_status(bad_mess) = high then
          s := bad_mess;
        else
          s := OK;
        end if;
      else
        chan := no;
        s := bad_device;
      end if;
      cmd_in.cmd := 0;
    end Get_interrupt;
```

Appendix 7
I/O handler package body

```
    procedure Ack_S_mess is
    begin
      S_status(mess_in) := low;
    end Ack_S_mess;

    procedure Ack_I_mess is
    begin
      S_status(mess_in) := low;
    end Ack_I_mess;

    function t_S_area_free return boolean is
    begin
      return S_status(mess_out) = low;
    end t_S_area_free;

    function t_I_area_free return boolean is
    begin
      return I_status(mess_out) = low;
    end t_I_area_free;

    procedure Send_t_S(done: out boolean) is
    begin
      if cmd_out.cmd = low then
        cmd_out.cmd := S_index;
        done := true;
      else
        done := false;
      end if;
    end Send_t_S;

    procedure Send_t_I(done: out boolean) is
      if cmd_out.cmd = low then
        cmd_out.cmd := I_index;
        done := true;
      else
        done := false;
      end if;
    end Send_t_I;

    for cmd_register_type use
      record at mod 8;
        cmd at 0 range 2..0;
      end record;
    for cmd_register'size use 8;
    for bit_type use (low => 0, high =>1);
    for status_word'size use 8;
    for cmd_in    use at 16#****#;
    for cmd_out   use at 16#****#;
    for S_status use at 16#****#;
    for I_status use at 16#****#;
  end Hardware_interface;

  package body In_handler is separate;
  package body Out_handler is separate;

end IO_handler;
```

Appendix 8
In handler task body

```
with Local_buffer_package;
separate (IO_handler)
task body In_handler is

  package S_buff_pac is new Local_buffer_package(from_S);
  package I_buff_pac is new Local_buffer_package(from_I);

  use Hardware_interface, S_buff_pac, I_buff_pac;

  type buffer_status is (non_full, full, overflow);

  S_buff: local_buffer(f_S_buff_size);
  I_buff: local_buffer(f_I_buff_size);
  buff_status: array (SC..IC) of buffer_status
                 := (others := non_full);
  chan: channel;
  status: mess_status;

  begin
    Init_buffer(S_buff);
    Init_buffer(I_buff);
    loop
      select
        accept In_interrupt do
          Get_interrupt(chan, status);
        end In_interrupt;
        case chan is
          when no => null;
          when SC =>
            if buff_status(SC) =/= non_full then
              buff_status(SC) := overflow;
            else
              if status = OK then
                begin
                  Put(S_buff, (OK, f_S_area));
                  exception
                    when others => Put(S_buff, (bad_mess));
                end;
              else
                Put(S_buff, (status));
              end if;
              if Is_full(S_buff) then
                buff_status(SC) := full;
              else
                Ack_S_mess;
              end if;
            end if;
```

Appendix 8
In handler task body

```
        when IC =>
          if buff_status(IC) =/= non_full then
            buff_status(IC) := overflow;
          else
            if status = OK then
              begin
                Put(I_buff, (OK, f_I_area));
              exception
                when others =>
                  Put(I_buff, (bad_mess));
              end;
            else
              Put(I_buff, (status));
            end if;
            if Is_full(I_buff) then
              buff_status(IC) := full;
            else
              Ack_I_mess;
            end if;
          end if;
        end case;
      or
        when not Is_empty(S_buff) =>
          accept Get(m: out from_S) do
            Get(S_buff, m);
          end Get;
          case buff_status(SC) is
            when non_full => null;
            when full =>
              Ack_S_mess;
              buff_status(SC) := non_full;
            when overflow =>
              Put(S_buff, (bad_device));
              buff_status(SC) := full;
          end case;
      or
        when not Is_empty(I_buff) =>
          accept Get(m: out from_I) do
            Get(I_buff, m);
          end Get;
          case buff_status(IC) is
            when non_full => null;
            when full =>
              Ack_I_mess;
              buff_status(IC) := non_full;
            when overflow =>
              Put(I_buff, (bad_device));
              buff_status(IC) := full;
          end case;
      or
        terminate;
      end select;
    end loop;
    Destroy_buffer(S_buff);
    Destroy_buffer(I_buff);
  end;
end In_handler;
```