2

IDA PAPER P-2488

REAL-TIME Ada DESIGN METHODOLOGIES AND THEIR IMPACT ON PERFORMANCE

Norman R. Howes

June 1991

91 6 18 056

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

~~91 6 18 056~~
~~9 [illegible] 031~~

IDA Log No. HQ 91-037919

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE June 1991	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Real-Time Ada Design Methodologies and Their Impact on Performance	5. FUNDING NUMBERS IDA CRP 9000-525
6. AUTHOR(S) Norman R. Howes	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard Street Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Paper P-2488
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard Street Alexandria, VA 22311-1772	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Public Release/Unlimited Distribution.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

This paper, written as part of an IDA Central Research Project, provides an overview of the existing state of Ada real-time design methodologies and qualifies the impact on performance of some of the best known and widely used Ada real-time design methodologies. The paper also discusses the limitations of these methodologies and indicates some alternative principles upon which a better design methodology might be based.

14. SUBJECT TERMS Ada; Design Methodologies; Real-time Systems; Simulation Drivers; Single and Multiple Processor Ada Platforms; Runtime System.	15. NUMBER OF PAGES 49
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

IDA PAPER P-2488

REAL-TIME Ada DESIGN METHODOLOGIES AND THEIR IMPACT ON PERFORMANCE

Norman R. Howes

June 1991

INSTITUTE FOR DEFENSE ANALYSES
IDA Central Research Project
CRP 9000-525

PREFACE

This paper, written as part of an IDA Central Research Project, provides an overview of the existing state of Ada real-time design methodologies and quantifies the impact on performance of some of the best known and most widely used Ada real-time design methodologies. The paper also dscusses the limitations of these methodologies and indicates some alternative principles upon which a better design methodology might be based. For each methodology that was analyzed, a design (by the authors of the methodology) of an example real-time system was chosen and implemented on a variety of single and multiple processor Ada platforms. Simulation drivers were implemented that could run these example programs (by providing simulations of the external events driving these example systems) as well as the alternative designs that were generated in this study. The results of the methodology designs were compared with our designs, and conclusions were drawn both from the quantitative measurements obtained and from the experience of implementing, debugging and transporting a variety of example real-time systems on a variety of Ada platforms.

The author wishes to thank the many reviewers and others that helped make these studies possible: Cy Ardoin, Karen Gordon, Dave Hough, Joe Linn, Terry Mayfield, Reg Meeson, Jim Pennell and Dave Rine. Special thanks go to: Joe Linn, who participated in numerous discussions of the merits of the various methodologies, of alternative designs for the example problems and of the difficulties involved with Ada real-time methodologies; Cy Ardoin, who helped isolate problems in the ENCORE multiprocessor runtime system and in the example systems of other authors, as well as serving as a reviewer; Dave Hough, who got the original Nielsen and Shumate example real-time system (the remote temperature sensor system) running as well as writing the first versions of the simulation driver for this system; and Dave Rine, who suggested and arranged for the graduate concurrent systems design course the author taught at George Mason University during the Spring semester of 1990.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

A General Accounting Office (GAO) Report, issued in March 1989 on the subject of *Status, Costs, and Issues Associated with Defense's Implementation of Ada* [GAO89], states that "Ada was designed for use in real-time computer applications and is being used successfully in some applications. However, Ada has not worked well in critical real-time applications - applications that have severe time and memory constraints and/or precise timing control." In this category fall real-time systems proper (with hard deadline scheduling requirements) and others such as embedded systems, concurrent systems, and distributed systems that have critical performance requirements. On page 29 of this report, under the heading *Promised Benefits of Ada Not Yet Achieved for Some Critical Real-Time Applications*, some of the problems experienced with what the GAO calls software requiring "very fast or tightly controlled computer processing" are enumerated.

In the GAO Report, the problems associated with using Ada for real-time applications are attributed to various factors such as the *compilers* that translate Ada programs into executable code, the run-time systems under which Ada programs execute, and the language itself. For the most part, these problems have to do with *efficiency* (what the GAO Report calls "very fast computer processing") or with *meeting exact timing requirements* (what the Report calls "tightly controlled computer processing"). While the author is in general agreement with this assessment, it is his belief that another contributing factor is the lack of a good method for designing real-time systems in Ada. Many of the existing textbooks and articles on this subject give poor advice on how to design efficient real-time systems in Ada. Project managers and systems designers often follow this advice in the literature and produce inefficient systems. This phenomenon will continue even if compilers, run-time systems, and the language itself are improved.

Most Ada real-time design techniques documented in the literature do not explicitly address efficiency (throughput). Many of them are primarily oriented toward supporting transportability, reusability of components, and maintainability. These methods lead to elaborate, highly complex designs that can be shown to be inefficient. Furthermore, the high level of complexity of these designs makes them difficult to debug, difficult to maintain and not very transportable.[1]

There have been a number of recent Ada oriented design methodology proposals for real-time systems in the literature. Probably the two most comprehensive are documented in the text books by R. Buhr [Buhr84] and K. Nielsen and K. Shumate [Nielsen88]. Both of these books conclude with examples of how these methodologies might be applied in designing real systems. Buhr's Chapter 6 and Chapter 7 show how a communication system with a layered protocol (e.g., the OSI Model) might be designed using Ada tasks to decouple the layers. Nielsen and Shumate's book has five appendices, each of which contains a case study of a real-time system. Some of the Nielsen and Shumate examples may not qualify as real-time systems, depending on what definition of "real-time" one uses. Their examples range from a system whose deadlines change dynamically to a system with no hard deadlines. Most of their examples focus more on the aperiodic behavior of tasks than on periodic behavior. They all have "cyclic" behavior as defined later in this report.

In [Howes89], the examples given in [Buhr, 84] were reviewed and shown by means of computer simulations and mathematical analyses to be less efficient (have lower throughput) than designs proposed by the authors of [Howes89]. In [Howes89], the concepts of *reducing mean service times for cyclic functions* and of using *physical concurrency* were introduced and used as a basis for developing the alternate designs presented therein. While reducing mean service time for cyclic functions and using physical concurrency are probably just new names for old concepts, what is new is the attempt to show how efficient real-time designs for implementation in the Ada programming language arise from applying these simple principles.

In a recent review of Nielsen and Shumate's book, A. Mili [Mili89] states: "The core of the book is part 4, which presents a complete Ada-oriented design methodology for real-time systems. This methodology is a synthesis of several existing methodologies: the modular approach to software construction, operation and test (MASCOT), the design approach for real-time systems (DARTS), Booch's object-oriented design, Yourdon's structured design, and Dijkstra's virtual layered machines methodology." While this assessment is partially correct, it fails to point out that the Nielsen and Shumate approach relies heavily on a previous work of Nielsen [Nielsen86] that attempts to extend the structured design concepts of cohesion and coupling to Ada tasks and packages.

1. Many real-time systems are not portable because they are designed to run on specific hardware configurations. What is being referred to here is that portion of code that is not rendered unportable by specific hardware or system dependencies.

Their idea about decoupling of processes (tasks) is essentially that additional tasks should be used as the decoupling mechanism. In fact, this reliance is so strong that in all five case studies, all concurrent processes are decoupled with additional Ada tasks as opposed to conditional entry calls. No where in the book is any other decoupling mechanism discussed than buffer tasks, transporter tasks, relay tasks, etc.

Recently, Sanden [Sanden89] has critiqued the design of Nielsen and Shumate's Case Study No. 1 and indicated that it might not be suitably efficient. He also proposed an alternative design that is based on the concept of *Entity Life Modeling*. The concept of Entity Life Modeling appears on the surface to be very similar to the concept of using physical concurrency in [Howes89]. Perhaps the significant differences between the designs presented in later sections of this paper and the one of Sanden are accounted for by a different understanding of what the entities or real-world processes are that are being modeled.

This paper deals with the efficiency aspect of the real-time system design problem as opposed to meeting exact timing requirements. In Section 5, we will indicate how the efficiency aspect relates to meeting timing requirements, but this is not our focus. There are several approaches to meeting exact timing requirements in real-time systems currently in use. For a certain class of problems, this can be done by an appropriate scheduling of the periodic tasks. An interesting discussion of this topic can be found in [Sha90]. The theory behind scheduling of periodic tasks begins at the point where the algorithms for accomplishing these tasks have already been designed, implemented and their timings measured. It is not concerned with how these tasks are to be designed for maximum performance (minimum execution time). It should be noticed that whatever method is used for meeting exact timing requirements, a collection of tasks with shorter execution times will be more apt to meet deadlines than the same set of tasks with longer execution times. Consequently, all real-time systems may benefit from improved efficiency.

2. BACKGROUND

During 1989, a simulation study was undertaken by the Institute for Defense Analyses (IDA) for the Ada Joint Program Office as part of a task to create a tutorial on using the NATO APSE (Ada Programming Support Environment). For the tutorial, an example was needed that could illustrate all of the capabilities of the APSE, including the host/target communication capabilities for the development of embedded systems. The example chosen has a history in the current literature. It originally appeared in the book titled *Real-Time Languages: Design and Development* by S. J. Young [Young82]. Nielsen and Shumate present an Ada version of it as Case Study No. 1 in the back of their book [Nielsen88].

The Remote Temperature Sensor (RTS) system, as this example is known, is a small, well-documented example of an embedded system dedicated to the monitoring of a physical system (in this case, sixteen furnaces). Neither the Nielsen and Shumate nor the Sanden solution for this system appeared to have been implemented, e.g., errors in the printed version of Nielsen and Shumate's solution indicated that their design had at least not been implemented exactly as described, and the lack of performance information in Sanden's comparison seemed to indicate that the comparison was theoretical rather than experimental. Consequently, it seemed that there existed no experimental quantification of how their solutions performed.

In an effort to quantify the level of performance of the Nielsen and Shumate and the Sanden designs, a simulation driver was implemented that could exercise implementations of both of these designs, and of other alternative designs that might come up. Prior to this Central Research Project, the Nielsen and Shumate design had been implemented and tested with this simulation driver on both SUN workstations and the VAX 8600. Also, several alternative designs based on the principles of reducing mean service times and using physical concurrency as suggested in [Howes89] had been implemented and tested on the SUN workstations and the VAX 8600.

This implementation and testing is a time consuming process. Testing of the implementations after they have been debugged and optimized often takes hundreds of computer runs for statistical accuracy and to evaluate the designs in a variety of operational scenarios. Our experiments show that the designs based on the concepts of reducing the mean service times of cyclic functions and the concept of physical concurrency produce designs that are significantly more efficient in terms of storage and throughput than the Nielsen and Shumate designs on sequential machines. It was thought that these designs would be more efficient on multiprocessing machines also.

One of the objectives of this Central Research Project was to determine if the solutions produced using the concepts of physical concurrency and the reduction of mean service time of cyclic functions were significantly better on a multiprocessor Ada implementation. Our new designs converted easily to the ENCORE Multimax 320, a shared memory multiprocessor machine, by simply recompiling. We were unable to get the Nielsen and Shumate implementation to run reliably on the ENCORE. Their design as presented in [Nielsen88] hangs on the ENCORE machine. This is partially due to errors in their design and partially due to errors in the ENCORE runtime system. After much debugging, it has run successfully only once. "Successfully" means it has run the necessary 20 seconds needed to measure throughput without hanging. Preliminary timings indicate that designs based on the principles of reducing mean service times of cyclic functions and of using physical concurrency are more efficient on the ENCORE than the Nielsen and Shumate design. However, the frequent errors encountered with their design has prevented the extensive experimentation needed to verify this conclusion and to determine how much more efficient the new designs are. Also, the Nielsen and Shumate design could not be made to run on the MIPS workstation (a 12.5 mip RISC architecture workstation), and the vendors of both the ENCORE and the MIPS machines are currently studying the software diagnostics that were produced when attempting to run it on their machines. On the other hand, the alternative designs based on the principles of physical concurrency and reducing mean service times of cyclic functions converted easily to the MIPS workstation by simply recompiling.

3. THE NIELSEN & SHUMATE APPROACH

In this section, we take a look at the Nielsen and Shumate design for the RTS and its throughput on a single processor machine. A Buhr [Buhr84] diagram of the original Nielsen & Shumate RTS design is shown in Figure 1. It consists of five packages (Host_Input, Host_Output, Temp_Reading, Rx_Handler, and Device _Handlers). Software for the Simulation drivers for the host (Host_Simulator) and for the digital thermometer (Dt_Simulator) was not included in the Nielsen & Shumate design and had to be written.

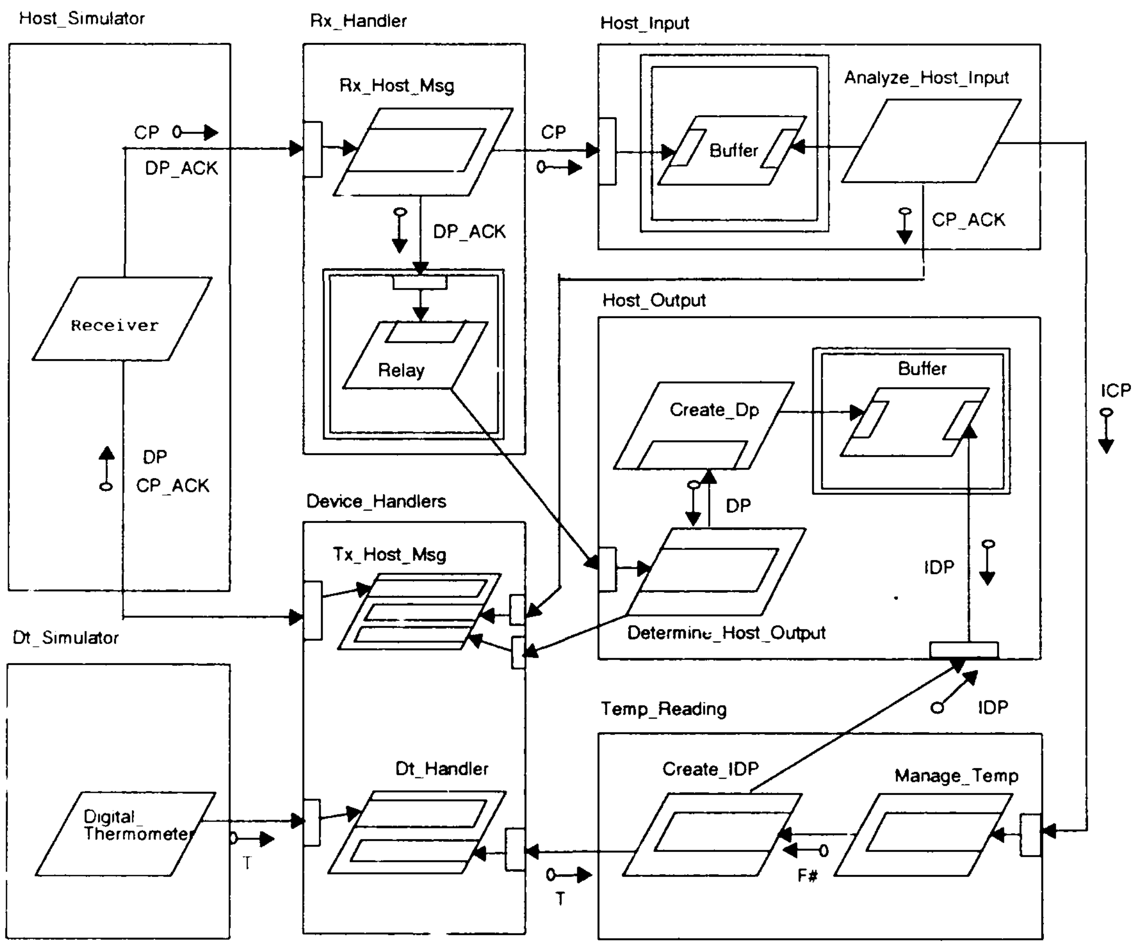


Figure 1. Original RTS Design by Nielsen and Shumate

In a nutshell, what the RTS system does is receive messages from a host system in the form of Control Packets (CPs) and sends messages back to the host in the form of Data Packets (DPs). The DPs contain a furnace number and temperature. These *temperature readings* are obtained by sending a furnace number to the digital thermometer which in turn returns the temperature. The CPs from the host are used to change the periodicity at which the various furnaces are read. They contain a furnace number and a time expressed in seconds that tells how often to read the temperature with the corresponding furnace number.

The code for the Nielsen & Shumate design (not including the code for the host or the digital thermometer) is given in their book. In implementing this code, we encountered several small errors that indicate that their design had not been implemented as described in their book. One problem was that the interface protocols were unsafe, relying on relative process speeds to perform correctly. This caused errors on output from the RTS. The same potential problem existed in the Input_Channel and Dt_Interrupt rendezvous (for input and reading furnace temperatures), but the timing of the system was such that problems were not encountered there.

While considering this problem, we concluded that their method of employing interrupt entries for input, output and furnace reading was unrealistic in that it took seven interrupt rendezvous to receive a seven character input message and nine rendezvous to transmit an eight character message. It was observed that for a real system, there were a number of methods that could be employed to transfer a seven or eight character message that would require only one interrupt rendezvous. As a result, we changed the simulator code and the RTS system code to pass messages via an ordinary rendezvous for purposes of this simulation.

It was also observed that for reading furnace temperatures, for a real system, the RTS would probably need to signal the digital thermometer when it was ready to read a given furnace, and the digital thermometer would need to signal the RTS when it had the temperature. The Nielsen & Shumate design does not explicitly signal the digital thermometer via accepting an interrupt entry like they do when sending a DP to the host. They assume that the RTS places the furnace number in a given memory location and then waits for an interrupt call from the digital thermometer signaling that a temperature has been provided, at which time the temperature resides in another predefined memory location. This would require the digital thermometer to have knowledge about when the furnace number memory location changed in the RTS. Although such an interface device could be built, it was thought improbable that a real system would work this way. Nonetheless, since their approach did not involve an unreasonable number of rendezvous (as in the case of the host input and output), and since it did not produce errors in the simulation, their method was retained and also employed in our designs to make all of the solutions uniform in their interface with the simulation driver.

Both the Nielsen and Shumate design and the Sanden design have what might be considered a severe design limitation. In the Nielsen and Shumate design, the task Rx_Host_Msg in package Rx_Handler both accepts interrupt entries to receive messages from the host and makes calls (1) to a buffer task in package Host_Input to buffer control packets and (2) to a relay task in the same package to get rid of data packet acknowledgments it has received from the host. Either of these two tasks could be busy at the time of the call resulting in the interrupt handler task (Rx_Host_Msg) being blocked.

The consequences of having an interrupt handler task blocked awaiting the acceptance of a call to another task depend to some extent on the particular implementation of interrupt entries one is using. The Ada Language Reference Manual says very little about the behavior of inter-rupt entries, other than that the run-time system can either queue interrupts or not and that depending on whether or not they are queued, they can be considered as either non-conditional or conditional entry calls. According to the requirements specification given by Nielsen and Shumate, the consequence of "loosing an interrupt" and its associated message (i.e., having another message arrive before the previous one is serviced) would not be disastrous. If the interrupt signified a DP acknowledgment and the interrupt was lost, the RTS system would simply have to retransmit the DP after a delay of two seconds. Similarly, if the interrupt corresponded to a CP, the host would have to retransmit after a similar timeout delay.

But still, frequent loss of interrupts will significantly degrade performance, and the loss of a CP interrupt might be especially unfortunate. One can imagine that in a real world system, the sending of a CP may be in response to the evaluation of a previous DP. For instance, a DP may contain a temperature value that is out of range for a particular furnace. In response, the host wants to immediately increase the reading rate of this furnace to more closely monitor the situation. Or, one could imagine the situation where the CP might contain a periodicity of zero which would be a signal to the RTS to shut the furnace down.

One can significantly reduce the probability of losing an interrupt by employing a different task interface structure. If the three entries in task Rx in package Input in Figure 2 are labeled A, B and C respectively, then the following entry acceptance structure can be used:

```
while not Finished loop
    select
        accept A
        – service interrupt
        end A;
        – buffer incoming message
    or
```

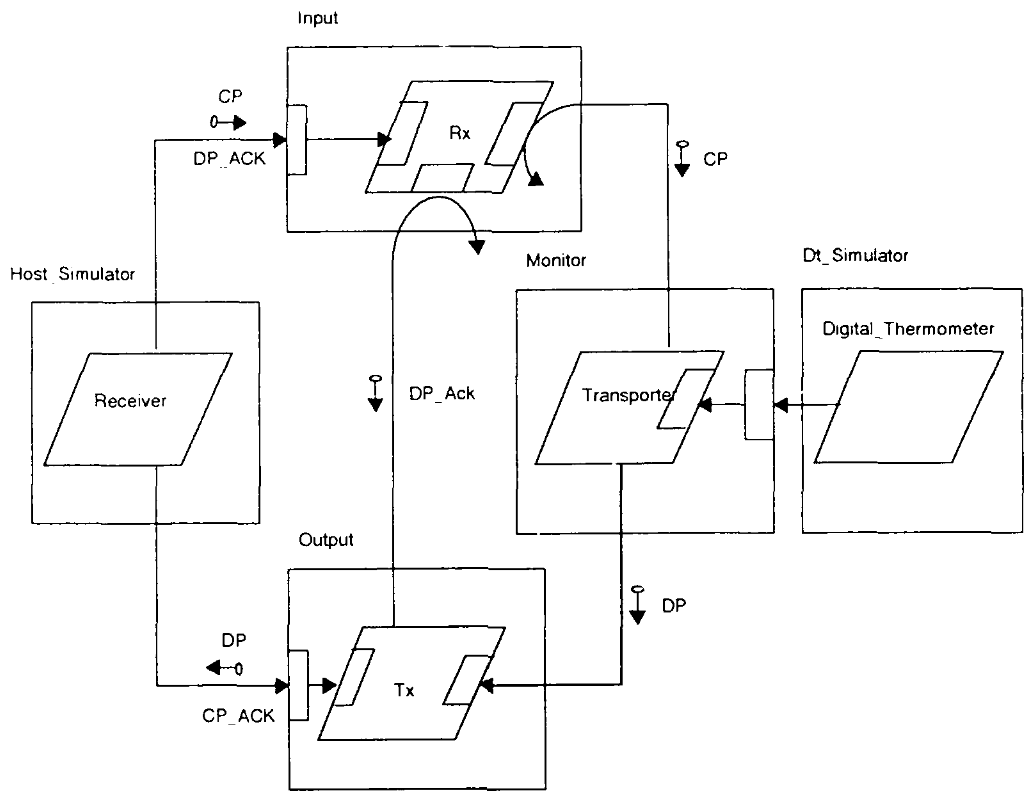


Figure 2. Redesigned RTS Using 3 Tasks

```
        accept B
        – give DP ACK to task Tx;
        end B;
    or
        accept C
        – give CP to task Transporter;
        end C;
    end select;
end loop;
```

This structure has the advantage that by doing the buffering of input messages from the host in the task Rx, the overhead of a rendezvous is avoided. Whereas, it takes at most a few microseconds to buffer the message in task Rx, it would take several hundred microseconds to rendezvous with a separate buffer task and risk waiting for the rendezvous even longer if the buffer task was busy at the time. Similar considerations hold for outgoing messages, but it is of far less consequence that interrupts for outgoing messages are lost.

Our performance simulations ran for 20 seconds. The number of DPs sent plus the number of CPs sent was divided by 20 to determine the number of readings per second (it was reasoned that a CP message took about as long to process as a DP and therefore should be counted in the DP readings). Each CP and each DP had associated acknowledgment messages, so the total number of messages being passed between the host and the RTS was exactly double the number of observed furnace readings. All the simulations ran with all furnaces being read, but with different periodicities. Furthermore, the various furnaces were started up at different times during the simulation (i.e., they did not all have the same start time). They were started by sending CPs to the RTS.

Originally, the Nielsen and Shumate design, running on a SUN model 3/50 (68020 processor) workstation using the Verdix Ada compilation system version 5.7 was only capable of making about eight or nine furnace readings per second. One of the reasons was that the delay parameters they chose for various time-outs were not optimal for maximum throughput. After, considerable tuning, which involved some minor code changes as well as adjusting parameters, we were able to obtain 30 furnace readings per second from the Nielsen and Shumate solution.

4. AN ALTERNATIVE APPROACH

An attempt has not been made to explain how one progresses from the specification to the design shown in Figure 1 using the Nielsen and Shumate methodology. The interested reader is referred to their text for their own explanation. Their path leads through a variety of diagrams from which one makes various decisions that allows one to progress to the next diagram. The path embraces many steps that are derived from other (primarily sequential) design methodologies. In this section, we give a brief intuitive explanation of how one progresses from the specification to the design using the concepts of physical concurrency and of reducing the mean service times of cyclic functions.

In [Howes89] some terminology was introduced for discussing the ways in which concurrency is introduced into a design. These three terms were *maximal concurrency, conceptual concurrency* and *physical concurrency.* By maximal concurrency we mean that concurrency is a goal in itself. For example, the best way of introducing concurrency into a compute-bound algorithm on a parallel architecture machine may be to introduce as much concurrency as possible. Here, other design goals such as simplicity of the algorithm or efficient usage of the processors may be sacrificed in order to bring the maximal number of processors to bear on the problem with the hope that this will achieve the minimal total execution time for the given architecture. By conceptual concurrency, we mean that concurrency is introduced to conceptually simplify a design that would be more complex otherwise. For instance, a priority messaging system might be expressed in a conceptually simple fashion by using concurrent tasks to model different priority message streams even though it might be more efficient to process the messages in a "pipeline" with some provision for higher priority messages to "overtake" lower priority messages in the pipeline. The algorithms and data structures for "overtaking" may be conceptually more complicated than the concept of multiple message streams.

By physical concurrency we mean that concurrency is only introduced to model physically concurrent objects or processes in the problem domain. The concept of physical concurrency is the basic structuring concept for our approach. To illustrate how the RTS was structured from this principle (as shown in Figures 2 and 3), we reasoned as follows: the problem, as described in Nielsen and Shumate's book, involves three "real-world" processes. The first is the handling of the incoming message stream from the host machine. The second is the handling of the outgoing message stream from the RTS. The third is the management of the temperature reading cycles of the furnaces. Using the concept of physical concurrency, we model each of these processes with an Ada task. This is why the alternate designs of this section all have three tasks.

To these three concepts, we could add a fourth, namely, *minimal concurrency.* By minimal concurrency we mean that avoiding concurrency is a design goal. In experimenting with alternate designs we also investigated solutions that almost completely eliminated concurrency, and a brief mention of the results with these designs is included in a later section. With the current state of Ada runtime technology for parallel architectures, it is possible that minimal concurrency designs will still give better throughput on parallel architecture machines. It is certainly true that they currently do for single processor machines. However, with a view toward the time that more efficient Ada runtime systems might become available for parallel architecture machines, we did not avoid concurrency in the alternate designs. Instead we adopted the physical concurrency philosophy and only used concurrency to model real world concurrency in the problem domain.

The next design decision with which we had to deal was how these three tasks should interface with one another. This is where we used the concept of reducing the mean service times for cyclic functions. By a *cyclic function*, we mean something the system has to do repeatedly that involves some form of coordination with another task or the external environment. One measure of throughput in a real-time system is how many times these cyclic functions are performed in a given period of time. If we measure throughput in this manner, the only way to increase throughput is for the time it takes to perform the cyclic function(s) to decrease. We call this time the *mean service time* of the cyclic function. This is analogous to the usage of the term in queueing theory. From queueing theory we know the formula (for Poisson arrivals) for the time spent in the queue as a function of how long it takes to service each element in the queue (the mean service time). If we think of each process we are modeling as a queue that is being serviced by the performance of some cyclic function, then the mean service time of the cyclic function corresponds to the mean service time for the queue, and we can think of our system as a collection of inter-related queues. We see that the only way for the throughput to increase is for the mean service times for the queue elements to decrease.

Consequently, we try not to introduce any decoupling tasks (buffer tasks, transporter tasks or relay tasks) between the three tasks we already have to model the three real-world processes, because in general, this will substantially increase the mean service time of the function. There are times when this is not possible, e.g., when two of these real-world processes need to share a single resource. But frequently we find someone mistakenly advocating the *protection* of a resource by a buffer task when a single task puts elements in the buffer and a single task takes elements out. Fortunately, there is no need for buffer (decoupling) tasks in the RTS.

But even if we verify that no decoupling tasks are needed, there are still several ways a pair of tasks may communicate with each other in Ada, and it may not be intuitively obvious which method is most efficient. Furthermore, since there may be many pairs of communicating tasks, the number of possibilities can be very large. For a given task pair, there are several issues such as:

- Do they need to be decoupled?
- If they need to be decoupled, what decoupling strategy should be employed?
- Which task should be the caller?
- Where should flow control reside for these communications?

Unfortunately, at this time we do not know a rule or set of rules that will decide these issues in general. But we do know that however we do it, it should somehow collectively minimize the mean service times of all the cyclic functions. Therefore we proceed as follows: we first choose the way that looks best intuitively, and then we experiment using the principle of minimizing mean service times of cyclic functions as our guide to experimentation.

In what follows, we will show how this was done for the RTS by showing an initial guess (Figure 2) of how it might be done, implementing this design and measuring its performance, analyzing the mean service times for the individual cyclic functions, and hypothesizing changes that will minimize these mean service times. One of the major strengths of the Ada language is that it is easy to analyze a number of alternate structures in a short time since the concurrency management facility is part of the language. Figure 3 shows a solution that was arrived at after a number of iterations. *The performance of each iteration was* measured and used as a basis for the next iteration.

In practise, the initial guess may be the first guess that actually runs (without deadlock, livelock, etc.). Once a running solution exists, it is easy to experiment with other task-to-task communication structures. The author was able to try over a dozen variations of the RTS system in less than a month. Each variation experimented with needs to have its throughput carefully measured and compared with previous experiments to determine trends of which kind of variation enhances performance. The design shown in Figure 3 had higher throughput than the one in Figure 2 on all implementations it was tested on. We say that such a solution is *fundamentally better* than the one it is being compared with. A design that is better on a given implementation, but not on all implementations will be said to be *implementationally better.* Of course, one can never know for certain that one solution is fundamentally better than another solution since a given solution can not practically be tested on every possible configuration, but we can talk about one solution *appearing* to be fundamentally better than another.

The design shown in Figure 3 appears to be fundamentally better than the design shown in Figure 2, which in turn appears to be fundamentally better than the Nielsen and Shumate design. We have not discussed the logic inside any of the tasks shown in Figure 2 or 3 and we will not go

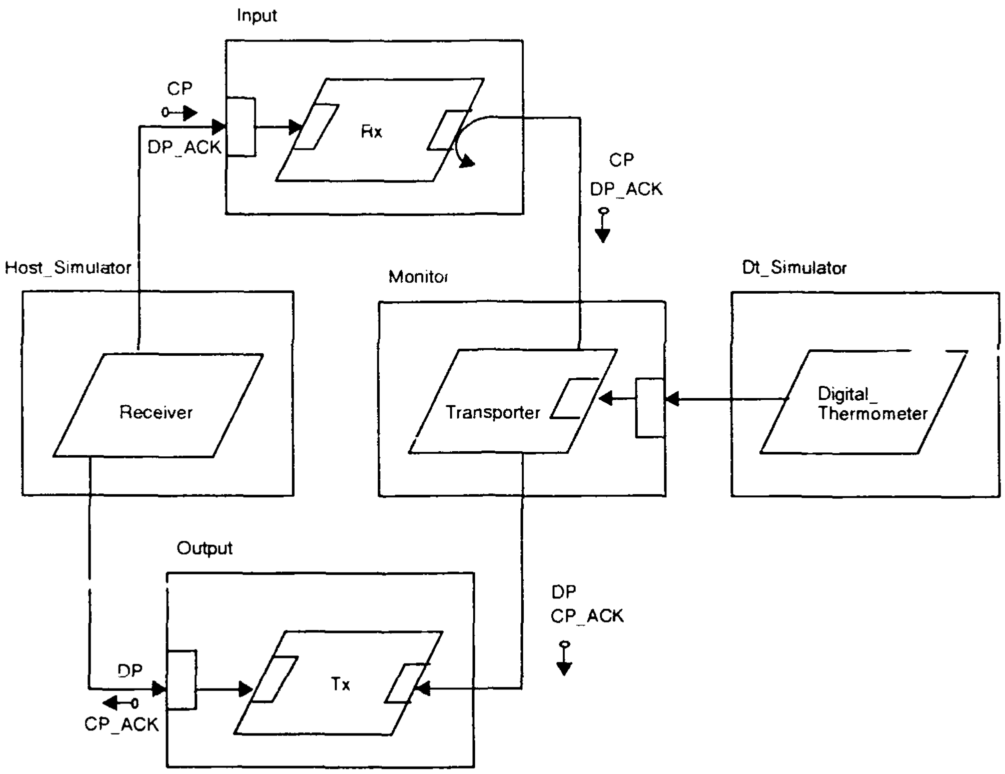


Figure 3. Redesigned RTS with symmetric I/O

into that kind of detail here. However, it needs to be pointed out that the *structuring principle* of using physical concurrency determines the performance envelope. Both the designs in Figures 2 and 3 have significantly better throughput than the Nielsen and Shumate design. But within this performance envelope there is still significant room for improvement using the *tuning principle* of reducing the mean service times of the cyclic functions.

5. RESULTS ON SINGLE PROCESSOR MACHINES

The design shown in Figure 2 was capable of 73 furnace readings per second on the SUN 3/50 workstation using the Verdix version 5.7 Ada compilation system. The design shown in Figure 3 is capable of 82 per second. This represents improvements in throughput of 243% and 273% respectively over the corresponding Nielsen and Shumate design. The design shown in Figure 3 differs from the design shown in Figure 2 in that flow control for outgoing messages is done in package Monitor in the task Transporter. The design shown in Figure 2 leads to a more even distribution of code between the packages Monitor and Output. Message buffering in task Tx in package Output is not needed in the design shown in Figure 3 while using the Nielsen and Shumate acknowledgment protocol that calls for a DP to be acknowledged before another one is sent. The reason the Tx task is retained in this design is so the Transporter task will not have to service interrupts. This design leads to a nice symmetry with respect to I/O.

It should be noted that the alternative designs have a disadvantage with respect to the Nielsen and Shumate design in that the Nielsen and Shumate design allows the task Rx_Host_Msg to make calls to other tasks. It has previously been explained why this might be very unacceptable in certain circumstances. In fact, our testing revealed that the incoming message interrupt handler could be blocked for up to 1 full second at a time during our testing. If it can be determined that this is not a problem, and the incoming interrupt handler can be blocked for durations of this magnitude, then yet more efficient 3 tasks designs can be achieved. Such designs were observed to operate at over 110 furnace readings per second (a 367% enhancement).

Another interesting observation is that if one knows in advance that the target architecture is a single processor machine, the most efficient design appears to be a single task version. In this case, a single task is retained to allow for interrupt entries. All processing of incoming and outgoing messages and of temperature readings is done within this single task. This design runs above 180 furnace readings per second (a 600% enhancement). Moreover, the time it is away from a state in which it can process an interrupt is only slightly greater than the designs shown in Figures 2 and 3. It also has a source code size only 1/4th the size of the Nielsen and Shumate design.

The design shown in figure 3 was rehosted on a MIPS workstation (12.5 MIPS) by simply recompiling. The throughput was significantly better, averaging 260 temperature readings per second on a large number of runs. However, the statistical variation between runs was quite large with the highest throughput recorded being 357 furnace readings per second and the lowest being 156 furnace readings per second. The Nielsen and Shumate design deadlocked on the MIPS machine, and the Verdix debugger was almost useless on the MIPS machine, because it would frequently hang without any indication of what was causing the problem.

10 days of manual debugging was not enough to get the Nielsen and Shumate program running on the MIPS machine. When the vendor (who was good enough to lend us the machine for the tests) learned of the nature of the difficulties we were having (due to our inquiries regarding the strange behavior of the Verdix compilation system on the MIPS), they withdrew the machine. The malfunctioning Nielsen and Shumate program and the working alternate solutions were given to the vendor's technical support staff for analysis.

Before discussing performance results in a multiprocessor environment, we relate how these measurements can be used to determine certain things about this system's ability to meet its timing requirements. The requirements for the RTS system as stated in [Neilsen88] do not specify how critical it is for the RTS system to meet its timing requirements, but it is clear from the requirements specification that the system has exact timing requirements to meet since the purpose of the CPs is to set the periodicities for the reading of the furnaces. In order to facilitate our discussion, we will hypothesize a scenario that makes the timing requirements explicit and indicates the criticality of meeting these requirements. Notice first that the RTS system does not lend itself very well to the type of analysis conducted in [Sha90] since the periodicities of the readings can change dynamically. If the requirements specifications were changed a little so that the 16 furnaces were 16 different sensors, for instance, then it might require 16 distinct tasks to make the readings. The rate monotonic algorithm discussed in [Sha90] deals with scheduling periodic tasks whose periods are fixed. Since we know that the Nielsen and Shumate design can make 30 furnace readings a second on the SUN 3/50, and that an alternate design is capable of 80 furnace readings a second, we hypothesize a scenario that the Nielsen and Shumate design can just handle within a small margin of resource utilization. Suppose the RTS system needed to read each of the furnaces once a second under ordinary circumstances, but if one of the readings were "out of range", the frequency of reading that furnace would immediately be changed to 10 readings per second.

Since the Nielsen and Shumate design can make 30 readings a second, it can handle one out or range condition (one furnace requiring 10 readings per second and 15 furnaces requiring one reading per second) plus have enough spare capacity to handle the arrival of a few new CPs. If a second out of range condition occurs, it will start missing its deadlines. This degradation of performance will occur in a "graceful" manner in the following sense. When a deadline is missed, the processing to meet this deadline will occur as soon thereafter as possible. If the system gets so far behind that two consecutive deadlines are missed for a particular furnace, the system will not try to "catch up" in the sense that it will handle only the most recently missed deadline as soon as possible. For instance, if two out or range conditions should occur at the same time, all of the furnaces that are in range will still get read once per second, but the two out of range furnaces will only be read 8 times per second rather than the specified 10 readings per second for an out of range furnace. Similarly, if there are three simultaneous out of range conditions, the out of range furnaces will only be read 5 times per second.

On the other hand, Our alternative design can handle six out of range conditions simultaneously with no missed deadlines (6 furnaces with 10 readings per second and 9 furnaces with one reading per second). Other scenarios could also be considered such as the ability to read more than 16 furnaces, or for the nominal reading frequency to be higher. In any of these cases, we see that the more efficient design is less likely to miss its deadlines. For this particular system, this measurement of throughput gives us a way of reasoning about the ability of the system to meet its deadlines. In general, complex real-time systems may not only have dynamically varying periodicities, but various different nominal periodicities. Furthermore, the "real" priorities (i.e., the intrinsic criticality of a task) may not be the same as the nominal (scheduling) priorities that are assigned to insure that a system nominally meet its deadlines.

If the rate monotonic scheduling algorithm is used, it insures that as the computing resources are consumed, the tasks with the lowest "scheduling priorities" begin missing their deadlines before those with higher scheduling priorities. The problem here is that the scheduling priorities are not necessarily the same as the real priorities. As pointed out in [Sha90], it is sometimes possible to "fool" the rate monotonic scheduling algorithm in such a way that for certain tasks, the real priority and the scheduling priority are the same. This of course requires more scheduling overhead, but it may be acceptable to insure predictable operation of the system. In general cases where the rate monotonic scheduling algorithm is not applicable, simulating performance of the system (or of a representative model of the system) as we have done here for the purpose of measuring efficiency can also be used to settle questions regarding whether a system will be able to meet its deadlines under certain conditions, or how the system will respond as the load placed on the system exceeds its processing capacity.

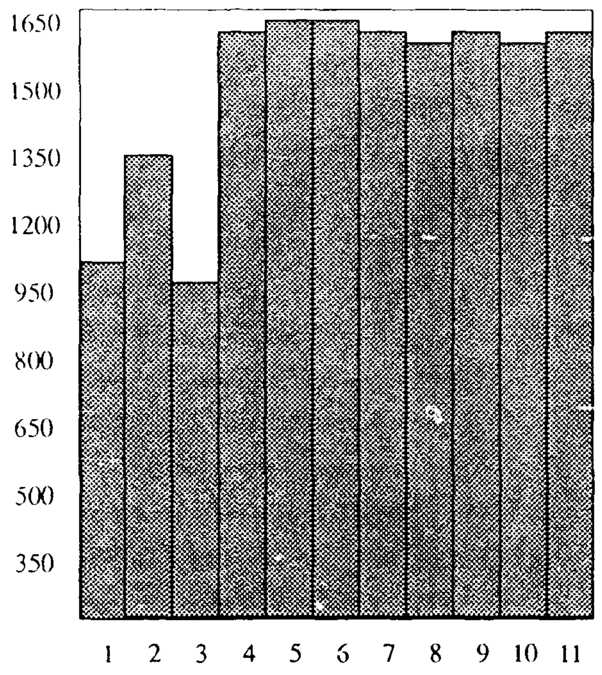
6. RESULTS ON A MULTIPROCESSOR MACHINE

The design shown in Figure 3 was converted to the ENCORE Multimax 320 by recompiling. The Ada compilation system used on the ENCORE was Verdix version 5.51 with a parallel runtime system written by the hardware vendor. The *spin scheduling* runtime option was used as it provided the best throughput, although it has some peculiarities when there are more tasks than processors. The ENCORE Ada runtime system allows the user to specify the number of processors to be used for a particular program. The table in Figure 4 shows the throughput for the design from Figure 3 as a function of the number of processors utilized.

The table in Figure 4 was constructed as follows. First the ENCORE was configured with one processor and the throughput of the Figure 3 design was measured. Then it was configured with two processors, then with three and so on, measuring the throughput of the Figure 3 design at each step. Throughput was measured as follows. The simulation driver together with the RTS program was run for 20 seconds and the number of data packets sent during that period was measured. This was repeated 10 times for each configuration and the results averaged to get the throughput for a given number of processors. So for 5 processors, 1670 data packets were transmitted in a 20 second period. As the number of processors was increased beyond 11 (end of table), the throughput declined to 1530 data packets in the 20 second running interval for all 14 processors.

It should be noted that even with all these processors, the best throughput on the ENCORE was essentially the same as the single processor SUN 3/50 workstation (83.5 readings per second).[2] This is consistent with the ENCORE performance measured in [Howes89]. The irregularities at the left hand side of the table are caused by the spin scheduling algorithm's implementation. With spin scheduling, each task is assigned to a processor until it blocks (e.g., at a call or a delay, etc.) at which time the processor is assigned to another task if one is waiting. If there are as many processors as tasks, a context switch never needs to be made for a rendezvous because every task will remain on the processor to which it was originally assigned. But when there are not enough processors to go around, context switches may take place. In this situation, the ENCORE runtime system performs unpredictably, with certain combinations of tasks and numbers of processors working better than others.

2. Although both the SUN workstation and ENCORE 320 experiments utilized the Verdix VADS system, the versions of the compilers, code generators and runtime systems were different



Throughput vs. Number of Processors

Figure 4. Throughput for Design

To date, the Nielsen and Shumate design still does not run reliably on the ENCORE as previously noted. One time, it ran for the full 20 seconds configured with all the processors and we obtained a single measurement of 1300 data packets sent, but this measurement may be high since not all the control packets in the simulation made it through. One of the tasks involved with the servicing of CPs died during this 20 second interval and the exception handlers of the other tasks were set to allow them to continue running in this state. Also, not all the functions involved with getting the DPs transmitted were being performed for the same reason. Consequently, not all the furnaces were being read and the DP processing was faster than if the program were executing correctly.

In fairness to the Nielsen and Shumate solution, it should be pointed out that we think the programming errors have been corrected in their design on the ENCORE and what we are now witnessing are problems with the run-time system. One problem we have been able to isolate in the ENCORE run time system is that the runtime system will often awaken an accepting task before the calling task is entered in the entry queue. Thereafter, whenever an entry call is made to the abnormally awakened task, an exception is raised. Why this problem happens so often with the Nielsen and Shumate design and never with our alternate design probably has to do with the processing time, for placing the calling task in the accepting task's entry queue, increasing as the number of tasks increases.

There appear to be yet other run-time system problems with the ENCORE. We have sent copies of the malfunctioning Nielsen and Shumate programs to ENCORE for their analysis and are awaiting a resolution. While the inability to rehost the Nielsen and Shumate solution on either the MIPS or ENCORE machines (both of which employ Vertex Ada compilation systems) now remains a run-time system problem, it still shows the additional strain that this type of design puts on a run-time system. Considerations such as these reflect on the transportability and maintainability of such designs.

This one data point suggests that the Nielsen and Shumate design's throughput could be considerably better in a multiple processor environment that can eliminate context switches for rendezvous (though still not as good as the three task designs). However, it would still have the big disadvantage of the large number of distinct threads of control. Such a design is very difficult to debug since there is such a variety of things it can do and it is difficult to get the errors to repeat frequently.

Fortunately, the design shown in Figure 3 converted to the ENCORE without debugging, so no experience was gained in debugging such a design on the ENCORE. However, it is felt that the relatively few distinct threads of control in the Figure 3 design (as compared with the Nielsen and Shumate design) would make debugging much easier. It is felt that the fewer threads of control will result in fewer initial timing problems when debugging commences.

Finally, no attempt has been made to date to tune the design of Figure 3 on the ENCORE. It is possible that some improvement could be made by experimenting with the task interfacing structures on the ENCORE as was done on the SUN workstation. There is no reason to believe that the ones used for the SUN design will prove to be best for the ENCORE.

7. THE DARTS APPROACH

Nielsen and Shumate's original version of the DARTS robot controller is shown in Figure 5. The specification for the robot controller is a bit more complex than the specification for the RTS. The robot controller, as the name implies, controls a robot. It does this by sending Axis_Output commands to the robot. Axis_Input messages are received from the robot that tell the current position of an axis. Task Robot_Handler in package Axis coordinates the sending and receiving of commands and axis messages to and from the robot. The robot motion commands come from task Axis_Manager in the form of *motion blocks.*

Axis_Manager gets these motion blocks from task Program_Interpreter in package Motion. Task Program_Interpreter interprets stored programs. These programs have both *motion commands* and *sensor commands.* When a program completes, the operator is notified via a message being sent to the control panel which lights up a button on the panel. The operator can then select another program that will cause the robot to perform some other job, or the same program can be selected again. The operator can stop and resume or abort the robot controller in the middle of a job (stored program). All of these options are communicated to the robot controller via the control panel. Finally, it should be mentioned that the sensor commands can retrieve information about the external environment or send information about out of bounds conditions for the robot movements.

Nielsen and Shumate follow the DARTS design methodology [Gomaa84] very closely in developing the design for Case Study 5. This methodology is somewhat different than the one they use for the design of the RTS system. In what follows, a concise overview of the DARTS method will be given and compared with the approach discussed above based on the principles of using physical concurrency and minimizing the mean service times of cyclic functions.

Gomaa [Gomaa84] starts with a brief survey of several design methods used for designing real-time systems and summarizes by stating:

> "The above survey of design methods shows that each method has some limitations in terms of real-time systems design. The two methods that come closest to satisfying the needs of real-time systems design are Structured Design and Mascot."

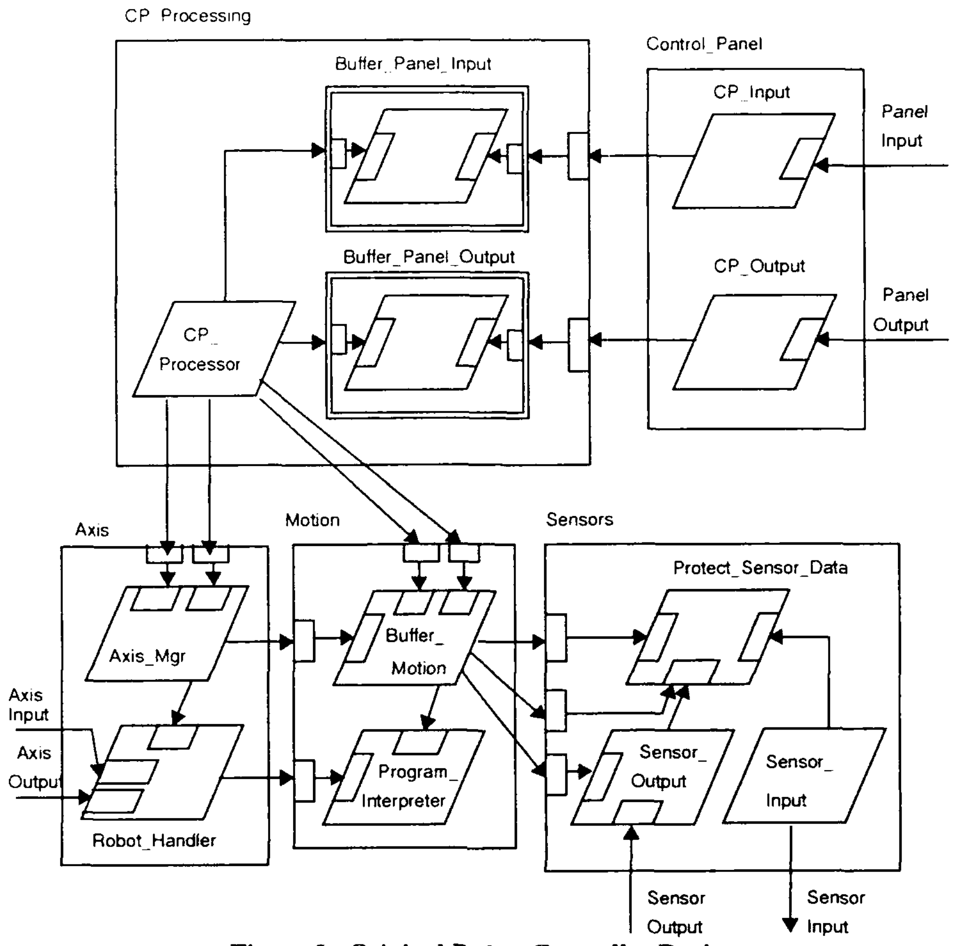


Figure 5. Original Robot Controller Design

Gomaa goes on to say that:

"The data flow approach to software design is particularly appropriate for the design of real-time systems because the data in these systems may be considered to flow from input to output and in between to be transformed by software tasks. The data-flow-oriented methods are best exemplified by Structured Analysis and Structured Design which are frequently used together. . . .

Structured Design [Yourdon78], also known as Composite Design [Myers76], consists of two main components: (1) two sets of criteria, cohesion and coupling, which are used for evaluating the quality of a design; and (2) a design method for guiding designers in a top-down decomposition of a system into modules. . . .

The Structured Design method consists of two design approaches. Transform Centered Design and Transaction Centered Design. With Transform Centered Design, the major streams of data are identified as they flow and are transformed from external input to external output. . .

Because real-time systems are usually data flow oriented, the DARTS method starts with a data flow analysis of the system."

From the original data flow, Gomaa jumps right in to decomposing the system into tasks. He states:

> "Having identified all the functions in the system and the data flows between them, we are now in a position to identify concurrency. The next stage of the DARTS method therefore involves determining how concurrent tasks will be identified on the data flow diagram."

Gomma then gives six heuristics for grouping the transformations in the data flow diagram into tasks. The problems with this approach are many. First, although data flow diagrams have been successfully used in decomposing programs with a single thread of control into modules, it is not clear that they will be as useful in decomposing programs into multiple threads of control. Second, the data flow diagrams contain no timing information about the problem space and therefore do not contain all the information necessary to make decisions about groupings into tasks. Third, data flow diagrams for a given problem are not unique. Virtually any two designers will produce a different data flow diagram for a given problem. Gomaa's heuristics applied to different data flow diagrams for the same problem give different groupings into tasks.

By using the principle of physical concurrency, one avoids these problems to a large extent. Since the structuring of the solution to the problem comes from considering the real-world objects and processes to be modeled, one does not have to be so careful in considering the timing relationships between tasks, because all the transforms related to subfunctions of a real-world process are going to naturally be grouped together. The principle of using physical concurrency does not solve the timing problems, rather it attempts to avoid them during the task structuring phase of the design. Then, within a task structuring, it attempts to optimize the throughput (minimize timing problems) by experimentation with the various task interfacing structures using the principle of reducing the mean service times for the cyclic functions. Gomaa also avoids the timing problems during the task structuring phase of the design, but in his method they are not only avoided, they are virtually ignored.

On the other hand, decomposing the system into transformations (via the data flow diagram) and then building back up to the level of tasks by grouping the transformations into tasks without any information about timing seems to be very dangerous from an efficiency point of view. It is not surprising that the design shown in Figure 6, due to one of the author's students at George Mason University, is significantly more efficient than the Nielsen and Shumate/Gomaa solution of Figure 5. We let the reader judge for himself whether designs produced with the physical concurrency structuring principle or the Gomaa heuristics applied to a data flow diagram produce a more natural decomposition into tasks. However, it cannot be argued that Gomaa's heuristics produce a more economical structuring.

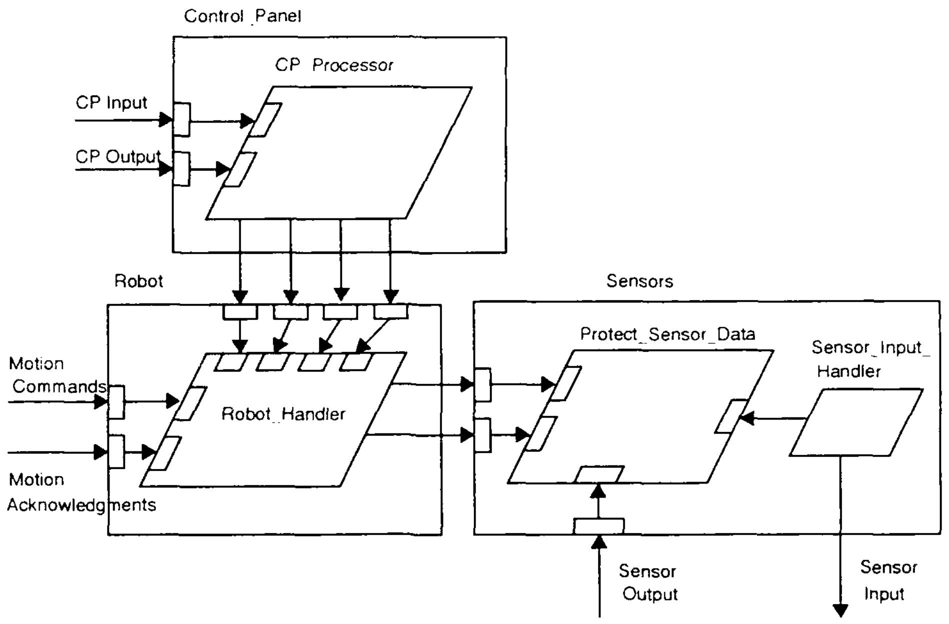


Figure 6. Redesigned Case Study 5

Both the Nielsen and Shumate/Gomaa design and the student design were implemented on a VAX 8620 using VMS Version 5.2 and DEC Ada Version 1.5. The throughput for each design was measured as the number of commands per second that could be interpreted and executed by the system, while maintaining a flow of commands from the control panel corresponding to 1 control panel command from the operator for every 3600 program commands executed. The throughput varied as a function of the mix of motion commands and sensor commands, as the sensor commands take less time to execute in the Nielsen and Shumate/Gomaa design.

The result for the Nielsen and Shumate/Gomaa design for a mix consisting of 100% motion commands was 794 commands per second while the student design of Figure 6 measured 2,179 commands per second (274% better). With a 100% sensor command mix, the Figure 5 design measured 1,341 commands per second while the student design measured 2,242 per second (167% better). Notice that the student design executes both motion commands and sensor commands at about the same speed, whereas the Figure 5 design executes sensor commands almost twice as fast as motion commands.

With a mix of 75% motion commands and 25% sensor commands, the Figure 5 design measured 886 commands per second while the student design measured 2,179. A 50/50 mix measured 900 commands per second for the design of Figure 5 and 2,162 for the student design.

The student design, it will be noticed, is very tightly coupled in that all task-to-task calls are blocking calls. While this design may be unnecessarily tightly coupled, it is still indicative of the performance improvement that can be obtained using the approach documented herein. This possibly undesirable feature could be removed at a fraction of the performance penalty paid by the Nielsen and Shumate/Gomaa solution.

8. CONCLUSIONS

To date, the principles of using physical concurrency and of reducing the mean service times of cyclic functions have been applied to three different types of real-time systems that have previously been discussed in the literature. They have been proven to produce significantly more efficient designs than the three other methods originally used to design these three systems.

These two principles do not suffice to decide all the questions one generally encounters when designing real-time systems. However, as far as they go, they seem to provide better solutions than other methods they have been compared against. Currently, the author is not aware of a good Ada real-time design methodology. The existing Ada real-time design methods have not been tested sufficiently to confirm their value, if any. This is an unfortunate situation within the Ada software engineering discipline. Methods are being hypothesized and documented with apparently little or no testing to determine if these methods produce *efficient* designs, which often is an important goal of real-time systems. It is somewhat like a physicist who advances a new theory to explain some phenomenon, but does not check to see if the theory is supported by experiment.

It is of some interest that issues of transportability, maintainability and reuse are being considered with respect to real-time systems (as in [Nielsen88]), but solutions to these problems should not come at the expense of efficiency and simplicity of the design, and further, solutions for say, maintainability, should not come at the expense of ease of debugging, testing or of transportability. If we cannot design efficient systems, what good is it to worry about reuse and transportability of inefficient designs?

The advantage of using the two design principles advanced in this paper is that they lead to designs with fewer concurrent threads of control, which are inherently more efficient, more easily debugged, more easily tested, more easily maintained, and more easily transported. Much more experimentation with these two principles is needed. Perhaps it is impossible to find another principle that can decide the questions of the optimum task-to-task interfacing structures to use in each individual case, but this should be a goal. Currently, there is no way to decide among the many possible designs (caused by the many task-to-task interfacing structures) within a given task structuring without extensive experimentation. For instance, it is not possible to decide between the design of Figure 2 and the design of Figure 3 without implementing them both and measuring their performance. The ability to eliminate or reduce this effort is highly desirable.

It is likely that the task structuring principle of using physical concurrency can be refined somewhat. The principle as it stands is easy to apply once one understands what the real-world objects and processes are that need to be modeled. What is needed is a better approach for deciding the objects and the processes. Naive answers to this question have been put forward in the past, but the problem is essentially unsolved.

Finally, it should be mentioned that if the proposals for asynchronous task-to-task communication are adopted in Ada 9X, most of the decoupling tasks (i.e., the buffer and transporter tasks) of Buhr and Nielsen and Shumate could be replaced by asynchronous task communications. In this case, the Nielsen and Shumate designs might begin to resemble our alternate designs. In this light, their decoupling tasks might be viewed as an artifact of not having asynchronous task communications in the Ada language. However, as shown in this paper, it is not necessary to have asynchronous task communications in order to avoid these decoupling tasks. Consequently, the unnecessary use of decoupling tasks is more of a design method shortcoming than a deficiency of the language.

9. REFERENCES

Buhr84	Buhr, R. J. A. 1984. *System Design with Ada*, Prentice-Hall, Englewood Cliffs, New Jersey.
GAO89	General Accounting Office Report GAO/IMTEC-89-9, March 1989.
Gomaa84	Gomaa, H. 1984. *A Software Design Method for Real-Time Systems*, Communications of the ACM, Vol. 27, No. 9, September.
Howes89	Howes, N. R. and Weaver, A. C. 1989. *Measurements of Ada Overhead in OSI-Style Communication Systems*, IEEE Transactions on Software Engineering, Vol. 15, No. 12, pp. 1507-1517, December.
Myers76	Myers, G. 1976. *Composite/Structured Design*, Van Nostrand Reinhold.
Mili89	Mili, A. 1989. *Review of Designing Large Real-Time Systems with Ada by K. Nielsen and K. Shumate*, IEEE Computer, pp. 124-125.
Nielsen86	Nielsen, K. 1986. *Task Coupling and Cohesion in Ada*, Ada Letters, Vol. 6, No. 4.
Nielsen88	Nielsen, K. and Shumate, K. 1988. *Designing Large Real-Time Systems with Ada*, McGraw-Hill, New York.
Sanden89	Sanden, B. 1989. *Entity-Life Modeling and Structured Analysis in Real-Time Software Design - A Comparison*, Communications of the ACM, Vol. 32, No. 12, pp. 1458-1466, December.
Sha90	Sha, L. and Goodenough, J. 1990. *Real-Time Scheduling Theory and Ada*, IEEE Computer, pp. 53-62, April.
Young82	Young, S. J. 1982. *Real Time Languages: Design and Development*, Ellis Horwood Limited, England.
Yourdon78	Yourdon, E. and Constantine, L. 1978. *Structured Design*, 2nd ed. Yourdon Press, New York.

Distribution List for IDA Paper P-2488

NAME AND ADDRESS	NUMBER OF COPIES
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. John Solomond Ada Joint Program Office Room 3D114, The Pentagon Washington, DC 20301-3081	1
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
IDA	
Dr. Cy Ardoin, CSED	1
Dr. David Carney, CSED	1
Ms. Anne Douville, CSED	1
Dr. Dennis Fife, CSED	1
Dr. Karen Gordon, CSED	1
Ms. Audrey Hook, CSED	1
Mr. David Hough, CSED	1
Dr. Norman Howes, CSED	36
Dr. Richard Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	1
Dr. Reginald Meeson, CSED	1
Dr. Richard Morton, CSED	1
Ms. Sylvia Reynolds, CSED	2
Mr. Clyde Roby, CSED	1
Dr. Richard Wexelblat, CSED	1
Dr. Robert Winner, CSED	1
Mr. Jon Wood, CSED	1
IDA Control & Distribution Vault	3