An Architectural Perspective of Real-Time Ada Applications

C. Douglass Locke

Lockheed Martin Corporation, Chief Scientist - Systems Support, Owego, NY 13827, USA
doug.locke@lmco.com

Abstract. While there are appear to be many ways in which real-time Ada systems can be designed, it is observed that they can be described using four architectural families: the Timeline, Event-Driven, Pipeline, and Client-Server architectures. This paper describes the principal characteristics of each of these architecture families with respect to their ability to provide bounded application response times, their cost, and safety at a high level. In addition, the use of important Ada constructs for each architecture family is discussed, and examples of application domains that use each of these architectures are identified.

1 Introduction

There are probably as many ways to use the Ada language as there are system and software architects and designers. The Ada language contains a number of very powerful constructs that can be used as tools for the architect. This is especially true for the software architect of a real-time system. The choice of Ada concepts to be used for a real-time application is strongly affected by the underlying software architectural choices.

Regardless of whether Ada is being used as the programming language or not, there are actually very few unique software architectures in general use by real-time software architects. These basic architectures have their roots in various traditional approaches to certain application domains, and they have naturally been carried over into systems for which Ada is the primary language. In many cases, these architectures have become so entrenched that the original reasons for their use has been lost in relative antiquity, and conscious tradeoffs are no longer made when new applications in the same domain are considered.

It is the purpose of this paper to discuss these basic software architectures for real-time systems used with (or without) Ada, comparing and contrasting them along several lines:

- The choices of Ada architectural constructs that are generally used for each
- The ability of the resulting architecture to meet its real-time constraints
- The response of applications using that architecture to errors encountered during system operation
- Costs associated with the architecture

These discussions are, of course, somewhat subjective, but they are based on this author's experience working with a wide variety of systems over a long period of time. Nevertheless, the reasoning on which these discussions are based should become clear, and it is expected that the reader will be able to assess the basis for the conclusions drawn.

It is noted at the outset that virtually all the basic software architectures used in practical real-time systems can be categorized at a high level into only four families, although in practice there are a many variations of these. In this paper, we denote these architectural families as:

1. **Timeline** (sometimes called a cyclic executive)
2. **Event-driven** (with both periodic and aperiodic activities)
3. **Pipeline**
4. **Client-Server**

Each of these four architectural families will be discussed in some detail.

Once a system's software architecture is defined, and when Ada has been chosen as the implementation language, there are many decisions that must be considered. While this paper cannot provide a definitive treatment of all of these, some of the most important architectural decisions involve:

- Ada Tasks (How many? How they should be used?)
- Communication (Shared memory? Messages? Protocols?)
- Synchronization (Protected objects? Rendezvous? Semaphores? Mutexes?)
- Shared Data
- Generics (Should thy be limited in some way?)
- Package Size and Content
- Exceptions (At what level should they be handled? What about "others"?)
- Extent of Application-Specific Types (When should predefined types be used, if at all?)
- Interface to Environment (e.g., POSIX, SQL, Motif, CORBA)

Of these decisions, only the first three will be discussed in this paper due to limited space.

In addition, it is instructive to illustrate how these decisions are frequently made in application domains such as air traffic control, aircraft mission processors, vehicle simulation systems, and flight control.

The remainder of this paper is organized into 4 sections. Section 2.0 contains an overview of each of the architectural families, while Section 3.0 discusses the major Ada decisions involved for each family in the light of these application domains. Section 4.0 then contains a brief summary and conclusion.

2 A Taxonomy of Real-Time Architectures

It is observed that virtually all real-time systems can be classified into four architectural families. Although there are many variations within these families, each of the families can be described in relatively concrete terms regarding the management of system resources (e.g., CPU, memory, I/O, and communications) which characterizes the resulting application's ability to meet its time constraints. Here, we discuss each of these families individually.

2.1 Timeline

The timeline architecture is at once the oldest real-time architecture, and conceptually the simplest. It is sometimes called a **frame-based** architecture because of its use of fixed time frames within which all the application procedures are executed in some predetermined sequence. It is also called a **cyclic executive** architecture because of its use of a simple time-driven executive that manages all application procedure invocation using a frame sequence table.

Essentially, the timeline architecture requires dividing the application into a fixed set of procedures that are called sequentially by a relatively simple executive, triggered by a timer set to expire at fixed intervals. At each time trigger, the executive uses the frame sequence table to call the appropriate procedure to the handle the task that is ready for that interval. The individual time intervals are called *frames* or *minor cycles*. The frames are grouped into a fixed sequence that is then repeated throughout the system execution; such a group is called a *major cycle*.

For example, consider the simple example illustrated in Figure 1. Here, we have a trivial application composed of four periodic procedures, each executing at rates of 40 Hz., 20 Hz., 5 Hz., and 1 Hz., respectively. The architect has chosen a frame, or minor cycle length of 25 ms., which is equal to the fastest procedure's 40 Hz. rate. To meet all the application rate

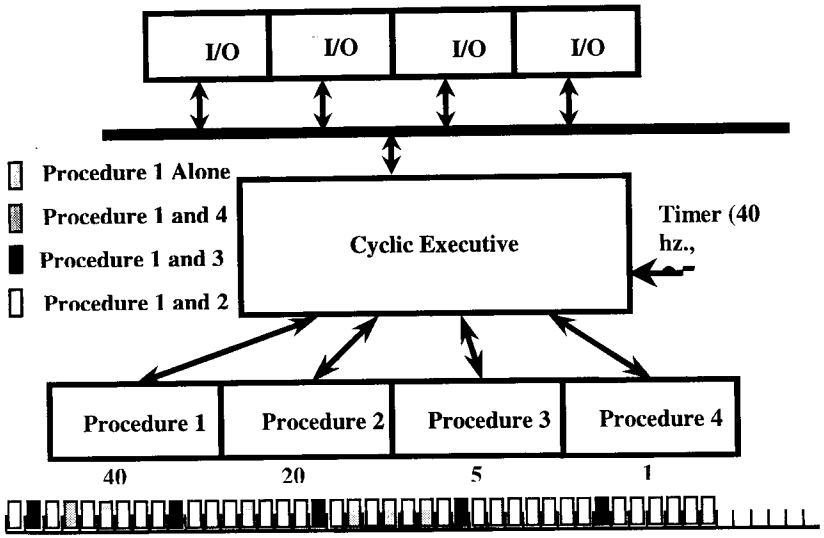

Figure 1

requirements, the procedures are executed in groups as shown, such that procedure 1 is executed every frame, procedure 2 is executed immediately following procedure 1 in the first frame of each major cycle and every other frame thereafter, procedure 3 is executed immediately following procedure 1 in the second frame and in every eighth frame thereafter, while procedure 4 is executed immediately following procedure 1 in the fourth frame and in every 40^{th} frame thereafter. In this way, each procedure is executed at its required rate.

Note that this architecture has several interesting properties. First, there is never any need for two procedures to execute concurrently, so there is no need for explicit synchronization. Because there is no need for concurrency, there is no overhead paid for context switching at the application level. Thus, at first sight, it appears that this architecture is inherently quite efficient, which is generally considered to be a desirable quality for a real-time architecture. Second, because it can be precisely determined at design time exactly when each procedure is to be executed, the system can be said to be deterministic.

However, as noted in [1], these advantages remain largely illusory. Efficiency is not a primary requirement for real-time systems, and neither is determinism. While efficiency and determinism can be useful, they must frequently be compromised (slightly) in favor of creating an architecture with predictable response behaviour. While it is true that a deterministic system will be able to meet its timing constraints, it really isn't necessary to precisely determine when each procedure is to be executed; rather it is sufficient that each procedure be able to predictably meet its own response requirements. Further, while eliminating concurrency and removing context switch overhead seems desirable, it can readily be shown that it is easy to limit the blocking delays caused by synchronization, and to bound the overhead from context switches, particularly using Ada 95, so that it is not necessary to eliminate them.

In fact, eliminating concurrency comes at an astonishingly high price. The timeline architecture requires every procedure to fit comfortably into its frame. Even though there may be (and frequently is) quite a bit of unused time available elsewhere in the schedule, this unused time is unavailable to individual procedures. Thus, long-running tasks must be manually broken into small procedures such that each will fit into the frame. This really amounts to the programmer having to create preemption points manually, then ensure for each resulting point that the task's state will remain sufficiently consistent between invocations. In other words, the programmer must do manually, with error-prone ad-hoc techniques, what any modern real-time operating system is much more capable of doing automatically with fully predictable results[4].

2.2 Event-driven

Although the timeline approach to real-time systems architecture produces the most common architectural style for real-time systems, probably the second most common architectural style is the event-driven architecture. The event-driven design (see Figure 2) waits for indications that a message has arrived, an I/O operation has completed, a timer has expired, or an external operation has resulted in an event such as a button depression. In the Ada language, this is generally seen as a rendezvous accept or a protected procedure invocation. In the event driven architecture, such events trigger the execution of all of the program's computation.

Thus an event-driven architecture consists of a set of tasks, each awaiting an event, where each task is provided with a priority. Task priorities are generally determined using either the time constraints or the semantic importance associated with the job to be done. For example, when an event arrives at a processor, an interrupt handler executing at interrupt priority will typically handle it initially. In the best designs, this interrupt handler will then execute for a minimal amount of time, triggering the execution of a secondary task. This secondary task will then run at a user-defined priority associated either with the time constraint associated with the event or based on its semantic importance alone.

The resulting event-driven architecture involves concurrency at the application level. This means that individual operations running at their correct priorities may preempt each other and results in the need for synchronization using such operations as the POSIX mutex, semaphore, or an Ada protected object. This synchronization adds complexity to the application architecture, but this is counter-balanced by the fact that concurrency increases the resource utilization levels at which the system can operate while meeting its time constraints.

The event-driven paradigm results in a significant problem for the application designer that is frequently overlooked until the later stages of system integration, frequently resulting in unpredictable response times. This problem results from the bursty arrival pattern almost always associated with events (other than events resulting from timer expiration.) The most common mathematical model for such arrivals is the Poisson arrival distribution, in which the system load produced by a burst of arrivals cannot be bounded at a predictable level over any specific time interval, resulting in an unpredictable response time. This generally results in the most common type of real-time systems failure: an intermittent "glitch" that leaves no trail, is unrepeatable, and frequently exhibits different symptoms each time it is observed.

In response to their concerns over the system response time, designers usually try to group together the total utilization generated by these random arrivals over a sufficiently long time interval to show that the system will achieve the average throughput required to handle all of these arrivals. However, this architecture pattern makes it impossible to predict the response time of individual event arrivals. This predictability problem can ameliorated by using any of several bandwidth preserving algorithms such as the sporadic server[6], but otherwise the event-driven paradigm makes it difficult to produce a predictable response time with reasonable levels of utilization.

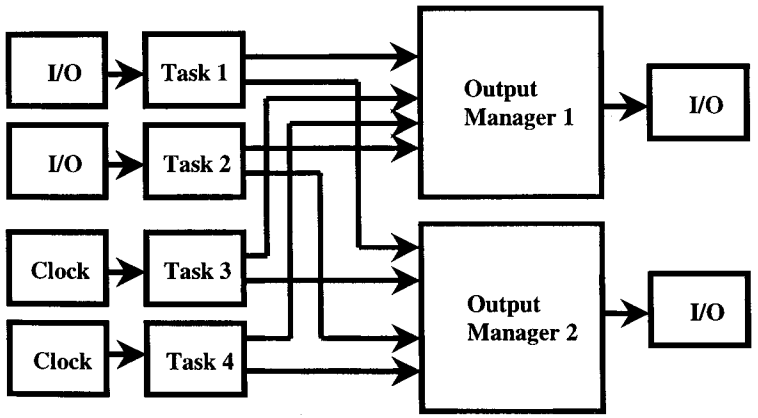

Figure 2

2.3 Pipeline

The pipeline architecture is similar to the event-driven architecture, but is intended specifically to take advantage of a distributed system by a allowing event arrivals to take place at one processor for initial processing, then pushing additional processing off to other tasks, processes, or threads in the same or any other processor in the system. Thus the pipeline architecture initially processes events but sends the remaining processing and

the final response to the events to separately scheduled entities elsewhere in the system (see Figure 3.) In the pipeline architecture, it is quite common for a single event to generate processing and many other schedulable entities, resulting in communication across many parts of network, and finally many outputs occurring at various points in the system. From the perspective above, and the time constraints involved, the most important performance parameter is the latency from the initial arrival of the event to the output of each of the resulting responses. This is referred to as the end to end time constraint of the system and may differ for each event or each type of event in the system.

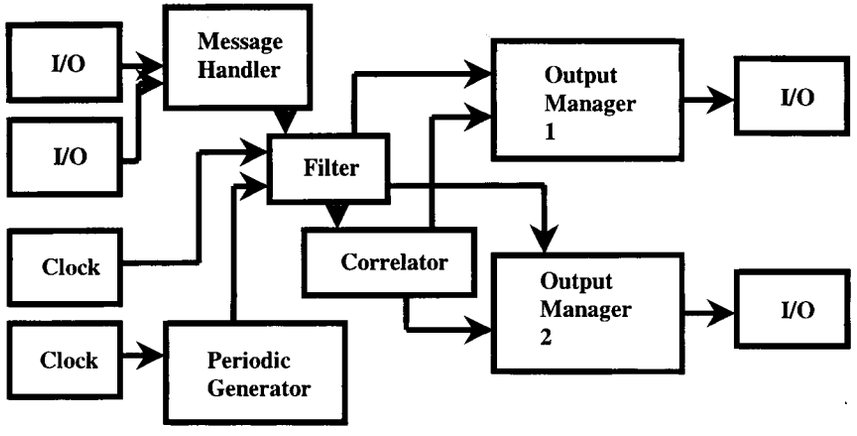

Figure 3

One of the key issues in any distributed system is the nature of the control flow. The flow of control through the system strongly affects the system load balance, the number and nature of the scheduling decisions required, and the difficulty of debugging the system as it is constructed and tested.

In the pipeline architecture, the control flow for each event moves with be event information through the system from the event source to each of the output destinations. This means that the analysis of the pipeline architecture must similarly follow the control flow sequence; the complexity of that sequence makes it somewhat difficult to predict the response time of the pipeline system. This also means that a separate scheduling operation occurs at each point in the pipeline as the event information arrives at each of the stages of the pipeline. This scheduling operation, of course, involves competition among all the schedulable entities in each processor; thus, entities handling a specific event are likely to compete not only with entities handling other events, but also with other entities handling other components of the same event.

Analysis of the resulting scheduling operations is critical to determining the end-to-end time constraints of the pipeline architecture. However because the sequence of schedulable entities handling each portion of the pipeline may, in fact, be handling multiple types of events in separate "pipes" running in opposite directions through the system, it becomes extremely difficult to define the correct priority for each schedule entity. For example, if a system is handling radar contacts eventually displaying them on an operator console, many of the schedulable entities may also be involved in handling operator control operations controlling the behavior of the radar sensors.

For these reasons, task priorities generally play only a minor role in pipeline systems. In fact, the priorities of tasks in a pipeline are frequently defined more by the direction of the pipeline flow than by analysis of the end-to-end time constraints to be achieved. For example, it can easily be shown that if task priorities increase as the event moves through a pipe, the queues of events at each intermediate stage will be minimized. In future systems, it may become possible to have the priority of the stages of the pipeline be adjustable according to the priorities of the arriving messages, thus allowing the system to preferentially a provide good response to some events while pushing others off that exhibit less stringent time constraints. At present, however, this priority handling mechanism is not generally available in commercial system infrastructures.

2.4 Client-Server

The client-server architecture is similar to the pipeline architecture in that events arriving at one or more or nodes of the distributed system are processed throughout the system as needed, based on the event type. Unlike the pipeline architecture, however, the control flow for the client-server model generally remains at the same node as the initial event handler. Although the event still arrives at an initial schedulable entity, the successive processing stages are invoked using remote procedure calls from the initial task, rather than being invoked using a one-way message (see Figure 4.) This means that the locus of control for any given event remains with the initial event handler, and all further processing, including responses, are made by one or more server entities, frequently at other nodes in the system.

Thus, the response time of the client-server architecture can be analyzed in the same way as for the pipeline architecture, but the infrastructure controlling the scheduling of each of the entities involved for each event is somewhat different. This is the architecture used, for example by the CORBA[1] standard. At present, the Object Management Group is completing a significant extension of CORBA, called *Realtime CORBA 1.0* [2], that will provide for priority propagation and scheduling throughout a real-time client-server system. This standard is expected to make the construction and performance analysis of real-time client-server systems significantly more robust.

At present, even with the CORBA real-time extensions, the only mechanism for managing the response time of the client-server system is the use of priorities. As with the other real-time architectures described here, priority can be determined using either response time requirements or semantic importance, but it has generally been found that it is more effective to assign priorities to messages rather than to tasks or threads. The message priorities can then propagated to the clients. This is one of the mechanisms available in the forthcoming CORBA real-time extension.

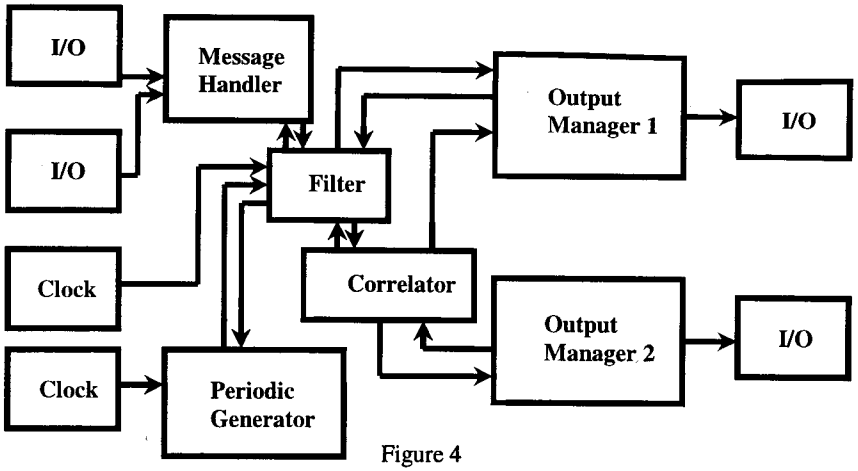

Figure 4

An additional benefit of the client-server architecture is the ease with which the system can be debugged relative to the pipeline architecture. With the pipeline architecture, it is difficult to determine the location of the control flow for a given event at any point in time, so debugging the processing of any event, either to correct response time problems or functional problems, can be very difficult. With the client-server architecture, because the sequence of operations is entirely controlled by a single thread, the location of a malfunctioning element needing further analysis becomes significantly easier to identify.

3 Using Ada in Real-Time Archtectures

There are many ways to utilize the Ada language in the construction of real-time architectures, but there are a few particularly common approaches that can be described for the architecture families described here. For each of the approaches, the Ada constructs commonly considered are briefly described, along with a few sample application domains that use or are considering using each architecture family.

3.1 Timeline

In the timeline architecture, the principal Ada construct used is simply the Ada procedure call. In the simplest systems, a small control program (a cyclic executive) is constructed in a single Ada task that executes periodically at the minor cycle rate. In Ada 95, this task would use the **delay until** statement to control the timing of each minor cycle. The cyclic executive maintains a count of which minor cycle is being initiated at each delay expiration, and uses a frame sequence table to determine which procedure(s) should be called for that cycle. Thus, concurrency is avoided and no further Ada tasks are needed. Of course, this also means that there is no need for the priorities, rendezvous, protected objects, asynchronous transfer of control, etc.

This architecture is most commonly used for such applications as aircraft and missile flight control, many avionics mission processors, industrial production controllers, and

other relatively simple applications. The minimal use of the more complex Ada features makes this approach quite common for safety-critical systems for which both the application and the run-time infrastructure must be certified.

3.2 Event-Driven

The event-driven architecture is generally constructed in Ada using a set of tasks for each event source or type. For periodic events, the task consists of a loop governed by the **delay until** statement. For aperiodic events, the task will frequently consist of a loop governed by an accept statement driven either by a rendezvous from another task or directly by an interrupt arrival from the run-time environment.

A well-designed event-driven real-time application will use no more tasks than is necessary to handle events with different time constraints. The most common Ada design errors found in event-driven systems is the use of too few tasks (i.e., attempting to handle multiple events with different time constraints in a single task,) or the use of too many tasks (e.g., using the task concept for encapsulation.) Ada tasks introduce overhead in the form of context swapping and synchronization, but when they are used to handle separate time constraints, this overhead is readily bounded and the system's ability to predictably meet its time constraints at high utilization levels is greatly enhanced.

Task priorities should generally be chosen to be consistent with such scheduling methodologies as Rate Monotonic Scheduling[3] or Deadline Monotonic Scheduling, thus providing for analysis leading to a predictable system. Proper use of the Ada protected object for task synchronization and the Ada 95 Real-Time Annex to handle priorities can result in a highly robust system that can meet all its time constraints. For aperiodic events, the use of a sporadic server (generally implemented using a **delay** statement in conjunction with the **accept** statement that triggers the event-driven task, can bound the processor utilization that can otherwise make event-driven architectures subject to timing anomalies under bursty load.

Event-driven system architectures are now frequently supplanting timeline systems because of their greatly increased robustness in meeting time constraints at high resource utilization levels, and significantly lower maintenance costs. Common exceptions are applications requiring safety certification, since the certification agencies commonly mandate the timeline approach. This is expected to change over the next few years as the predictability made possible with the event-driven approach becomes more widely understood, and as tools such as Ada Ravenscar Profile[5] become more widely available.

3.3 Pipeline

The pipeline system, like the event-driven system, is generally constructed using a set of Ada tasks, one for each stage of each of the pipelines. Synchronization between these tasks can be handled using the protected object, but the management of priorities is more difficult. This is because the decomposition of the system functions into tasks is likely to result in tasks that must be executed by event arrivals with multiple time constraints. This is a source of widespread priority inversion, resulting in systems that can miss time constraints even at relatively low utilization levels.

One way to improve the processing of tasks that receive messages from sources with disparate time constraints is to arrange for such tasks to have incoming messages arriving in multiple queues, one for each time constraint. Then the task checks each queue in order of increasing time constraints (non-preemptive Deadline Monotonic order), minimizing the resulting priority inversion. A further useful practice is to perform only minimal processing in tasks characterized by arrivals with disparate time constraints, pushing more extensive processing load to tasks handling activities with single time constraints (and appropriate priorities.)

Presently, the pipeline approach is being used to construct command and control systems, air traffic control, and occasionally vehicle simulation systems. It has also been occasionally used for such application domains as satellite ground systems and submarine combat control systems.

3.4 Client-Server

The client-server architecture is constructed similarly to the pipeline architecture in terms of its use of Ada tasks, but the propagation of messages involves waiting upon a return message to the sender in response to each message sent, thus utilizing a remote procedure call. Ada 95 describes such processing in the Distributed Systems Annex, although implementations have not been quick to support it. In the meantime, systems being built to a client-server architecture are using the Ada bindings available for CORBA, and are expected to use the capabilities provided by the Realtime CORBA 1.0 extensions when they become available. Thus, the client tasks can be expected to be defined as they would be for the pipeline architecture, while the propagation of priorities to server tasks is handled using CORBA mechanisms.

The client-server architecture has not yet been widely used by Ada real-time applications, but with CORBA, it is expected that this architectural model will be used by many of the application domains previously implemented using the pipeline approach. This would include such application domains as air traffic control, command and control, industrial automation, and supervisory systems.

4 Conclusions

The choice of architecture family for a particular system has a major effect on many of the most critical success factors for a real-time system. For uniprocessor or federated processor applications, the most likely candidates are the timeline or event-driven architectures; however, except for safety critical systems, the best choice in terms of application internal complexity, maintainability, reliability, and life-cycle cost, is a well-designed event-driven approach[4]. For distributed applications, the current choice is likely to be the pipeline approach, with careful attention to message priority management, communications latency (generally bounded only stochastically), and processor utilization. This results from the immaturity and lack of resource management support from existing infrastructures for real-time client-server approaches. With the availability of the Real-time CORBA extensions, CORBA will provide a strongly viable client-server alternative for many distributed applications.

Provision of a robust synchronization alternative in Ada 95, the protected object, as well as other Ada 95 changes described in the Real-Time Annex, make Ada highly suitable for either uniprocessor or distributed processor support of real-time systems. The Ada facilities can be used with any of the architectural choices presented here. Of course, not every facility available in Ada should be used in a real-time system, but detailed recommendations are beyond the scope of this paper.

References

1. The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1996, Object Management Group, Framingham, Massachusetts, USA
2. The Joint Revised Realtime CORBA Submission, March, 1999, Object Management Group, Framingham, Massachusetts, USA
3. Liu, C.L. and Layland, J.W., Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. JACM 20 (1):46-61, 1973
4. Locke, C. D., Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives, Real-Time Systems Journal, Kluwer Publishers, vol 4, 1992
5. The Ravenscar Tasking Profile for High Integrity Real-Time Programs, A. Burns, B. J. Dobbing, G. Romanski, in Reliable Software Technologies – Ada-Europe '98, Lecture Notes in Computer Science, Springer-Verlag, June, 1998
6. Sprunt, H. M. B., Sha, L., Lehoczky, J.P., Aperiodic Task Scheduling on Hard Real-Time Systems, Real-Time Systems Journal, Kluwer Publishers, 1989