The Journal of Real-Time Systems, 4, 37-53 (1992)

Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives

C. DOUGLASS LOCKE
International Business Machines, 6600 Rockledge Dr., Bethesda, MD 20817

Abstract. We contrast the software architecture of a hard real-time application using a fixed priority task structure against the software architecture of the same system using a cyclic executive structure to satisfy hard real-time deadlines in response to a set of embedded system requirements. We identify the perceived and actual advantages and disadvantages of both approaches, consider the types of applications which can take advantage of these approaches, and make recommendations related to the attributes of such applications that might be needed with both approaches. We conclude that the fixed priority approach, when priorities are assigned using rate monotonic priorities, generally dominates the cyclic executive approach for hard real-time systems.

1. Introduction

In current implementation practice, there have traditionally been two basic approaches to the overall design of application systems exhibiting hard real-time deadlines: the cyclic executive (also called a time-line) architecture, and the fixed priority architecture. The names are derived from the structure of the underlying executive software and its application interface, but the consequence of using either of these architectures is that the fundamental software structure of the resulting application design is determined by the chosen architecture. It is the purpose of this paper to characterize the properties of the application which are determined by the resulting structure imposed by each of these approaches.

Recent papers such as those by Baker and Shaw [1] and Sha and Goodenough [6] have significantly changed the software engineering perspectives on how these two competing architectures can be exploited by application designs, and have described some of the individual characteristics of the resulting application software structures with respect to their abilities to predictably meet time constraints, their ease of successfully completing integration and test, and the cost of their subsequent maintenance. However, papers describing either of these architectural choices do not describe their relationships in sufficient detail to allow designers to draw conclusions on their appropriate use in application designs; it is the purpose of this paper to provide the global perspective of both of these approaches needed to draw these conclusions. It is assumed that when a fixed priority task structure is to be used, the priorities will be assigned using a rate monotonic assignment model, since such an assignment can be shown to be optimal in the sense that if any fixed priority assignment can generate a successful schedule, a rate monotonic assignment will generate one [5].

This paper is intended to provide an accurate comparison of the application architectural characteristics resulting from these two fundamental architectural choices, drawn from the current literature as well as from extensive experience designing, building, and maintaining such systems. For each software architectural approach, we will identify its strengths and

weaknesses and make specific recommendations for the use of each approach based on the real-time attributes of the application domains involved.

We begin by defining the principal application design objective which most directly determines the software architecture to be used for a given application:

> *the architecture must be capable of providing a provable prediction of the ability of the application design to meet all of its time constraints.*

This objective differs from the frequently expressed objective of *determinism*. Although determinism is usually not explicitly defined in such instances, it is frequently used to describe a system in which the actual execution time and sequence is completely predetermined and fixed for the entire execution of the system. While it is certainly true that determinism is sufficient for predictability, determinism is not necessary to achieve predictability. In addition, our objective specifies provability of its predictability, but does not necessarily require that a proof be accomplished. In this case, provability means that the design techniques used must be derivable from rigorously defined execution models which produce provable results within the constraints of the model.

There are two observations that will materially assist in placing this discussion into its proper perspective:

1. Our principal discussion will revolve around the scheduling of the CPU resource which has remained the traditional subject of scheduling discussions for real-time systems. However, in a broader sense, it is critical to understand that scheduling actually refers to the concept of sequencing the use of *any shared resource whose use involves meeting application time constraints*. The traditional cyclic executive design has captured this idea in that every aspect of the application is statically scheduled, including all its resources such as I/O, networking, and CPU scheduling using the basic structure defined by the cyclic executive. Similarly, proponents of the fixed priority structure (using rate monotonic scheduling) also describe their work in terms of a total set of resources to be used, although this has not always been made completely clear.
2. Whichever overall design approach is selected, the *complete application design structure will be dictated by the choice of the cyclic executive or fixed priority scheduling paradigm.* Thus, for a cyclic executive structure, the application is divided into a set of procedures which are assumed to be non-preemptible, because preemption is not required for a uniprocessor cyclic executive design. The cyclic executive itself then calls these procedures, one at a time, according to their positions in a predetermined time sequence. Similarly, using the fixed priority approach, the application is divided into a set of prioritized tasks; the scheduling decision for each resource is determined exclusively by the priority of the tasks. In the fixed priority scheduling paradigm, these tasks are generally preemptible; thus careful attention must be paid to the handling of resources shared by the tasks so that their consistency requirements, as well as their time requirements, are guaranteed to be met.

It must be noted that accurate knowledge of the worst-case execution time for each task is required for predictability with either design approach. In the following discussions of

each approach, we will discuss the implications of understating the execution time, and alternatives for handling the resulting overload condition.

Throughout this discussion, we will refer to a simple example application. To permit its use in illustrating the advantages and disadvantages in system design, it will be vastly simpler than realistic applications, but it will contain elements with considerable similarity to many of the actual real-time applications which can be designed using these two paradigms.

Consider an industrial controller processor which is performing two tasks, each task handling one of two inputs. The first task samples the temperature of a chemical reaction at a constant frequency of 10 hz, while the second responds to any of several switch depressions made by an operator to control the process. The application periodically outputs valve control commands in response both to the sensor readings, and to the operator decisions. It is necessary that the outputs computed from the sensor inputs be generated within the same 100 ms. cycle in which the sensor is read; it is also required that an operator command result in a control action within 75 ms.

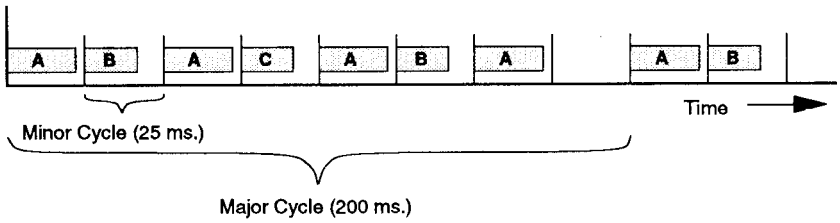
Thus we have defined the universe of discourse in which we shall compare these executives. In Section 2, we describe the cyclic executive, including its structure, the resulting structure of the application with which it works, and the detailed advantages and disadvantages which it presents to systems using it, including its recommended uses in the light of its strengths and weaknesses. Similarly, in Section 3, we describe the rate monotonic scheduling approach, including its advantages, its disadvantages and its recommended uses. In Section 4, we conclude by comparing and contrasting these two approaches.

2. The cyclic executive

As illustrated in Figure 1, the cyclic executive executes an application which is divided into a sequence of non-preemptible tasks, invoking each task in a fixed order throughout the execution history of the program.[1] The cyclic executive repeats its task list at a specific rate called its cycle, or major cycle in the common situation in which all tasks do not execute at the same frequency. When the frequencies are not identical, the task list defines a sequence such that each task is repeated sufficiently often that its frequency requirement is met. In this case, the execution of each individual task is called a minor cycle, and the frequency of the major cycle will be set to the least common multiple of the frequencies of each task [1].

The period of every minor cycle is generally set at the same value to simplify the implementation, which can then be triggered by a periodic timer interrupt or by some other periodic external hardware interrupt which keeps the program in synchronization with its external environment. At each activation, each application task is required to complete its execution within its minor cycle period, which is also called a *frame*. The term *frame* is especially used in systems in which the minor cycle period is not a constant.

Thus, each application task becomes fundamentally periodic at a fixed rate which is determined by its positions in the cyclic execution list (or time line). A common and simple design for a cyclic executive is to load the period of the minor cycle into a count-down timer which generates an interrupt when it reaches zero. If the system is behaving properly (i.e., it is not overloaded), the software will be in a wait state or background task at the



Task	Period
A	50 ms.
B	100 ms.
C	200 ms.

Figure 1. Typical cyclic executive time sequence structure.

time the interrupt is generated, causing the cyclic executive to immediately execute one or more tasks (sequentially) whose periods are ready to begin. A processor overload can be detected when the executive observes that the interrupt has occurred when the software is not in the wait state or a background task.

Thus, the frequency of the timer interrupts must be equal to, or greater than, the frequency of the highest frequency task(s), and it follows that every task must operate at integer sub-multiples of the timer interrupt frequency. This means that the frequencies of all the tasks scheduled by a cyclic executive must always be related harmonically.

Assuming that every task completes at or before its previously determined worst-case execution time, timing predictability results from determining that each task can be fit into a corresponding frame at the correct frequency. As previously stated, this determination is dependent on accurate knowledge of the execution time of each task.

With reference to our controller example, a cyclic executive application would contain two cyclic tasks within the timeline, one for each of the two inputs. However, we have a common dilemma here; the response time requirement for the operator input, which must be used to determine its execution frequency, is not harmonically related to the sensor input frequency. We have, then, three fundamental choices, illustrated in Figure 2:

1. Operate the timeline at a frequency which is a common multiple of the two tasks' frequencies (e.g., in this case, with a minor cycle period of 25 ms.), placing the operator task in the timeline at least every 3 minor cycles, and the sensor task every 4 minor cycles. Note that because these tasks have non-harmonically related frequencies, their frames will occasionally overlap, generating conflicts in their schedules. This is resolved in our sample problem by placing task B (i.e., the operator response task) one frame early when the conflict occurs in order to ensure that it meets its timing requirements. In this case, the result is an increase in the major cycle to 400 ms., rather than 300 ms. which would have represented the least common multiple of their frequencies.
2. Operate one of the tasks at a slightly higher frequency than its natural frequency (e.g., handle the sensor every 75 ms.).

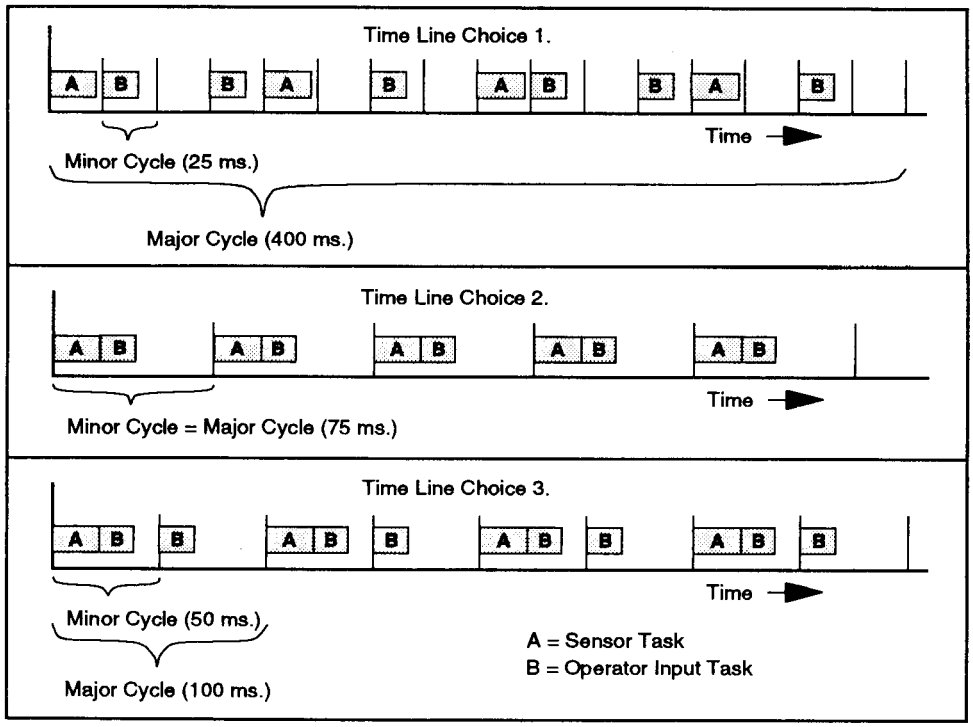


Figure 2. Timeline choices for cyclic executive design.

3. Some combination of both (e.g., handle the operator input every 50 ms. and process the sensor every other cycle.

All three choices are commonly made in existing designs, but each has drawbacks. If the faster minor cycle (choice 1) is chosen, the additional clock inputs produce higher overhead by the executive, with no added functional capability for the application, and the execution time for either task cannot exceed 25 ms. In addition, the problem we encountered with frame conflicts in this case is fairly common; the solution has the effect of increasing task B's frequency slightly (6 executions every 400 ms., rather than 4 every 300 ms.). If task B requires 20 ms. maximum to execute, for example, its processor utilization increases from about 27% to 30%, resulting in a loss of 3% total available processor load.

If the decision were made to execute one of the tasks more frequently to bring them into a harmonic relationship (choices 2 and 3), significantly higher processor utilization will result, with no increase in functional capability for the application. For example, if the operator task execution time bound is 35 ms., changing its period from 75 ms. to 50 ms. changes its contribution to the overall processor load from 47% to 70%, a loss of 23% of the total available processor load. In practice, such design choices are frequently made, but are commonly made so early in the system requirements definition process, by arbitrary specification of harmonic time relationships, that the significantly increased system load remains unnoticed.

From a timing perspective, we must observe that there is one common error situation which must be either prevented or carefully managed in every application constructed under the cyclic executive: frame overrun. A frame overrun is a condition in which a task execution time exceeds its frame (observed by noting that the minor cycle timer interrupt has occurred when the software is still executing an application time-line task). This situation is frequently considered to be a problem only during the initial test phase of the system implementation rather than a general execution-time problem, since it is assumed that eventually the task will be completely debugged and thus will not generate frame overruns. This view is quite dangerous, however, since there are many reasons for frame overruns which make them difficult to prevent. Moreover, it has been observed that virtually every practical system will encounter frame overruns at some point during it lifetime, frequently under unanticipated high load stress (i.e., the time when the correct system execution is most critical).

Correctly handling a frame overrun is extremely difficult. There are basically two choices: Either the cyclic executive allows the task to overrun, potentially slipping the entire remainder of the time line, or it aborts the task generating the overrun. Both approaches are dangerous. The first risks a random timing failure in virtually any part of the system, likely to appear in an unrelated system function, while the second has the likelihood of creating an inconsistent state both in the persistent data within the overrun task and in data it shares with other tasks. For example, in our sample application, if the operator task is allowed to exceed its frame, the sensor sampling rate will be reduced, violating its timing requirement, and possibly compromising industrial safety. On the other hand, if the operator task is terminated to prevent the overrun, it may have left a control state vector in an inconsistent state, causing the sensor task to mis-control the system in a subsequent period. Thus, apparent "intermittent" software failures (indistinguishable from intermittent hardware faults such as might be caused by a bad solder joint or a failing integrated circuit gate) are a likely result of either approach. It is exactly such problems which generate heated debates between software and hardware personnel in the course of integrating most real-time systems.

2.1. Advantages of the cyclic executive

There are several advantageous characteristics of this technique for structuring a real-time application:

1. It is clear that the entire future execution schedule is predetermined, assuming a frame overrun does not occur. This means that it is possible to predict the entire future history of the state of the machine, once the start time of the system is determined (usually at power-on). Thus, assuming this future history meets the response requirements generated by the external environment in which the system is to be used, it is clear that all response requirements will be met. Thus it fulfills the basic requirement of a hard real time system.
2. Because no individual task can ever be interrupted by other application tasks during its execution, there is no requirement for application task preemption; thus, executive overhead can be kept low because there are no unexpected context switches. In fact,

there is no necessity for allowing external interrupts (other than the timer interrupt which triggers each minor cycle), since no task will process the event until its cycle begins, when it can poll for the condition that would have generated the interrupt synchronously. In addition, as long as a shared memory multiprocessor is not used, this means that there is no necessity to protect the integrity of shared data structures or other shared resources by the provision of mutual exclusion (e.g., rendezvous, semaphores, monitors). The net result of this set of related advantages is that the resulting software is simple, efficient, and generally fast.

3. Because of the determinism in the schedule throughout the entire application execution, it is clear that it is possible for it to exhibit very low *jitter* in the execution of each periodic task. Jitter refers to the variation in the time a computed result is output to the external environment from cycle to cycle.[2] That is, a low-jitter computation triggered at the start of each cycle will end with a highly precise time relationship relative to its previous completion time and its next succeeding completion time, such that there will be a suitably small change (as defined by the external environment) in the time variation from frame to frame. For tasks with low jitter requirements, it is necessary either to limit the difference between the best and worst-case execution times, or to divide the task into two separate tasks. One task would perform the computation, while the next, triggered by a separate timer event, would output the result. Controlling jitter turns out to be quite important for certain classes of real-time systems such as feedback control systems, for which the stability of the basic transfer function depends on the tight control of jitter in the feedback loop, or for applications dependent on specialized I/O devices which require precise timing relationships between inputs and outputs.

2.2. *Disadvantages of the cyclic executive*

From the perspective of the resulting software architecture, using a cyclic executive has several disadvantages:

1. The most serious disadvantage is the application fragility produced in a software design using the cyclic executive. By "fragility," we mean that any change to the system, either during the integration of the system (due to software errors or changes in its requirements), or later during the maintenance of the system when additional functionality is to be implemented, will generally produce the result that some part of the application will exceed its frame time bound as defined for its original cyclic schedule.

   For example, we might need to add an additional sensor to our sample application to help control the process (e.g., we might need to read the reaction pressure every 30 ms.). The sampling period of this sensor will require modifying not only the basic time line to insert the additional task, but will require modification to both of the existing tasks, even if their functions are to remain unchanged. The time bounds of all the tasks are now modified, since there will be additional shared data to consider, and all timing relationships, so carefully hand-crafted in the original time-line design, will be potentially affected, thus necessitating a new analysis and extensive retest.

Assuming that we are doing sufficient analysis to enable prediction of a frame overrun, let us consider the case of an existing cyclic executive controlled application in its maintenance phase for which we have a new requirement added to one of its periodic tasks which we have determined will take us into a potential frame overrun condition. There are two basic design choices at this point. The first is that we can completely restructure the application; that is, we may go back to the beginning of our time analysis and define a completely new task sequence. If this is successful, all deadlines will again be met.

The second approach is to identify a second task with a similar period but with a sufficiently large unused time in its frame to execute the added function. This is a much more commonly used method because it involves the least risk of unforeseen timing problems and is generally accomplished by encapsulating the new function into a procedure to be called at the end of the second existing task. From the perspective of the correctness of the result, it is sufficient to ensure that any variables newly shared by the modified tasks will be left in a sufficiently consistent state that they can be used by any other tasks needing shared access to them.

This approach, however, results in severely compromising the functional integrity of the program. Each such change must take into account not only the previous design of every task, but all previous such changes to both tasks. As additional changes are made in this way, it becomes more and more likely that future changes will result in unforeseen functional and timing inconsistencies. It has been repeatedly observed that after some number of such maintenance cycles the program must undergo a major revision or rewrite in order to render it usable over the long term. Many existing hard real-time systems have undergone multiple major rewrites over their lifetimes largely because of such a loss of functional and conceptual integrity.

2. Another problem with the cyclic executive approach is the handling of functions whose execution time is long compared to the period of the highest rate cyclic task. This is a common situation in systems with multiple sensor requirements in which a sensor with a long period requires significant processing to extract the information, while another sensor requires a high sampling rate but with significantly less computation. The most common solution to such a problem is to arbitrarily break the long task into multiple short ones which can be fit into multiple frames between the high rate sampling frames. This practice, while it solves the immediate problem, is actually a process of preempting the long task by manually inserting (and later manually maintaining) the preemption points. As with any other form of preemption, the issues of consistency of shared resources managed by such tasks must be accounted for, so that inconsistent results are not obtained by other intervening tasks. This is frequently handled by using multiple input and output buffers in a way which is completely analogous to the way such synchronization problems are handled in systems with fully preemptible tasks.

2.3. Cyclic executive summary

Thus we see that the cyclic executive model, subject to problems of high fragility and difficulties handling frame overrun, results in a conceptually simple application design which,

by virtue of its fundamental determinism, can be shown to meet its time constraints, along with a natural ability to produce excellent jitter characteristics.

At this point however, we must note that while these cyclic executive advantages appear initially compelling, most of the advantages of the cyclic executive are found to be illusory, while its disadvantages are found to result in major, excessive software life-cycle costs.

For example, the advantage of determinism is realized only if the system actually requires determinism. In most hard real-time systems, however, the actual requirement is one of predictability of meeting response requirements, not determinism. By this we mean that it is not necessary that we are able to entirely predict the future history of the system in order to determine that the response requirements are met.

Another advantage cited is low overhead, due to low context switch cost, resulting from the absence of task preemption. The fact that many cyclic executives use procedure calls to effect context switches for task invocation does indeed result in low overhead. There are, however, other hidden overhead sources which, in a large number of cases, overshadow the savings in context switch overhead. One of these hidden overhead sources was previously cited when we discussed the manual process of breaking lengthy tasks into multiple partitions in separate frames; in this case, the preemption overhead has simply been manually moved out of the executive, and into the application, where its maintenance becomes both more expensive and more error-prone.

A further hidden cost of using a cyclic executive is a consequence of the fact that the cyclic executive requires all tasks to share a harmonic relationship. This effect is presently so pervasive that harmonic relationships are frequently stated in high-level system requirements even though no analysis has been done validating the frequency requirements. An example is a common requirement that displays should be updated at a rate such as 20 hz to maintain apparent continuous motion (20 hz is harmonically related to other common requirements such as a 40 hz sensor update rate). For many displays and phosphors, tests have shown that a frequency of about 16 hz is sufficient for continuous motion. If, for example, the display update computation requires 10 ms., the use of this artificially high periodic rate costs 4% of the total processor utilization for just this one task. This means that 40 ms. will be wasted each second; a processor can perform a large number of "expensive" context switches in 40 ms., especially with current processors.

Thus if we look at the advantages we have described for the cyclic executive, we see that, in general, only the advantage of ease in supporting very tight jitter requirements remains. But even here, the most frequent requirement for tight jitter refers to the timing of task outputs, which means that the jitter requirement will be met only if the execution time of the task is nearly identical each cycle. Since this will not generally be true due to various effects such as data-sensitive algorithms, DMA operations, instruction pipelining, and cache misses, the output data must frequently be buffered and output by a separate task in a subsequent frame to achieve low jitter. This solution is equally applicable to either the cyclic executive or the fixed priority executive as is discussed in Section 3, which does not result in excessive life-cycle costs. Additionally, in most applications, low jitter is not a requirement for every event in the system; tight jitter requirements are typically present only for a very small set of the actual response requirements. Most requirements with significant response time components (e.g., display updates, position or status measurements and computation, trackball or mouse management) do not have any specific timing requirements beyond completing prior to their next cycle.

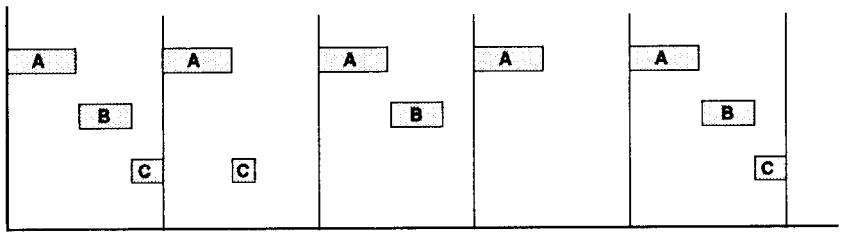
3. The fixed priority executive

Fixed priority executives and operating systems have been used in real-time systems for many years, although the ability to reason about their response time characteristics has been rather weak compared to the cyclic executive approach until recently. In the past, the selection of priorities has been done using the semantic importance of each task. Thus, tasks deemed to be critical to the success of the system have been granted a high priority, while less important tasks execute at lower priority. In terms of predictability, it is easy to determine the ability of the highest priority task to meet its deadlines, but determining the timing predictability of lower priority tasks has generally been left to the system test phase of a project.

In recent years, there have been numerous papers describing the use of rate monotonic scheduling as a paradigm for designing fully predictable hard real-time systems. Rate monotonic scheduling is simply the use of a preemptible, fixed priority executive to execute a set of periodic tasks whose priorities are ordered monotonically increasing with the rate (i.e., frequency) of the task. The use of rate monotonic priority assignments were first defined and analyzed for periodic tasks by Liu and Layland [5] as optimal for fixed priority real-time systems, in the sense that if a task set is schedulable (i.e., can meet all of its timing constraints) using a fixed priority executive, it will be schedulable using rate monotonic priority assignments.

Thus, in this section we describe the attributes of an application design using a fixed priority executive handling a hard real-time application whose priorities are ordered rate monotonically.

As illustrated in Figure 3, the fundamental real-time application model used for rate monotonic scheduling consists of a set of periodic, independent, preemptively scheduled tasks for which the deadline of each task is defined to be the start of the following period. The principal result described by Liu and Layland[5] is that when a rate monotonic schedule (i.e., a set of fixed[3] priorities in which the priorities are defined in the order of increasing



Task	Period
A	50 ms.
B	100 ms.
C	200 ms.

Figure 3. Typical fixed priority (rate monotonic) time sequence structure.

periodic frequencies) is used for a set of tasks under this application model, every task can meet its deadlines as long as the total utilization[4] of the processor does not exceed a value dependent on the periodic relationships of the tasks. This value is generally in the range of 88% to 98% [2], but can never be less than *ln2* (approximately 69%).

Until recently, however, this result has not been considered to be generally practical by the real-time application community because its fundamental model (i.e., independent periodic execution of all tasks) was considered to be inappropriate for existing real time applications. Recently, this original work by Liu and Layland has been greatly extended, principally by researchers at Carnegie Mellon University and the Software Engineering Institute [7]. Their work extends the model to include not only periodic tasks, but also aperiodic tasks with time constraints, including tasks which are dependent on each other for shared data, I/O devices, or other resources, and therefore must (at least occasionally) be synchronized.

The goal of the rate monotonic scheduling work is the same as the fundamental goal of real-time system design itself; to make it possible to produce a system with the ability to predictably meet its timing constraints as well as its functional performance. Thus, an analysis of a fixed priority application using rate monotonic priority assignments, as described in [8], is capable of providing complete system timing predictability. The application structure consists of a set of periodic tasks, a set of aperiodic tasks [4] (each with its response time requirement), and a set of shared, synchronized resources whose internal integrity or consistency are to be protected using critical sections handled with software mechanisms such as mutexes or the Ada rendezvous. Unlike applications designed using the cyclic executive, this model does not require that all periodic task frequencies have a harmonic relationship.

With reference to our sample problem, a reasonable software structure using a rate monotonic model would consist of two preemptible tasks. One is a periodic task executing every 100 ms. which handles the sensor input, processing, and output. The other executes in response to the operator input. The choice of priorities is related to the required response time; the aperiodic operator input task is higher priority because it must respond within 75 ms., while the periodic sensor task has a lower priority with a response requirement of 100 ms. The critical problem is to correctly handle the management of the aperiodic input, since its untimely (possibly rapid) arrival could cause the processor to be temporarily overloaded and prevent the periodic sensor task from meeting its time constraints. A simple and robust mechanism to handle this is the sporadic sever [6] which is a facility to allow the task to execute immediately in response to an aperiodic event (such as an interrupt), but which prevents it from repeating for some time interval after its occurrence. This facility, implementable either within the executive or within an application task, provides for fast response at a suitably high priority related to its time constraint if inputs do not arrive more quickly than specified, and slower response otherwise. In any case, the response of all tasks can be fully predicted using the rate monotonic task model.

In parallel with the frame overrun problem discussed in the context of the cyclic executive model, the rate monotonic model also must address the problem of either transient or persistent overload. Unlike the cyclic executive, the problem of overload is not isolated to an arbitrary time frame, but rather is expressed as an overall utilization in excess of the applicable utilization bound. Even if individual tasks exceed their execution bounds, as long as the overall execution bounds are not exceeded, the time constraints will continue

to be met. In any case, however, if the analysis shows that a potential for overload exists, or if there is a possibility that a task might exceed its individual execution budget, the situation can be anticipated. This can be done at several levels depending on the type of errors to be handled. For example, the operating system can provide detection of missed deadlines, or exceeded execution budgets (using such mechanisms as signals or Ada exceptions). These detected problems can then be handled by the application itself, which can take steps to ensure the consistency of the tasks' internal and shared data, possibly reducing processor load by adjusting task frequencies or occasionally skipping cycles.

3.1. Advantages of the fixed priority executive using rate monotonic scheduling

From the perspective of the application software architecture, there are several intrinsic advantages of the fixed priority executive approach:

1. The principal advantage, which strongly affects the total life cycle cost of the system, is the predictability of the entire set of tasks relative to meeting the application's externally defined requirements, even when these requirements are changing. Since this predictability is based only on knowledge of the total processor utilization of the task set, determining the schedulability of a system when an additional task is added requires recomputing only the total schedulability bound and determining whether the new utilization caused by the additional functionality causes the new bound to be exceeded. If the new bounds are not exceeded, there is no possibility of impacting the execution of any of the functionally unaffected tasks and therefore the schedulability of the system is unaffected. On the other hand, if we now find that schedulability bounds are exceeded, we know from the rate monotonic theory which tasks' schedulability can no longer be assured, and we can predict the performance of the system with the added capabilities.

   If it is determined through this prediction that the application does not meet its minimum requirements (i.e., one or more of the critical tasks will no longer be assured to complete in time), the rate monotonic theory also provides tools with which to analyze potential corrective actions, such as the use of additional processors, faster processors, or reduction of functionality.
2. It has already been noted that this technique does not require the use of harmonic frequency relationships among the periodic tasks. Thus, it becomes possible for the periodic tasks to use frequencies which are natural to the external application requirements and not suffer a performance degradation. Such an efficiency is generally impossible using a cyclic executive, both because of the forced harmonic relationships in the periodic rates, and because of the necessity for each frame to be sufficiently underutilized that the variabilities in executive time caused by interrupts, DMA, instruction prefetch, memory caching, and other effects can be handled.
3. The rate monotonic approach allows the structure of the application task to more accurately reflect the application requirements, since there is no need to arbitrarily break up individual tasks in order to meet frame length limitations. Rather, we can let tasks be executed for as long as necessary, constrained only by their own utilization specifications. Using preemption in the rate monotonic approach, there is no danger that a long

running low frequency task will cause a high frequency task to miss its time constraints, as long as the total utilization bound has not been exceeded.

4. A rate monotonic schedule exhibits a high degree of stability, meaning that in the event of an overload (presumably predicted by analysis using the rate monotonic task model), the task(s) which will miss their time constraints can be pre-identified. This stability derives from the fact that, in an overload, those higher priority tasks whose utilization has not exceeded the rate monotonic utilization bound are guaranteed to continue to meet their time constraints. Tasks at or below the priority at which this utilization bound is exceeded may miss their time constraints; further analysis can be performed to determine which will meet them and which will not. Moreover, techniques are available for ensuring that critically important tasks whose time constraints might be missed in transient overloads can have their priorities adjusted to ensure that they will meet their requirements, and for analyzing the consequences on the remainder of the application [7]. This characteristic means that timing surprises need not take place during system integration, test, or maintenance.

3.2. Disadvantages of the fixed priority executive using rate monotonic scheduling

As with any non-optimal technology, there are disadvantages:

1. The most important potential disadvantage results from the property of periodic systems previously described for the cyclic executive: jitter. Generally, the rate monotonic application model does not define a tight timing constraint on the actual completion time of a task, other than ensuring that it completes prior to the end of its period. This is also true of the cyclic executive approach, but the variability of completion time of a correctly executing cyclic executive controlled task is limited to its frame, ordinarily a small portion of its period, while the completion time variability of a rate monotonic controlled task is limited only by its period. As a result, there may be a significant jitter generated by the rate monotonic schedule, especially for a low rate (hence, low priority) task. By this we mean that the start and completion times of a task may be delayed arbitrarily due to preemption by higher priority tasks. The rate monotonic schedule predicts the ability of each task to complete prior to its next period, but does not guarantee exactly when during each cycle that completion will occur (except for the highest priority task, which will have no problem meeting a tight jitter requirement) in a rate monotonic application.

   Within this model, however, a tight jitter requirement can be handled in several ways. If only a single task has such a requirement, it can be arbitrarily given a sufficiently high priority to ensure that its requirements can be met, modifying the prediction analysis by adding its execution time as a blocking factor to tasks whose rate monotonic priority would otherwise have been higher [3]. Alternatively, the task can be separated into partitions which can meet the jitter requirement in the same manner described previously for meeting low jitter requirements using a cyclic executive. For example, assume that the jitter requirements involve the time of output of a feedback computation; a separate output task (with very low execution time and very high priority) can be created to accept buffered data from the normally prioritized computation task and perform the output

operation. Standard synchronization techniques [8] can be used, if required, to ensure the availability and consistency of the output data.

In the context of our example, suppose that the sensor task were required not only to output its computed response prior to its next period, but must output it exactly 95 ms. after the availability of the input value (perhaps within, say, ± 1 ms.). This can be done by triggering a very high priority task with the same interrupt used to schedule the sensor task, having it perform a 95 ms. delay (such as the Ada **delay** command, as long as the Ada run-time environment has sufficiently accurate clock resolution, and implements a preemptive delay expiration) operation, then write the result of the sensor task from its buffer. As long as the priority of this output task is sufficient that it can never be delayed beyond 1 ms., its timing requirement will now be met. Ensuring this is quite simple using a rate monotonic analysis [6].

2. Many hard real-time applications must be coded in Ada. For applications implemented using the Ada tasking model and the Ada rendezvous for synchronization, a rate monotonically supported application requires modifications to the typical Ada run time environment to support the model, even though the tasking model and rendezvous were intended from the outset to support time-constrained applications. This is a specific problem for the 1983 Ada standard because of the fact that Ada included scheduling in its domain, unlike almost every other language used for real-time systems, but there was an incomplete understanding of the impact of its scheduling model on the ability to meet time constraints. Similar problems for other language environments (such as C) are handled entirely within the underlying operating system or executive which are typically under close supervision by the hard real-time application designer.

   This effect with respect to Ada derives from the assumption under the rate monotonic model that priorities are used for all resource management and thus the use of FIFO rendezvous queueing, non-priority sequenced **select** statements for rendezvous, and other structures in Ada which do not always use priorities to control the sequences of the underlying resources, result in unpredictable behavior. The changes to Ada run-time environments required to support this model, however, do not affect the ability of a run time environment supporting them to pass the Ada validation tests, and do not violate the fundamental principles underlying the Ada concurrency model. It is expected that the current Ada 9X update effort will remove some or all of these problems. In the meantime, applications can avoid these effects by avoiding the rendezvous (as is done for cyclic executive designs), or by avoiding rendezvous queue build-up, such as by assigning high priorities to server tasks accepting rendezvous.

3.3. Fixed priority executive summary

Thus, from a software engineering perspective, the use of a fixed priority executive to drive the architecture of a hard real-time application results in the ability to support a greatly increased separation of the concerns of program correctness from the concerns for meeting real-time constraints. The application model does not require that individual task designers must be responsible for ensuring that other tasks' time requirements are met, such as by observing arbitrary frame time limits. Additionally, it will never be the case that a function

from one task must, during maintenance, be placed arbitrarily into another task to avoid compromising the overall schedulability of the system.

In the past, the requirement for providing arbitrary preemption of low priority tasks by higher priority tasks has been stated as a reason for avoiding the fixed priority executive in favor of the cyclic executive [1] due the perception that it requires a more complex timing mechanism to trigger tasks, as well as additional complexity for saving and restoring task states. On the contrary, the timing mechanism is insignificantly more complex than for the cyclic executive, since both use either a timer expiration or an external hardware-generated interrupt to trigger task executions. If multiple tasks with different (harmonic or non-harmonic) frequencies are being triggered by a processor timer expiration, it is merely necessary to use a time-event queue which contains a time-ordered list of tasks to be triggered; when the timer expires, the timer interrupt handler schedules the affected task, removes it from the head of the time-event queue, and sets the timer to the next event time in the queue. Such timer designs are very efficient, and have been in use for many years.

With respect to the additional complexity of saving and restoring task state, note that the state must be saved and restored in any case when a timer (or other) interrupt is handled. If this state is saved in a reserved area associated with the task, the interrupt handler returns to a dispatcher to restore the state of the highest priority ready task, rather than unconditionally restoring the state of the interrupted task. Thus, the additional complexity can be very small.

In addition, hard real-time application designers are increasingly using commercially available real-time executives rather than custom executives, virtually all of which provide these timer and context switch mechanisms, as well as the basic fixed priority task management required for a rate monotonic application structure.

4. Conclusions

Thus we have come to a perhaps surprising conclusion: we find that for hard real-time applications consisting of multiple, conceptually concurrent tasks, executing either periodically, or in response to load-constrainable aperiodic stimuli,[5] the rate monotonic approach dominates the cyclic executive approach with respect to the resulting software engineering considerations. As described in the detailed advantages and disadvantages of each approach, we find the cyclic executive to be the model of choice only for a rather small set of real-time applications; those hard real-time applications for which a low jitter requirement pervades many tasks and includes tasks with multiple frequencies. In addition, such an application must be one for which the solutions described in Section 3.2 do not apply: this means the application must have tasks with tight jitter requirements which require so much execution time that applying them as a blocking factor to high rate tasks, or inserting separate buffered I/O tasks, would render the system non-schedulable. It is likely that there are few such applications, if any.

For all other hard real-time applications in this domain, we therefore recommend the use of the fixed priority approach using rate monotonic priority assignments as extended in the references cited. The overall advantages include being able to predict the ability to meet application response requirements, the overall system efficiency resulting from the

use of natural, non-harmonic periodicity, the ability to generate a system which represents an increased separation of timing and functional concerns, and a robust structure when modification is required during system maintenance.

It is important to note that there are many other scheduling and application software structures described in the literature, such as Earliest Deadline scheduling [5], and many others derived from it, which have important and potentially useful characteristics. The two paradigms described in this paper are merely the two simplest, and at present, the two most likely to be considered by application designers, or supported by real-time executive vendors. In the future, as more advanced techniques are considered or adopted by application designers, the software architectural properties resulting from them will need to be considered in a similar way to determine their usability and costs in practical system designs.

Acknowledgments

The author gratefully acknowledges the insightful comments of the referees, which have resulted in many significant improvements in the paper, as well as a number of helpful suggestions made by Liu Sha in his review of an early draft of this paper.

Notes

1. This order remains fixed unless a mode change occurs which necessitates adding or deleting tasks to or from the active set. The problem of handling mode changes is an interesting topic which is generally orthogonal to the problem of whether a cyclic executive or a rate monotonic structure is to be utilized, so it will not be discussed further in this paper.
2. The name "jitter" is derived from the visual effect of observing the output signal on an oscilloscope which is being triggered by the timer event controlling the computation; If the output is not generated at precisely the same interval each cycle, the signal appears slightly to the left or right of its expected time, jittering on the oscilloscope time baseline.
3. In a similar way as for the fixed task sequence executed by a cyclic executive, these priorities remain fixed unless a mode change occurs which necessitates changing periods or time constraints. Again as before, the problem of handling mode changes is an interesting topic which is generally orthogonal to the problem of whether a cyclic executive or a fixed priority structure is to be utilized.
4. The utilization of a task is the ratio of its worst-case execution time to its period; the total utilization is the sum of the individual task utilizations.
5. All aperiodic stimuli must be load-constrainable, either by the software design, or by the external environment, regardless of the software architecture or executive structure used, if hard real-time constraints are to be met.

References

1. T. P. Baker and A. Shaw. 1989. The Cyclic Executive Model and Ada. *Real-Time Systems: The International Journal of Time-Critical Computing Systems* 1, (1):7–25.
2. J.P. Lehoczky and L. Sha. 1986. The Average Case Behavior of the Rate-Monotonic Scheduling Algorithm, School of Computer Science, Technical Report, Carnegie Mellon University.

3. J.P. Lehoczky, L.Sha, and Y. Ding. 1989. The Rate Monotonic Scheduling Algorithm-Exact Characterization and Average Case Behavior. *Proceedings of the Real-Time Systems Symposium*, Santa Monica, CA. USA: pp. 166–171, IEEE, December.
4. J.P. Lehoczky, L. Sha, and H.K. Strosnider. 1987. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *Proceedings of the Eighth Real-Time Systems Symposium*, pp. 261–270, IEEE, December.
5. C.L. Liu and J.W. Layland. 1973. Scheduling Algorithms for Multiprogramming in A Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1): 46–61.
6. L. Sha and J.B. Goodenough. 1990. Real-Time Scheduling Theory and Ada. *IEEE Computer*, 23(4): 53–62, April.
7. L. Sha, J.P. Lehoczky, and R. Rajkumar. 1986. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. *Proceedings of the Seventh Real-Time Systems Symposium*, pp. 181–191, IEEE, December 1986.
8. L. Sha, R. Rajkumar, and J.P. Lehoczky. 1990. Priority Inheritance Protocols—An Approach to Real-Time Synchronization, *IEEE Transactions on Computers*, 39(9), September.