The Cyclic Executive Model and Ada

T. P. Baker[1]
Department of Computer Science
Florida State University
Tallahassee, FL 32304

Alan Shaw[2]
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract:
Periodic processes are major parts of many real-time embedded computer applications. The programming language Ada permits programming simple periodic processes, but it has some serious limitations; producing Ada programs with real-time performance comparable to those produced to date using traditional cyclic executives requires resorting to techniques that are specific to one machine or compiler. We present and evaluate the cyclic executive model for controlling periodic processes. The features and limitations of Ada for programming cyclic executive software are discussed and demonstrated, and some practical techniques for circumventing Ada problems are described.

1 INTRODUCTION

The Programming language Ada has been mandated by the U.S. Department of Defense (DoD) as the single common programming language for defense mission-critical computer applications, which include many hard real-time systems. Some commercial developers of hard real-time systems, including avionics and industrial process control systems, have also expressed an intent to use Ada. However, many people are concerned that Ada may be inappropriate or inadequate for programming real-time software. One critical issue appears to be how (or perhaps whether) Ada can be used to express designs based on the widest used and best understood paradigm for designing software to meet hard real-time requirements – the cyclic executive.

There are three objectives and contributions in this paper. One is to define the notion of a cyclic executive, discuss its advantages and disadvantages, and present several implementation techniques. While the literature reports many applications of the cyclic executive method for controlling periodic processes, for example, in [7] and [11], nowhere can one find a complete presentation of what it is, why it is used, and what are the major issues and problems surrounding it. A second purpose is to evaluate Ada as a programming language for constructing real-time software within the cyclic executive model. We are not breaking new ground here but are bringing together material that has appeared in various places in recent years, for example [13] and [21]. The third aim is to present a variety of solutions to some of the more serious Ada problems. Most of these, such as timer control or handling frame overruns, can be treated directly with the Ada tasking features at some expense to machine independence or clarity, e.g. [16]; another approach that has the advantages of predictable and efficient performance is to use a low-level tasking package, e.g. [5].

The next section describes the cyclic executive approach to building real-time programs. After a review of the relevant Ada tasking features, Section 3 illustrates and evaluates two "standard" methods within Ada to code a cyclic executive, one using the delay statement and the second employing timer interrupts. The next section deals with mode changes using Ada and Section 5 then discusses several ways to solve the frame overrun problem. Section 6 shows how many of the Ada problems can be treated with a standard low-level tasking package.

2 THE CYCLIC EXECUTIVE

2.1 Definitions and Rationale

Periodic processes are important, if not the most important, software components of real-time computer systems. A *periodic process* consists of an "action" (i.e., a computation) that is executed repeatedly, in a regular cyclic pattern. The duration of the time interval between the possible start of one execution and that of the next is a constant, called the *period* of the process. A periodic process also has a *deadline* for completion of its action. Generally, in the absence of data buffering, the deadline cannot be greater than the period; that is, the action must be completed by the time it is due to be repeated. In many applications the deadline is assumed to be the same as the period. A third characterizing feature is the time required to execute the action, typically a worst case execution time. For scheduling purposes, a periodic process can be defined as a triple (c, p, d), with $c \leq d \leq p$, where c is the execution time, d is the deadline, and p is the period [18].

A *cyclic executive* is a control structure or program for explicitly interleaving the execution of several periodic processes on a single CPU; the interleaving is done in a deterministic fashion so that execution timing is predictable. It can be viewed as an implementation technique for a design methodology in which a real-time system consists mainly of a collection of periodic processes. The process interleaving is defined according to a "cyclic schedule".

[1]Supported in part by the the U.S. Office of Naval Research (contract # N0001487G0116).

[2]Supported in part by The National Science Foundation (grant CCR-8700435).

A *cyclic schedule* specifies an interleaving of actions that will enable processes to execute within their periods and deadlines. It is divided into one or more *major schedules*, which describe the sequence of actions to be performed during some fixed period of time, called the *major cycle*. The actions of a major schedule are executed cyclically, going back to the beginning at the start of each major cycle. Because of the periodicity constraints, the length of a major cycle is equal to the lowest common multiple of the periods of its constituent processes. Different major schedules correspond to different *modes* of operation of the system; control of execution switches between major schedules in response to real-time events.

Each major schedule is further divided into one or more *minor schedules* or *frames*. Frame boundaries correspond to points at which correct timing is enforced, via hardware interrupts generated by a timer circuit. Each frame is allocated a fixed length of time during which its sequence of actions must be executed.

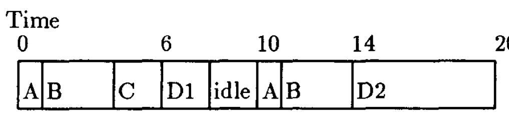
If the actions of a frame are completed early, the processor either idles or executes one or more *background* processes until the beginning of the next frame. If the actions of a frame are not completed on time, it is an error, called a *frame overrun*.

A major restriction of cyclic schedules is that no frame may be longer than the shortest period of all the processes being scheduled. For convenience and simplicity, it is common practice to require frames to be of equal length. The length of a frame is then called the *minor cycle* of the system. Note that in this case the enforcement of frame boundaries is especially simple, using a periodic timer. Because of the restriction imposed by the shortest period, any action that takes more than one minor cycle needs to be broken up into subactions, each of which is short enough to complete within one frame. An action or subaction that is selected as a scheduling unit in a frame has been called a strip, slice, scheduling block, or chunk.

Example:

Consider the four processes A=(1,10,10), B=(3,10,10), C=(2,20,20), and D=(8,20,20). (We assume deadlines are equal to periods in each case.) The action of D is divided into two subactions D1 and D2 executed in sequence with times 2 and 6, respectively. The other actions are not divided.

One acceptable major schedule for these processes, expressed in the form of a Gantt chart, is:



This schedule has a major cycle of 20 time units and a minor cycle of 10 time units. It consists of two frames, {A, B, C, D1} and {A, B, D2}. (Where there is no danger of confusion, we use a process name to denote its actions; for example, action A is the action of process A.)

There are typically several choices of minor cycle that are consistent with the a given set of processes. Given a collection of processes represented by a set of triples $\{(c_1, p_1, d_1), ..., (c_n, p_n, d_n)\}$, the requirements these impose on the minor cycle m are:

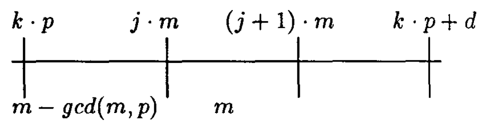
1. $m \leq d_i$, for $i = 1, ..., n$.

2. m must be greater than or equal to the computation time of the longest (sub-) action.

3. m must divide the major cycle, M.
(This is equivalent to requiring that m divide one of the p_i.)

4. $m + (m - gcd(m, p_i)) \leq d_i$ for $i = 1, ..., n$.
(This subsumes Requirement 1. *gcd* stands for the greatest common divisor function.)

The latter requirement says that (if the processes are started in phase) between every release time and the corresponding deadline there must be a complete frame. The worst case is when the release time comes just after the start of a frame. The closest these two events can be without being coincident is when they are separated by $gcd(m, p_i)$, and $gcd(m, p_i) < m$. In this case, the delay between the release time and the beginning of the next frame is $m - gcd(m, p_i)$.



It is especially interesting that the minor cycle is not necessarily as small as the greatest common divisor of the periods, nor is it sufficient that it be less than or equal to all the periods.

For example, consider three processes E=(1,14,14), F=(2,20,20), and G=(3,22,22). Requirement 1 narrows the list of possible minor cycles to the range 1...14. Requirement 2 eliminates 1 and 2, narrowing the range to 3..14. Requirement 3 eliminates candidates that are not divisible by (one of) 2,5,7, or 11, leaving only 4,5,7,10,11, and 14. Requirement 4 reduces the final list of candidates to 4,5, and 7.

Of course, the chief benefits of the cyclic scheduling model are timing predictability and simplicity. Deadlines can be enforced within the precision of one frame. By scheduling an action within a frame (or combination of frames) that lies entirely between the release-time and the deadline of the action, we can be sure the action will not be executed too early, and that the deadline will not be missed without detection. Moreover, if we have a reliable upper bound on the execution time of each action we can be assured that every process will always meet its deadline.

Implementations are particularly simple and efficient. Actions and subactions could be explicitly interleaved either by a preprocessor or manually; alternatively, the schedule could be represented by a table of action and subaction pointers, that is interpreted by the executive. In either case, context-switching at run-time is very fast. Another benefit is that the schedule can be constructed to enforce resource and precedence constraints, including exclusive access to shared resources, without the risk of deadlock or unpredictable delays.

These virtues, however, must be weighed against a number of problems, complexities, and disadvantages, that are discussed next.

2.2 Issues and Problems

One major task involved in the cycle executive approach to controlling periodic processes is to produce a *schedule*. Assuming that we are given major and minor cycle times, a set of periodic processes each characterized by a (c, p, d) triple, and a breakdown of process actions into scheduling blocks with their execution times, the problem is to generate an interleaving of the process actions or subactions to meet their deadline and period constraints. This problem is known to be NP-hard for one processor, which means that in the worst case an exponential amount of work appears necessary to determine whether a feasible schedule exists. (Non-preemptive scheduling is related to the bin-packing problem, and is discussed in [12].) Fortunately, if we do not insist on optimality, practical cases can be scheduled using heuristics.

First, it should be noted that the existence of a schedule *requires* that the *processor utilization*, defined as the sum of the ratios c/p over all processes, must be less than or equal to 1. One useful scheduling algorithm is the *rate monotonic* one; this preemptive algorithm assigns static priorities to processes based on their periods - the shorter the period the higher the priority. In a classic paper, [15] show that in the case when deadlines are identical to periods this algorithm will produce a feasible schedule whenever one exists based on static priorities. Furthermore, a schedule always exists if processor utilization is less than approximately 0.693 (ln 2). Note that this algorithm does not use any *a priori* division of actions into subactions. However, the relevant heuristic for our purposes is that one should try to fit actions and subactions into a schedule starting with that subset having the shortest periods.

A second preemptive scheduling result, that is more general, is the optimality of the *earliest deadline* scheduling algorithms [9]. This is a dynamic policy that always selects that ready process with the nearest deadline; it is optimal in that it will always produce a schedule *if* one exists. The obvious and useful heuristic for producing a cyclic schedule is to try to schedule those actions or subactions in earliest deadline order. In the example schedule of Section 2.1, the actions and subactions appear according to *both* a rate monotonic and earliest deadline for each frame; processor utilization is $(1/10 + 3/10 + 2/20 + 8/20) = 0.9$.

A principal reason that the above results are not used directly is that they require processes be preemptible at arbitrary points, rather than at points corresponding to subaction completions. Consequently, the implementation of the schedule, i.e. the executive, would need a more complex timing mechanism to trigger the next process to be resumed and a more complex context-switching mechanism for saving and restoring process states. The result may be too much execution overhead for real-time performance. Moreover, arbitrary preemption may not be acceptable due to synchronization and resource management requirements.

Scheduling becomes more complicated if one includes the additional constraints due to possible resource contention among processes or due to synchronization requirements. These constraints impose restrictions on the order of interleaving actions and subactions among processes. Common single-unit resource sharing, typically implemented by critical sections, can be handled simply by defining each critical section as an indivisible subaction [11]. An example of a heuristic technique that appears practical for tasks with resource constraints is given in [22].

Another difficulty is splitting actions into subactions, i.e., determining the scheduling blocks for each periodic process. Almost always, this is done manually, based on natural "regions" of code such as critical sections or in terms of functional units, e.g. [6]. In principle, a compiler could also produce scheduling units, for example, by defining a "basic block" as a subaction as in [10]. In both cases, manual and automatic generation of scheduling units, there remains the very difficult job of predicting the execution time of a subaction.

Timing predictability for computer programs is still very much a glimmer of hope for the future, rather than a reality. Measurements usually produce average case behaviors, analysis techniques are only just beginning to be developed for worst case results, e.g. [20], and code simulators that attempt to follow worst case paths are still considered research, e.g. [17]. Because of this state of affairs, predictions for action execution times (the c component of the periodic process characterization) and for subaction execution times are at best approximations, and often optimistic ones. Errors in timing estimates may cause frame overruns, and are the reason that overruns occur.

The response to a frame overrun varies, according to the requirements of the application. Most often, the execution of the (minor) schedule for the frame is simply terminated, and execution of the schedule for the next frame is begun. In certain cases it may also be acceptable to suspend computation of the overrun minor schedule and complete it later as background, or to complete the minor schedule, pushing back the start of the next minor schedule past the beginning of the next frame. It is common practice to log overruns, and if the number of consecutive overruns exceeds some preset threshold a fault recovery routine may be called.

Mode changes also present some difficult problems. One question is when to make a mode change. Possibilities include changing immediately upon receipt of the corresponding event (thereby interrupting the current slice), doing it at the end of the current action or subaction, changing at the end of the current frame, or waiting until the completion of the major cycle. In any case, the mode transition may require special processing to clean up or reinitialize certain variables. The latter amounts more or less to "restarting" some processes.

A final issue relates to the level of programming implied by the cyclic executive approach. Because of the need for timing predictability, interleaving of actions from independent processes, and prerun time scheduling, software construction has been a low-level activity. This appears incompatible with the more abstract process-oriented concurrent programming paradigm implemented by languages such as Ada. However, the cyclic executive technique may not be inherently incompatible with high-level programming so much as that it is incompatible with Ada's model of concurrency. For example, new tools are being developed that aim to automatically generate cyclic schedules from a high-level graphic representation of a system design. There has been some success in adapting such tools to generate Ada code, using the pure procedural subset of Ada as a lower-level implementation language.[3]

We have been implicitly assuming a single-processor execution environment. Though producing optimal schedules for multiple processors is more difficult than for a single processor, the cyclic executive model can be successfully adapted to shared-memory multiprocessor systems and to distributed systems [6, 14]; in fact, the synchronous discipline it imposes can simplify many of the problems that arise with such systems, such as achieving consensus.

We have also so far addressed only periodic processes, ignoring the other main constituent of real-time software; i.e., *sporadic* (also called aperiodic or event-driven) processes. One standard method for treating these in the cyclic executive model is to translate each sporadic process into an equivalent periodic process based on the worst case frequency of events or interrupts. One then prepares a schedule with the equivalent periodic processes included. If a particular frame includes the start action or subaction of an equivalent periodic process, then the cyclic executive makes a run-time test at the time of initiation of the frame to determine whether or not the corresponding sporadic process has been triggered. One translation scheme is derived in [18], where the equivalent periodic process P has the same execution time c as the corresponding sporadic process Q, a deadline equal to Q's execution time, and a period

$$p = minimum(ds - c + 1, ps)$$

where ds is the deadline of Q and ps is the minimum time between successive triggers of Q.

[3]One example of such a software generation tool that has been adapted to Ada is "Autocode", a product of Integrated Systems, Inc. of Santa Clara, CA.

The above method may be too pessimistic, resulting in poor processor utilization. While it does indeed handle the worst case scenario where *all* sporadic processes are triggered simultaneously by maximum frequency bursts of events, the probability of this situation occurring may be vanishingly small. An alternate method is to allocate some smaller amount of slack time in each minor cycle for possible sporadic processes and to rely on the frame overrun mechanisms for event overloads.

3 CYCLIC EXECUTIVES IN ADA: Basic Solutions and Difficulties

3.1 Relevant Ada Tasking Features

Ada provides a fairly general notion of concurrent process, which is called a "task" [2]. Though space does not permit a complete discussion of Ada tasking here, we will review those concepts that are most germane to the topic of this paper.

In contrast to the paradigm of *deterministic clock-driven* scheduling behind the cyclic executive, Ada's tasking system is based on a *nondeterministic event-driven* scheduling model. This is a serious problem for designers of real-time systems who want to use Ada. Some people have suggested that real-time systems designers must change their methodology in order to use nondeterministic scheduling. This may be advisable for some applications, but not for many *hard* real-time applications, for example, those where hardware interfaces impose tight deadlines and precise timing requirements. Moreover, synchronism often is a necessary assumption in reducing difficult mathematical problems to manageable proportions. Thus, if it comes to a conflict between Ada and deterministic scheduling, Ada may lose.

An Ada task is not ordinarily periodic; it executes indefinitely, subject to its own internal logic. The Ada Reference Manual [2] describes each task as executing on its own virtual processor. If, as is normally the case, several tasks need to share one physical processor, their executions are implicitly interleaved in some (unspecified) fashion by the Ada run-time system.

The only direct means of controlling task timing is the **delay** statement, which allows a task to suspend its own execution for a specified duration. The timing of a task may also be controlled, indirectly, through a *rendezvous* with a task of known timing. For a rendezvous to take place one task must call an *entry* of another task, and the other task must execute an **accept** statement for that entry. Whichever task attempts to rendezvous first waits for the other task to execute the complementary statement, at which time the rendezvous begins. During the rendezvous, the calling task waits while the accepting task may execute an action.

A hardware interrupt is handled as a special case of a rendezvous. The hardware device is viewed as the caller and the handler as the acceptor. Thus, if there exists a hardware timer capable of generating interrupts it can be used to control the timing of a task.

There are two mechanisms in Ada for explicitly causing a task to abandon its normal execution path. First, there is abortion. A task may be aborted by another task or itself using the **abort** statement. Second, there is the *exception* facility of the language. Examples and evaluations of all of the above features are presented below.

3.2 USING THE DELAY STATEMENT

3.2.1 Naive Solutions

At first glance, there seems to be a very simple and straightforward way of implementing periodic processes as Ada tasks, using a **delay** statement. In particular, page 9-12 of the Ada Reference Manual suggests the following. (We illustrate with the same set of periodic processes and schedule as the example of Section 2.1.)

```
task CYCLIC_EXECUTIVE_1;

task body CYCLIC_EXECUTIVE_1 is
     use CALENDAR;
     INTERVAL: constant:= 0.01;
     NEXT_TIME: TIME:= CLOCK + INTERVAL;
     FRAME_NUMBER: INTEGER:= 1;
begin loop delay NEXT_TIME - CLOCK;
           FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
           case FRAME_NUMBER is
           when 0=> A; B; C; D1;
           when 1=> A; B; D2;
           end case;
           NEXT_TIME:= NEXT_TIME + INTERVAL;
           if CLOCK>NEXT_TIME
           then HANDLE_FRAME_OVERRUN; end if;
     end loop;
end CYCLIC_EXECUTIVE_1;
```

In this example the function CLOCK, defined in the standard package CALENDAR, is used to calculate the delay required to schedule the next execution of the task in order achieve periodic execution. The constant INTERVAL is the desired period, which is the minor cycle time of the schedule.

Alternatively, the schedule could be table-driven. Instead of explicitly interleaving the periodic process actions in the executive code, we can store the schedule in a table that is interpreted by the cyclic executive [21].

These apparent solutions offer a clean abstract algorithmic description of the cyclic executive architecture. However, they have a number of more or less serious problems that make them unsatisfactory for hard real-time applications. Some of these problems arise because the scheduling task runs concurrently with the other system activities, such as background tasks.

3.2.2 Problems with Ada Semantics and Performance

The main difficulty is the accuracy and predictability of timing. The **delay** statement does not provide any upper bound on the amount of time a task may be delayed. The Ada Reference Manual says only that the execution of a task that executes a **delay** statement is suspended "for at least the duration specified". Some attempt has been made to compensate for this in the way the delay is computed in the above code. In fact, using this technique, the Reference Manual says: "... the interval between two successive interactions is only approximate. However, there will be no cumulative drift as long as the duration of each iteration is (sufficiently) less than INTERVAL" (page 9-12 of [2]). There remains the problem of essentially unlimited "jitter" – i.e., variation in the interval between executions of the action. In the case that the average duration of each iteration is greater than INTERVAL, there will also be cumulative drift. If there is a frame overrun, it is not detected until *after* the frame has completed.

Purchasers of Ada compilers intended for real-time applications are insisting (successfully) on stronger semantics for the **delay** statement. Specifically, they are requiring that the delay be implemented preemptively, with some known accuracy. Thus, there is some implementation-defined value, EPSILON, defined such that a task executing "delay D;" is guaranteed to be released (awakened) so that it is again eligible for execution within D + EPSILON seconds [4]. This interpretation is also likely to be officially incorporated in the next revision of the Ada language Standard. It does reduce the problem of jitter, provided the delayed process is able to preempt the CPU when it is released.

Since Ada allows static priorities to be assigned to tasks, whether or not a delayed task can resume execution as soon as it becomes eligible depends on the priority of the delayed task relative to other tasks that may be eligible for execution at the same time. No matter how priorities are assigned to tasks, if there are several tasks with relatively prime periods there will be times that a task "waking up" from a delay will need to wait for another task to relinquish the CPU – i.e., there will be some jitter. Predicting jitter for a periodic task thus depends on complete knowledge of the activities of all other tasks of equal or higher priority.

Even if the delayed task has higher priority than all other tasks, it will not resume execution immediately if the Ada run-time environment is in a critical section. This was a fairly serious problem in the early Ada implementations. Complex operations performed by the Ada run-time support system, such as task creation and termination, can be very time-consuming; if these are implemented as monolithic critical sections, they can add significantly to delays, in an unpredictable fashion. Some of the more recent implemented and proposed Ada run-time systems try to keep critical sections short by dividing up these longer operations, but they cannot eliminate the critical sections entirely.

Another problem area is with computational overhead, due to the generality of the Ada tasking features employed. In our example above, overhead comes from several sources: (1) the type TIME will probably require a 64 bit representation to meet Ada's requirements, so that the arithmetic used to calculate the next delay is rather expensive; (2) the

[4] A formal analysis of how this kind of delay statement semantics makes it possible to reason about higher-level language software for controlling periodic processes is presented in [20].

function CLOCK is likely to be expensive, either because it involves an I/O operation or because there must be mutual exclusion between the asynchronous operations of fetching the (64-bit) clock value and updating the clock; (3) the **delay** statement, if implemented in the most accurate way, will involve inserting the task into a delay-queue (ordered by time), possibly resetting a hardware timer, switching to another task, and eventually switching back. Even the simple use of the rendezvous feature, for example in a table-driven implementation of the executive, can incur significant overhead. The large size and unpredictable nature of these overheads has been demonstrated in the experimental study of several Ada compilers reported in [8].

Another difficulty results from the uncertain semantic relationship between the **delay** statement and CLOCK function. Presumably the calendar clock will need to be adjusted occasionally, to keep the system time synchronized with the rest of the world. Ada does not define what, if anything, happens to any pending delays when this happens. The Ada specification also omits any definition of the accuracy of the TIME value returned by CLOCK; consequently, there is no guarantee that meeting a constraint on TIME will come close to meeting the same constraint on real-time (where, for example, real-time is defined by the National Bureau of Standards "clock").

3.3 Using an Interrupt from a Hardware Clock

A second way of implementing periodic execution in Ada is to make use of periodic interrupts generated by hardware timers. Here, machine-independence is sacrificed for more precise control over timing. For example, if TIMER'address is the address associated with a regular interrupt with period TICK, the following would work:

```
task CYCLIC_EXECUTIVE_2 is
   entry TIMER_INTERRUPT;
   for TIMER_INTERRUPT'address use at TIMER'address;
end CYCLIC-EXECUTIVE_2;

task body CYCLIC_EXECUTIVE_2 is
   FRAME_NUMBER: INTEGER:= 1;
begin loop accept TIMER_INTERRUPT;
           FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
           case FRAME_NUMBER is
           when 0=> A; B; C; D1;
           when 1=> A; B; D2;
           end case;
       end loop;
end CYCLIC_EXECUTIVE_2;
```

Presumably the arrival of the interrupt will be as accurate as the hardware supports. However, if a TIMER_INTERRUPT arrives while CYCLIC_-EXECUTIVE_2 is executing, the interrupt will be blocked, according to Ada semantics. The language does not specify whether an interrupt arriving during this time is queued or lost. If the interrupt is lost, there is a serious problem, since an entire minor cycle will be skipped. If it is buffered, the problem may be less serious; the next frame will be started as soon as this one completes. Unfortunately, if the condition causing the overrun is persistent, the schedule will drift. In either case, the result may be unsatisfactory.

This problem can be circumvented by splitting the action into a separate task from the handling of the interrupt, in such a way that the interrupt is always enabled.

```
task CYCLIC_EXECUTIVE_3 is -- the task that
                           -- controls timing
   entry TIMER_INTERRUPT;
   for TIMER_INTERRUPT'address use at TIMER'address;
   pragma PRIORITY(SYSTEM. PRIORITY'last);
end CYCLIC_EXECUTIVE_3;

task ACTION is -- the task that does the work
   entry NEXT_FRAME;
end ACTION;

task body CYCLIC_EXECUTIVE_3 is
begin loop accept TIMER_INTERRUPT;
           select ACTION.NEXT_FRAME;
             else HANDLE_FRAME_OVERRUN;
           end select;
       end loop;
end CYCLIC_EXECUTIVE_3;

task body ACTION is
   FRAME_NUMBER: INTEGER:=1;
begin loop accept NEXT_FRAME;
           FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
           case FRAME_NUMBER is
           when 0=> A; B; C; D1;
           when 1=> A; B; D2;
           end case;
       end loop;
end ACTION;
```

This approach is proposed and described in more detail in [16], though Ada tasking has undergone changes since that paper was published, rendering some details obsolete. A more recent and more complete case study can be found in [13].

All solutions based on a timer interrupt require that a hardware timer be available – one that is not already in use by the standard Ada run-time environment. Since many Ada implementations use one timer to implement standard delays and another to implement CALENDAR.CLOCK, it is likely that there may be none left. Finally, few compilers today provide an efficient implementation of interrupt entries, and some do not support them at all. The solution above may therefore not be feasible.

4 MODE CHANGES

A cyclic executive can also be coded to support mode changes. As an example, consider a two mode system with mode 0 controlling the four periodic processes of our example and mode 1 controlling another set. For simplicity, we will assume that the frame sizes (minor cycle times) in both modes are identical, and that modes change only on frame boundaries. Doing mode changes at times other than frame boundaries is similar to the problem of detecting and handling frame overruns, discussed in Section 5, and is amenable to similar techniques.

```
MODE: MODES; pragma SHARED(MODE);
NEW_MODES: MODES; pragma SHARED(NEW_MODE);
MODE_CHANGED: BOOLEAN; pragma SHARED(MODE_CHANGED);

task MODE_CONTROLLER is
     entry SWITCH;
     for SWITCH use at ...; -- some input device
end MODE_CONTROLLER;

task body MODE_CONTROLLER is
begin loop accept SWITCH do
           if MODE=0
           then NEW_MODE:=1;
           else NEW_MODE:=0; end if;
           MODE_CHANGED:=true;
           end SWITCH;
      end loop;
end MODE_CONTROLLER;

task body CYCLIC_EXECUTIVE_4 is
begin loop accept TIMER_INTERRUPT;
           if MODE_CHANGED
           then MODE_CHANGED := FALSE;
                MODE := NEW_MODE; end if;
           case MODE is
           when 0 => ACTION_MODE_0.NEXT_FRAME;
           when 1 => ACTION_MODE_1.NEXT_FRAME;
           end case;
      end loop;
end CYCLIC_EXECUTIVE_4;

task body ACTION_MODE_0 is
begin -- the same code as ACTION of Section 3.3 except
      -- that FRAME_NUMBER is reset if MODE_CHANGED.
end ACTION_MODE_0;
```

One special problem related to mode-changing concerns background tasks. In typical applications a mode change will involve restarting certain background tasks associated with the new mode and may require adjusting the relative allocation of processor cycles among the background tasks. Ada provides no way of doing this.

The main weakness of the above solution is that frame overruns are not detected. This could be accomplished by using the code of CYCLIC_-EXECUTIVE_3 and moving the mode change checking and case analysis into a single ACTION task that contains separate sections each on ACTION_-MODE_0 and ACTION_MODE_1 processing.

5 HANDLING FRAME OVERRUNS

How a frame overrun should be handled often depends on the application, and sometimes the individual frame. One policy is to simply log the overrun. This is relatively simple to code in Ada, given that the overrun is detected. Other policies, such as terminating the offending frame and restarting it at its next scheduled minor cycle or suspending the offending frame and resuming it at some later convenient time, are much harder, requiring "erroneous" Ada programming.

The fundamental problem in treating frame overruns using suspension or termination policies is that one task (the one that *detects* the timing fault) must be able to asynchronously alter the control flow of another task (the one which has overrun its alloted time), so that it restarts at a known point. There are additional difficulties in properly suspending the offending task until it is scheduled again and also in doing these activities without undue performance penalties. (These problems also arise in implementing mode changes.)

Within standard Ada, two possible mechanisms are available for terminating frames; these are abortion and exceptions. We will consider abortion first.

5.1 Abortion

Abortion does not provide an adequate solution, unless the compiler performs several optimizations. Task abortion is slow, in general, due primarily to complications introduced by pending rendezvous and activations, and by families of dependent tasks. This cost can be reduced for very simple tasks, though compilers do not presently bother. A second problem is that an aborted task cannot be restarted; in particular, it is no longer callable. The closest thing to restarting such a task is to create a new task of the same type. Because the new task is not the same task as the aborted one, any tasks previously in communication via rendezvous with the aborted task will need to be informed of the replacement task's identity.

One way to avoid this problem is to make the abortable task accessible via an access variable, as follows:

```
task type ACTION is -- the task that does the work
  entry NEXT_FRAME;
end ACTION;

type ACCESS_ACTION is access ACTION;

CURRENT_ACTION: ACCESS_ACTION:= new ACTION;

task body CYCLIC_EXECUTIVE_5 is
begin loop accept TIMER_INTERRUPT;
           select CURRENT_ACTION.NEXT_FRAME;
             else abort CURRENT_ACTION;
                  CURRENT_ACTION:= new ACTION;
           end select;
      end loop;
end CYCLIC_EXECUTIVE_5;
```

Unfortunately this solution is implementation-dependent. It requires that this special case of abortion and creation of a new task be done more quickly than can be done for the general case. It also is likely to exhaust memory, since it would be very difficult for an implementation to determine that the storage of the terminated task could be reused [5]. This is a common complaint about existing Ada implementations.

An unavoidable defect is that any tasks with pending entry calls for the aborted task will have TASKING_ERROR raised in them, rather than having their calls transferred to the replacement task.

[5]UNCHECKED_DEALLOCATION is defined to have no effect for tasks.

Also the aborted task cannot pass any state information on to the replacement task except through global variables.

5.2 Exceptions

In the 1980 version of the Ada language, each task had an attribute, 'failure, which could be raised in that task by another task [1]. This feature was removed in 1983 when the current version of the Ada language standard was established. It appears the reason for this change was the belief that abortion provided an adequate substitute. If still present, this feature could be used to cut off the execution of tasks which overrun their frames, as illustrated below:

```
task body CYCLIC_EXECUTIVE_6 is
begin loop accept TIMER_INTERRUPT;
           select ACTION.NEXT_FRAME;
             else raise ACTION'failure;
           end select;
      end loop;
end CYCLIC_EXECUTIVE_6;

task body ACTION is
   FRAME_NUMBER: INTEGER:= 1;
begin loop accept NEXT_FRAME;
        begin FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
              case FRAME_NUMBER is
              when 0=> A; B; C; D1;
              when 1=> A; B; D2;
              end case;
        exception when others=> RECOVER_FROM_OVERRUN;
        end;
      end loop;
end ACTION;
```

Of course, the attribute 'failure is no longer part of the Ada standard. An implementation is free to support it, however, just as an implementation is free to support the other useful tasking features dropped from earlier versions of Ada (e.g., changeable priorities, and predefined generics for semaphores and signals).

Lacking this feature, a programmer can attempt to achieve the same effect. One somewhat *ad hoc* technique, reported by [19], makes indirect use of ACCESS_CHECK to enable one task to raise an exception (CONSTRAINT_ERROR) in another. The basic idea is for the subject task (e.g. ACTION above) to access all its data via a single global access variable, say DATA. If another task (e.g. CYCLIC_-EXECUTIVE_6 above) detects that the subject task has exceeded its time slot, it sets DATA to the value "null". The next attempt to dereference DATA will result in CONSTRAINT_ERROR being raised. Our example may then be coded:

```
DATA: ACCESS_DATA:= new DATA_RECORD;
TMP: ACCESS_DATA;

task body CYCLIC_EXECUTIVE_7 is
begin loop accept TIMER_INTERRUPT;
           select ACTION.NEXT_FRAME;
             else TMP:= DATA; DATA:= null;
           end select;
      end loop;
end CYCLIC_EXECUTIVE_7;

task body ACTION is
   FRAME_NUMBER: INTEGER:= 1;
begin loop accept NEXT_FRAME;
        begin FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
              case FRAME_NUMBER is
              when 0=> A; B; C; D1;
              when 1=> A; B; D2;
              end case;
        exception when others=>
              DATA:= TMP; RECOVER_FROM_OVERRUN;
        end;
      end loop;
end ACTION;
```

The suppositions behind this approach are that ACCESS_CHECK is not suppressed and that it is implemented economically, for example, using a hardware trap for illegal addresses. It also assumes that references via DATA are made very frequently in A, B, C, D1, and D2. Under these (implementation-dependent) assumptions, this technique would work, at the expense of much extra indirect addressing.

A second, less satisfactory method uses polling; the writer of the subject task periodically checks a global variable to determine whether it has permission to proceed. When another task detects that the subject task has exceeded its frame time, it withdraws this permission by changing the value of the global variable. When the subject task sees that its permission is withdrawn, it raises an exception in itself, unwinding outward to the point where it is to wait to be rescheduled. This polling technique is not particularly attractive, since it not only imposes significant computational overhead, but also destroys the modular purity of the coded actions.

So far we have only addressed the problem of terminating an overrun frame, so that execution can resume with the next scheduled frame. If we wish instead to temporarily suspend execution of the frame, and resume it later when there is free time, an additional mechanism is needed. Ada provides no such mechanism, short of polling; that is, the actions and subactions would need to have explicit voluntary suspension points interleaved within their code. Implementation of frame suspension therefore requires augmenting the standard Ada tasking model, as described in the next section.

6 AUGMENTING ADA TASKING

Given that programming cyclic executives in Ada requires use of nonportable features that go beyond the standard semantics, there is a good case for augmenting Ada with an adequate and separate set of low-level tasking operations. This could be provided as a package; if such a package became standard it would promote more portable and reusable real-time software. One example of such a low-level tasking package is Lace [5, 4]. We will explain some features of Lace, and show how they might be used.

6.1 The Lace Package

Tasks are identified by values of the type

TASK_ID. The value NULL_TASK stands for no task. A task can find out its own ID by calling the function SELF. A new task is created via a call to NEW_TASK. A new task is initially "held"; that is, it is not yet eligible for execution. It becomes eligible for execution when some (other) task calls RELEASE.

Normally, a processor is preemptible. That means that an interrupt handler can cause control of the processor to be transferred from its current task to some other task (other than an interrupt handler). By calling DISABLE_PREEMPTION, a task makes the current processor non-preemptible. Once a task makes its processor non-preemptible, it remains that way until the task calls DISPATCH.

Interrupt handlers may call PREEMPTION_OK to determine whether it is safe to preempt. (This attribute is set to false by DISABLE_PREEMPTION.) Having determined that it is safe, a handler can preempt by calling PREEMPT.

DISPATCH makes the current processor preemptible and gives Lace a chance to switch the processor to another task. Control is given to the highest priority task of those tasks that are not already executing on some processor and are not "held".

A task may hold another task so that it is no longer eligible for execution, via a call to HOLD. If the task is executing when a HOLD is invoked on it, it will continue executing until it is preempted through an interrupt (assuming its processor is preemptible) or until it calls DISPATCH.

Lace also provides a procedure FORCE_CALL(T,P). The effect is to force the task with ID T to call the procedure with address P at the next dispatching point. FORCE_CALL can be used to raise an exception in a task that has overrun its frame.

6.2 Cyclic Executive Examples

Lace features are employed to program frame-overrun handling in the following example:

```
FRAME_OVERRUN: exception;
FRAME_DONE: BOOLEAN;
  pragma SHARED(FRAME_DONE);

procedure RAISE_OVERRUN is
begin raise FRAME_OVERRUN;
end RAISE_OVERRUN;

task body CYCLIC_EXECUTIVE_8 is
begin loop accept TIMER_INTERRUPT do
           if FRAME_DONE
           then LACE.RELEASE(ACTION'ID)
           else LACE.FORCE_CALL(ACTION'ID,
                  RAISE_OVERRUN'address);
           end if;
           if LACE.PREEMPTION_OK
           then LACE.PREEMPT; end if;
           end TIMER_INTERRUPT;
      end loop;
end CYCLIC_EXECUTIVE_8;

task body ACTION is
begin loop begin LACE.DISABLE_PREEMPTION;
                 LACE.HOLD(SELF);
                 FRAME_DONE:=TRUE;
                 LACE.DISPATCH;
                 FRAME_DONE:=FALSE;
                 FRAME_NUMBER:=(FRAME_NUMBER+1) mod 2;
                 case FRAME_NUMBER is
                 when 0=> A; B; C; D1;
                 when 1=> A; B; D2;
                 end case;
           exception when others=> null;
           end;
      end loop;
end ACTION;
```

When a frame overrun is detected by the interrupt handling CYCLIC_EXECUTIVE_8, it causes an exception to be raised in ACTION, by forcing ACTION to call a procedure which raises the exception. The exception causes ACTION to exit to an exception handler, after which it loops back and starts the next frame. Note that it is conceivable that the exception could be raised twice, if a second timer interrupt occurs between the time the first exception is raised and the time ACTION has made its way back around the loop into the exception-handling frame of the **begin** block.

This solution would work for a single processor, but Lace was designed also to permit a multiprocessor implementation. In that context we do not know how to implement FORCE_CALL without waiting, so a call cannot be forced directly by an interrupt handler. There is also the danger of a race in which ACTION finishes the frame and suspends itself between the time the handler discovers the frame is not complete and the time it forces ACTION to raise an exception. In this case ACTION would wait until the next timer interrupt before the exception takes effect. Solving either of these problems requires waiting. Since the handler cannot wait, we need to introduce an auxiliary "enforcer" task, who can wait. When a handler wants to raise an exception in a task, it simply releases the enforcer task, which then takes care of the actual work of arranging for the exception to be raised.

In a similar fashion, the Lace operations can be used to implement restarts of background tasks, and suspend and resume overrun frames. The priorities of background tasks can also be adjusted via additional Lace operations. By selective suspensions and releases, the mix of background tasks can also be changed.

7 CONCLUSIONS

We have presented the cyclic executive model for controlling real-time periodic processes. The facilities and limitations of Ada for programming cyclic executive software have been discussed and demonstrated, and some practical techniques for circumventing Ada problems have been described.

Ada certainly was not designed to specially support the cyclic executive paradigm. Special support is not necessary, so long as the standard features are sufficient to permit programming the kinds of periodic execution required in practical real-time applications. As Ada stands, the standard features are at best marginally sufficient for this.

Using several implementation-dependent features, it does appear possible to achieve periodic execution in Ada with a reasonable degree of control, although in an awkward way. It seems desirable to concentrate machine dependent features sufficient to write a traditional cyclic executive in a single informally-standardized package of low-level tasking operations. A first step in this direction has been made by the ACM working group [3].

A more basic problem is that Ada and other higher-level concurrent programming languages are incompatible with the traditional lower level approaches to writing and controlling periodic software with hard real-time constraints. Higher level languages must have precise real-time semantics and real-time control features before they can be used conveniently for this and other real-time programming.

ACKNOWLEDGEMENT

We are grateful to Kevin Jeffay for some helpful comments and suggestions on the paper.

References

[1] *Reference Manual for the Ada Programming Language*, proposed standard document, U.S. Department of Defense (July 1980).

[2] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, Ada Joint Program Office (January 1983).

[3] Ada Runtime Environment Working Group, "A Catalog of Interface Features and Options for the Ada Run Time Environment", ACM SIGAda (1986).

[4] T.P. Baker, "A Low-Level Tasking Package for Ada", Proc. SIGAda Int. Conf. on the Ada Programming Language, Dec. 1987.

[5] T.P. Baker and K. Jeffay, "Corset and Lace: Adapting Ada Runtime Support to Real-Time Systems", Proc. IEEE Real-Time Systems Symp., Dec. 1987, pp 158-176.

[6] T.P. Baker and G. Scallon, "An Architecture for Real-Time Software Systems", IEEE Software, May 1986, 50-58.

[7] G.D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System", Comm. ACM, Vol. 27, September 1984, 926-936.

[8] R.M. Clapp, L. Duchesneau, R.A. Volz, T.N. Mudge, and T. Schultze, "Toward Real-Time Performance Benchmarks for Ada", Comm. ACM, Vol. 29, Aug. 1986, 760-778.

[9] M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes", Proc. IFIP Congress, 1974, pp. 807-813.

[10] M.D. Donner, "Control of Walking: Local Control and Real-Time Systems", CMU-CS-84-121, Dept. of Computer Science, Carnegie-Mellon Univ., May 1984 (Ph.D. Dissertation).

[11] S.R. Faulk and D.L. Parnas, "On Synchronization in Hard-Real-Time Systems", Comm. ACM, Vol. 31, March 1988, 274-287.

[12] M. Garey and D. Johnson, *Computers and Intractability*, Freeman (1979) 236-244.

[13] P. Hood and V. Grover, "Designing Real Time Systems in Ada", SofTech Report 1123-1, submitted to HQ, US Army Communications and Electronics Command (January 1986).

[14] D.M. Koch and T.P. Baker, "Verification of Cyclic Schedules for Hard Real-Time Systems", technical report, FSU Department of Computer Science (1987).

[15] C.L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", JACM, Vol. 20, January 1973.

[16] MacLaren, Lee, "Evolving Toward Ada in RealTime Systems", Proceedings of the ACM-SIGPLAN Symposium on Ada, SIGPLAN Notices, 15. 11 (1980) pp. 146-155.

[17] A.K.-L. Mok, P. Amerasinghe, M. Chen, S. Sutanthavibul, and K. Tantisirivat, "Synthesis of a Real-Time Message Processing System with Data-Driven Timing Constraints", Proc. IEEE Real-Time Systems Symp., December 1987, pp 133-143.

[18] A.K.-L. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", Ph.D. Thesis, MIT, May 1983.

[19] O. Roubine, oral remarks at the International Workshop on Real Time Ada Issues, Moreton-Hampstead, England (13-15 May, 1987).

[20] A. Shaw, "Reasoning About Time In Higher-Level Language Software", TR#87-08-05, Dept. of Computer Science, Univ. of Washington, Aug. 1987. Accepted for publication in IEEE Trans. on Software Engineering.

[21] A. Shaw, "Software Clocks, Concurrent Programming, and Slice-Based Scheduling", Proc. IEEE Real-Time Systems Symp., Dec. 1986, pp 14-18.

[22] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-13, May 1987, 564-576.