Ada* and Cyclic Runtime Scheduling

Philip E. Hood
SofTech Inc.

Abstract

An important issue that must be faced while introducing Ada into the real time world is efficient and predictable runtime behavior. One of the most effective methods employed during the traditional design of a real time system is the cyclic executive. This paper examines the role cyclic scheduling might play in an Ada application in terms of currently available implementations and in terms of implementations that might be developed specifically to support real time system development.

The cyclic executive solves many of the problems faced by real time designers, resulting in a system for which it is relatively easy to achieve appropriate timing behavior. Unfortunately a cyclic executive carries with it a very high maintenance penalty over the lifetime of the software that it schedules. Additionally, these cyclic systems tend to be quite fragile when any aspect of the system changes.

This paper presents the findings of an ongoing SofTech investigation into Ada methods for real time system development. Section 1 discusses cyclic scheduling in general - what it is and why it is used. Section 2 examines how cyclic scheduling might be applied to Ada real time systems. Methods of introducing cyclic schedulers into applications without violating Ada semantics is explicitly discussed. Several classes of cyclic schedulers will be evaluated on their compatibility with the Ada world. Section 3 briefly examines how future systems might use a cyclic scheduler without paying the high price levied upon current systems. The topics covered include a description of the costs involved in using cyclic schedulers, the sources of these costs, and measures for future systems to avoid these costs without giving up the runtime performance of a cyclic system.

1.0 Cyclic Executive Description

A cyclic executive provides a mechanism for enforcing a predetermined ordering of processing events in a system. All processing to be performed is arranged within a schedule of finite duration. This schedule is repeated at a specified rate called the major cycle. The major cycle is broken down into a number (usually a power of two) of equal minor cycles. Each minor cycle is assigned a processing frame containing a list of processing elements (routines) to be performed during the associated minor cycle. An example of the basic cyclic executive structure is shown in Figure 1.

---

* Ada is a registered trademark of the U.S. Government (AJPO)

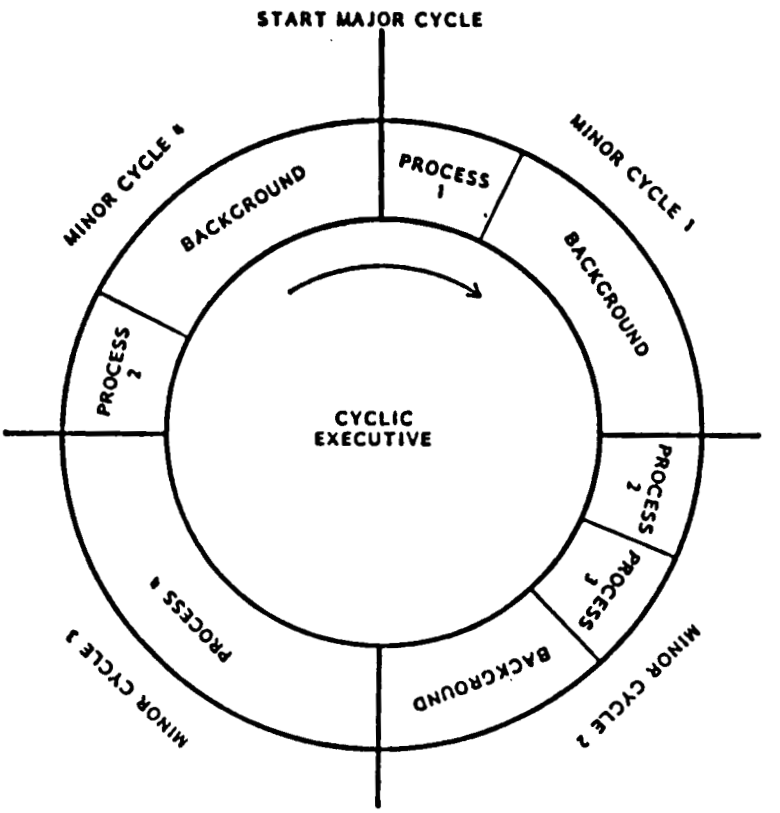

Figure 1 - An Example Cyclic Executive Structure

Although all cyclic executives share the structure we have described, they vary in almost every other aspect. Many types of cyclic executives have been developed to support various applications, and each one is different from the last. Some of these variations, such as mode changes, varying frame assignments and handling frame overruns, are discussed below.

1.1 Mode Changes

One of the advantages of a cyclic executive is that the static schedule can be tuned to optimize the system's timing performance for the expected load conditions. The load on the system, however, may not be constant. A change in the system load may cause the cyclic executive to allocate run time in a very inefficient manner (a job with a long allocated run time may have little or no processing to perform).

To solve this problem mode changes are introduced into the system. A mode change can change both the processing to be performed and the cyclic schedule. The more variation possible in the loading of the system, the more mode changing operations will be necessary. Each mode change is expensive in terms of new coding and tuning that must be performed and in terms of the damage to the program structure that always accompanies tuning operations.

1.2 Varying Frame Assignments

Schedule variations do not always require a mode change. If the variations can be localized to one frame, then that frame can use a local scheduler to resolve the problem. This solution of course, requires the overhead of some run time scheduling. Moreover, every possible scheduling possibility must be verified during system tuning.

1.3 Handling Frame Overruns

The greatest amount of variation between cyclic executives lies in the handling of frame overruns. We will consider the following four methods, by no means a complete list (many variations and hybrids exist): overruns ignored, overruns logged, overruns suspended, and overruns terminated.

1.3.1 Overruns Ignored

In some systems the problem of frame overrun can be adequately addressed during system debugging; these systems may choose to ignore overruns during runtime. The designer is responsible for verifying that overruns can never occur. This type of executive is typical of systems with either very simple software or over-confident designers.

1.3.2 Overruns Logged

This strategy is used in systems where no runtime action is appropriate when a frame overruns. The overrun is recorded for handling off line. This approach results in a very realistic executive for any system in which tuning issues can be adequately addressed. In a properly tuned cyclic executive application, frames should not be overrunning. Thus if this type of scheduler is inadequate, it implies that a cyclic schedule is not capable of providing a reliable schedule for that application and must be enhanced.

1.3.3 Overruns Suspended

When a frame overruns in this type of system, it is suspended and the next frame is allowed to start on time. When there is free time the suspended frame is allowed to complete.

This method greatly complicates data access in the application software. A built-in efficiency of a cyclic executive is the synchronization implied by static frame assignments. Additional synchronization is normally unnecessary during shared data references. When frame suspension is introduced, the implied synchronization is disrupted, and consequently references to shared data must include the appropriate synchronization mechanisms.

1.3.4 Overruns Terminated

When overruns occur in a system using this strategy, the overruning frame is terminated. It is restarted from the beginning at its next scheduled start time. This mechanism avoids the synchronization problems of the suspension mechanism but introduces its own problems. Software components that could possibly overrun frame boundaries must be written very carefully so that valuable data is not lost. There is also a potential problem with data that is incompletely updated when the frame is terminated - if this data is used by other components, serious problems could arise.

2.0 Ada Implementation of Cyclic Executives

Some varieties of cyclic executive fit very well into Ada, others do not map so naturally into the language.

The basic cyclic structure is fairly easy to implement in Ada. MacLaren [1] and Hood [2] show how to write simple cyclic executives in Ada. The basic cyclic scheduler for this type of executive is shown in Figure 2. This type of executive ignores the issues of varying loads and overrunning frames.

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#1000054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 2. A Simple Cyclic Executive

The simple structure can be easily expanded to incorporate mode changes and variable frame assignments. Figure 3 shows a cyclic executive with mode changing. Each mode is represented by a complete list of frames to be scheduled in that mode. At the beginning of each major cycle, the executive decides which schedule to run. Varying frame assignments require no change to the cyclic scheduler; instead a local scheduler is created in the varying frame as shown in Figure 4.

Overruns can be logged by adding a task to receive the periodic interrupt and to check whether or not the previous schedule has completed. This type of scheduler is shown in Figure 5.

None of the cyclic variations discussed so far has been difficult to implement in Ada. The last two variations, namely overrun suspension and termination, are considerably more difficult. In both cases, these executives could only be written if they were heavily supported by the underlying run time system.

The only asynchronous scheduling point provided by Ada occurs when an interrupt is received, so this fact must be used in both the suspension and termination variations. Asynchronous response to an interrupt is not guaranteed by the Ada specification, however any Ada implementation that has any value in the development of real time systems have to provide asynchronous interrupt handling. The only ways to terminate an executing piece of Ada code are either to raise an exception or to abort the task. Asynchronous exceptions are not allowed in the Ada semantics, leaving only the abort statement. The abort statement is not guaranteed to stop the aborted task from executing at any particular time. Therefore, a frame termination executive could be written in Ada only if the underlying implementation guarantees the immediate termination of aborted tasks.

The overrun suspension executive has similar problems. The only way to ensure the new frame will have precedence over the old one is to introduce the new frame as a task with higher priority than the old frame task. This technique works for the frames in a given major cycle, but when the first frame is reintroduced at the beginning of the next major cycle, it must wait for all the frames from the previous major cycle to complete before starting. This behavior is clearly not desirable. In order to implement this type of executive in Ada, the implementation must provide some sort of dynamic priority mechanism. Standard Ada priorities are not dynamic, thus a additional priority scheme must be introduced. These new priorities can not interfere with the workings of the Ada priority system but can be used to assign relative priority to tasks that either have no standard Ada priority, or have the same standard priority.

In general, these executives require more control over the processing resources than can be obtained using a single thread of control (single Ada task). The resulting cyclic executive must be implemented using Ada tasking facilities. Ada tasking facilities, however, lack support for the primitive (and often dangerous) functions necessary for these variations of the cyclic executive.

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#100054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
    type Mode_Type is (Mode_1, Mode_2);
    Mode: Mode_Type := Mode_1;
  begin
    loop  --forever
      case Mode is
        when Mode_1 =>
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_3;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_4;
        when Mode_2 =>
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
      end case;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 3. A Cyclic Executive with Mode Changing

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#100054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
    Status: Frame_Package.Status_Type := Frame_Package.Status_Type'First;
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4 (Status);
    end loop;
  end Cyclic_Scheduler;
end Executive;

separate (Frame_Package)
procedure Frame_4 (Status : in Status_Type) is
begin
  case Status is
    when Good =>
      Application_1;
      Application_2;
    when others =>
      Application_1;
  end case;
end Frame_4;
```

Figure 4. A Cyclic Executive with Frame Level Scheduling

```
with Error_Handling_Package;
with Frame_Package;
package body Executive is
  task Tick_Handler is
    entry Clock_Tick;
    for Clock_Tick use at 8#1000054#;
  end Tick_Handler;

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
  end Cyclic_Scheduler;

  task body Tick_Handler is
  begin
    loop  -- forever
      accept Clock_Tick;
      select
        Cyclic_Scheduler.Minor_Cycle_Tick;
      else
        Error_Handling_Package.Log_Overrun;
      end select;
    end loop;
  end Tick_Handler;

  task body Cyclic_Scheduler is
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 5. A Cyclic Executive with Overrun Logging

4.0 Avoiding the Cost of Cyclic Scheduling

Cyclic scheduling is very costly over the lifetime of a software system. The reason is very simple: cyclic systems require software to be developed in modules according to their time consumption rather than according to functional considerations. Two phases of development are totally dominated by the timing structure of a cyclic executive: detailed design/coding stage and tuning. During detailed design, frame assignments are designed and coded specifically to fit into their assigned time slots. Functionality is traded back and forth between routines and frames, depending on where there is time.

Tuning can be though of as temporal debugging, during which timing errors are found and corrected. The correction methods include dividing up existing routines and shifting functionality between frames and routines. The end result is a very fragile schedule which meets the timing requirements but suffers several drawbacks: minor changes are likely to have sufficient impact on the schedule to require complete system retuning. Functional components are so dispersed that to understand any single component requires knowledge of the entire system. The structure of the software has been totally lost and maintenance efforts can only degrade the structure further. Finally, one has a system in need of constant and expensive maintenance.

In order to reduce the cost of this type of system, the creation of the cyclic structure must be separated from the creation and maintenance of the software components. Modern software engineering techniques can be applied to the system development and maintenance issues, with an extra step added to derive a cyclic implementation from a more general design. The code developed would be structured according to functional rather than timing considerations. The timing of the system would move from the detailed design and coding steps into a new precompilation step.

This extra step might be implemented as a machine-assisted (programmer directed) set of program transformations which parallel the cyclic design process that would normally take place during the software design. The transformational sequence as well as the the untransformed source would be save for future rederivations after necessary program maintenance is performed.

A tool assisted tuning system need not be limited to cyclic transformations. While there may always be a class of real time systems requiring cyclic runtime performance, there is an equally large number of systems that do not require such extreme measures. Many of these systems would benefit from the flexibility of an Ada style runtime scheduler. This type of scheduling allows more flexibility in dealing with runtime loading variations, and is far more robust when maintenance changes are made. These systems still require tuning, although not to the same extent. For these systems, other types of tuning transformations can be made available, such as replacing monitor tasks with semaphores or simplifying groups of tasks using program inversion techniques [3]. By applying these techniques, a system can be tuned until the appropriate level of predictability and efficiency has been reached.

Conclusion

There will always be systems that have a need for the runtime performance of cyclic scheduling. Many of the cyclic scheduling models fit well within the Ada language. In order for the cost of a cyclic system to be brought under control, new methods must be developed to for their creation. These methods ought not be limited to the creation of cyclic systems; however, they should provide a more general approach to the development of real time systems, with cyclic scheduling as one of many options for achieving real time performance.

References

[1] MacLaren "Evolving toward Ada in Real Time Systems," SIGPLAN Notices, 1980

[2] Hood, Philip and Grover, Vinod, Designing Real Time Systems in Ada, Technical Report 1123-1, SofTech Inc., Waltham, MA, January 1986.

[3] Rajeev, S., On Applying Ada to Real-Time Systems: The Inversion Technique and Some Examples, Technical Report TP 148, SofTech, Inc., Waltham, MA, March 1983.

[4] Rajeev, S., Certain Optimizations in Ada Tasking Implementations Technical Report 9074-2, SofTech, Waltham, MA, January 1983.

[5] Gonzalez, M.J., Jr., "Deterministic Processor Scheduling," ACM Computing Surveys, Vol. 9., No. 3., September 1977.