Practical Implementations of Embedded Software Using the Ravenscar Profile

Stephen Michell
National research Council
1250 Grand Lake Rd. Box 5300
Sydney, Nova Scotia, Canada B1P 6L2
email stephen.michell@nrc.ca

February 20, 2002

Abstract

There is significant industry interest in the use of the Ravenscar Tasking profile for high performance and high integrity systems. This paper takes one such design paradigm, the single writer and multiple readers problem, shows why it is difficult to implement using Ravenscar, and provides an effective Ravenscar implementation.

1 Introduction

There is significant industry interest in the use of the Ravenscar Tasking profile[Burns 1999][IRTAW 1997][IRTAW 1999] for high performance and high integrity systems. The reduction in scheduling overheads and reduction in object code needing analysis (kernel footprint) provides significant savings for this community. The cost of this reduction, however, is that the complexity of code rises and the usage paradigms of features changes. Before these communities can optimally use Ravenscar there must be design patterns developed that show how the capabilities can be effectively used.

This paper takes one such design paradigm, the single writer and multiple readers problem, shows why it is difficult to implement using Ravenscar, and provides an effective Ravenscar implementation. The problem is examined in 3 forms, using a standard form with Ada (non-Ravenscar) protected objects, using a simple protected object and time-based sequencing, and finally using a multi-protected object implementation.

2 Problem Statement

The Reader-Writers problem is a classic situation that arises in real-time systems. Typically, a task produces data on a periodic basis which must be consumed by other tasks. In highly periodic systems this often occurs at the same periodicity. The readers are not permitted to read the same data twice and cannot miss a read. The writer is assumed to be free-running; readers must follow the writer, i.e. they synchronize their reads with each write, either using a common time base or an intertask communication paradigm such as a protected object.

```
package Types_Pkg is
  type Data_Type is record
    First  : Integer;
    Second : Integer;
  end record;
  Consumer_Period : constant Ada.Real_Time.Time_Span
                  := Ada.Real_Time.Milliseconds(40);
  Producer_Period : constant Ada.Real_Time.Time_Span
                  := Ada.Real_Time.Milliseconds(40);
  Number_Consumers: constant := 4;
end Types_Pkg;
-------------------------------------------
with Types_Pkg;
package Buffer_Pkg is
  protected type Buffer_Type is
    procedure Read(  D :    out Types_Pkg.Data_Type);
    procedure Write( D : in     Types_Pkg.Data_Type);
  private
    Buf : Types_Pkg.Data_Type;
  end Buffer_Type;
end Buffer_Pkg;
----------------------------------------------
package body Buffer_Pkg is
  protected body Buffer_Type is
    procedure Read(  D :    out Types_Pkg.Data_Type) is
    begin
      D.First  := Buf.First;
      D.Second := Buf.Second;
    end Read;
    procedure Write( D : in     Types_Pkg.Data_Type) is
    begin
      Buf.First  := D.First;
      Buf.Second := D.Second;
    end Write;
  end Buffer_Type;
end Buffer_Pkg;
-------------------------------------------
with Buffer_Pkg;
package Producer_Pkg is
  task type Producer_Type( Buffer : access Buffer_Pkg.Buffer_Type)  is
  end Producer_Type;
end Producer_Pkg;
--------------------------------------------
with Buffer_Pkg;
package Consumer_Pkg is
  task type Consumer_Type( Buffer : access Buffer_Pkg.Buffer_Type)  is
  end Consumer_Type;
end Consumer_Pkg;
```

Figure 1 - Implementation of Readers-Writer

Figure 1 provides the basic implementation of the situation using a producer task, consumer task, and buffer package. A common types package is included. Routines omitted include the work of the producer and the consumer, and the package which creates the protected object. It is irrelevant to the producer and consumers where the actual protected object is created since it's location is passed in as a discriminant when each task is created.

There is no scheduling in the protected object in Figure 1. Scheduling must be done individually, such as a common delay. Figure 2 provides an implementation of the producer and consumer task types which use Ada's real time facilities and delay for the same period. Unfortunately, this approach provides no guarantees and is subject to jitter because the coupling between the components is too loose and any variations can cause reads or writes to miss a turn.

```
with Types_Pkg;
with Ada.Real_Time;
package body Producer_Pkg is

  task body Producer_Type is
    Next_Period_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
    New_Data : Types_Pkg.Data_Type;
  begin
    loop
      Next_Time_Period := Ada.Real_Time."+"(Next_Period_Time,
                                             Types_Pkg.Producer_Period);
      delay until Next_Period_Time;
      Produce_Data( New_Data);
      Buffer.Write( New_Data);
    end loop
  end Producer_Type;

end Producer_Pkg;
-------------------------------------------------
with Types_Pkg;
with Ada.Real_Time;
package body Consumer_Pkg is
  task body consumer_Type is
    Next_Period_Time : Ada.Real_Time.Time := Ada.Real_Time.Time_First;
    New_Data : Types_Pkg.Data_Type;
  begin
    loop
      Next_Time_Period := Ada.Real_Time."+"(Next_Period_Time,
                                          Types_Pkg.Producer_Period);
      delay until Next_Period_Time;
      Buffer.Read( New_Data);
      Consume_Data( New_Data);
    end loop
  end Producer_Type;
end Producer_Pkg;
```

Figure 2 Typical Producer and Consumer Task Implementations

3 An Ada95 Approach

Consider the standard Ada [IS 8652:1995] approach. By changing the definition of the Buffer protected object to include a Read entry queue we can force all readers to queue upon the object until the writer has written new data. Figure 3 shows the specification and body for this new buffer. No changes to the producer or consumer code are required except that we could possibly reduce or the delay time for consumers.

In this implementation, all consumers wait in a common queue. After updating the data, the producer sets the boolean guard to True to release the consumers and exits the protected object. All queued consumer tasks take a copy of the data in turn and also exit the protected object, reducing the variable Read Count until it is 0. The last reader returns Read_Count to the number of readers and sets the Data_Available boolean to False, preventing any more reads until a new set of data is written.

There is a problem with this implementation in that one or more readers may not be available immediately for the read and another task could end up reading twice in its place. One solution to this issue is to give each task a dedicated entry (say using entry families) with its own boolean.

```
package Buffer_Pkg is
  protected type Buffer_Type is
    entry Read(  D :    out Types_Pkg.Data_Type);
    procedure Write( D : in     Types_Pkg.Data_Type);
  private
    Buf            : Types_Pkg.Data_Type;
    Data_Available : Boolean := False;
    Read_Count : Integer     := Types_Pkg.Number_Consumers;
  end Buffer_Type;
end Buffer_Pkg;

package body Buffer_Pkg is
  protected body Buffer_Type is
    entry Read(  D :    out Types_Pkg.Data_Type) when Data_Available is
    begin
      D.First  := Buf.First;
      D.Second := Buf.Second;
      Read_Count := Read_Count-1;
      if Read_Count <= 0 then
        Data_Available := False;
        Read_Count     := Types_Pkg.Number_Consumers;
      end if;
    end Read;
    procedure Write( D : in     Types_Pkg.Data_Type) is
    begin
      if Data_Available then
        raise Types_Pkg.Protocol_Error;
      end if;
      Buf.First  := D.First;
      Buf.Second := D.Second;
      Data_Available := True; -- releases readers
    end Write;
  end Buffer_Type;
end Buffer_Pkg;
```

Figure 3. Standard Ada95 Producer-Consumer Buffer

An alternative solution used here is to have each consumer delay for a while (say 1/2 the delay of the producer) and then attempt the read. By this time all other readers should have completed and the entry barrier will be properly set to False.

4 Ravenscar Implementation

The solution proposed above for Ada95 does not work in Ravenscar. Either of the variations, either one queue with multiple callers or multiple entries, is in violation of the Ravenscar Tasking Profile. If one attempts to implement it, the kernel would raise Program_Error when a second consumer attempts to call Read.

Are there other alternatives? Since there is only one producer, can we have the producer block on the entry? Unfortunately this reverses the protocol and does not guarantee that all consumers will consume the data before the producer must produce again. This method also runs the risk that the producer will block at the buffer and miss a production cycle, in violation that the producer is strictly periodic.

We could use suspension objects and semaphores, but semaphores are notoriously error prone and do not support the ceiling priority protocol [LV000].

A solution to this situation in Ravenscar is to create a single protected object for each consumer with an entry to call and be suspended. As before, the producer will update the data and release each consumer. Figure 4 shows the design of a Buffer_Pkg that has the necessary 2 types of protected objects. It contains a Buffer protected object as before, reverted back to the protected procedure interface of Example 1, plus a new protected type Reader_Type which acts as a suspension object for each consumer.

It is possible to couple both protected objects (Buffer_Type and Reader_Type), and a safe buffer management can be created, except that there is a danger of deadlock on multiprocessor systems (Ravenscar is not intended for multiprocessor systems, but such code could easily be ported to a multiprocessor system and fail). Instead we use a third protected object for the producer which knows access to both the buffer_type and the Reader_Type.

```
package Buffer_Pkg is

  protected type Buffer_Type( Readers: access Reader_Array) is
    procedure Read(  D :    out Types_Pkg.Data_Type);
    procedure Write( D : in     Types_Pkg.Data_Type);
  private
    Buf            : Types_Pkg.Data_Type;
    Data_Available : Boolean := False;
    Read_Count     : Integer     := Types_Pkg.Number_Readers;
  end Buffer_Type;

  type Reader_Access_Type is access all Reader_Type;
  type Reader_Range is range 1 .. Types_Pkg.Number_Consumers;
  type Reader_Array is array(Reader_Range) of Reader_Access_Type;
  protected type Reader_Type(
        Buffer : access Buffer_Pkg.Buffer_Type)  is
    entry Read(  D :    out Types_Pkg.Data_Type);
    procedure Release;
  private
    Ready : Boolean := False;
    -- NOTE - The priority of all Protected objects in this package
    --        must be the same!!!
  end Reader_Type;

  protected type Writer_Type(Buffer : access Buffer_Type;
                              Readers: access Reader_Array )  is
    procedure Write(  D : in    PC.Types_Pkg.Data_Type);
  private
  end Writer_Type;

end Buffer_Pkg;

------------------------------------------------------
package body Buffer_Pkg is
  protected body Buffer_Type is
    procedure Read(  D :    out Types_Pkg.Data_Type) is
    begin
      D.First  := Buf.First;
      D.Second := Buf.Second;
    end Read;
    procedure Write( D : in     Types_Pkg.Data_Type) is
    begin
      Buf.First  := D.First;
      Buf.Second := D.Second;
    end Write;
  end Buffer_Type;

  protected body Reader_Type is
    procedure Release is
    begin
      Ready := True;
    end Release;

    entry Read( D :    out Types_Pkg.Data_Type) when Ready is
    begin
      Buffer.Read(D);
      Ready := False;
    end Read;

  protected body Writer_Type is
    procedure Write( D : in     PC.Types_Pkg.Data_Type) is
    begin
      Buffer.Write(D);
      for i in Reader_Range loop
         Readers(I).Release;
      end loop;
    end Write;
  end Writer_Type;
end Buffer_Pkg;
```

Figure 4 Buffer and Suspension Protected Objects in Ravenscar

A requirement to avoid priority inversions means that a task in one protected object should be able to use the services of the other protected object without departing the first one(since a 2-step algorithm causes the caller to leave one protected object, drop it's priority to normal, then call the second protected object, letting other tasks possibly interleave). From a practical point of view, this means that the Writer_Type protected object needs to call both the Buffer.Write and then release each Reader_Type via the release procedure. The Reader_Type protected objects only need to know their Buffer so that they can call Buffer.Read. This is solved by defining all 3 protected types (Buffer_Type, Reader_Type and Writer_Type) in the same package and exchanging access values through discriminants for the configuration.

Now, there is no change to the producer task body or the consumer task body. We do need to alter the specification to pass in the new protected objects Writer_Type and Reader_Type.

This leaves a small amount of careful crafting of the actual objects, since we have 3 protected objects interlinked. We can do that using "access all buffer_type" and "access all reader_type". A package which accomplishes this is shown below.

```
package body Main_Pkg is
  -- the specification for this package is null
  -- except for a pragma Elaborate_body;

  Buff          : aliased Buffer_Pkg.Buffer_Type;

  -- next the 4 reader suspension objects, define and link to Buff
  Reader_One    : aliased Buffer_Pkg.Reader_Type( Buffer => Buff'Access);
  Reader_Two    : aliased Buffer_Pkg.Reader_Type( Buffer => Buff'Access);
  Reader_Three  : aliased Buffer_Pkg.Reader_Type( Buffer => Buff'Access);
  Reader_Four   : aliased Buffer_Pkg.Reader_Type( Buffer => Buff'Access);

  -- now combine all suspension objects in 1 array to pass to the buffer
  Consumer_Wait : aliased Buffer_Pkg.Reader_Array
                := (1 => Reader_One'Access,
                    2 => Reader_Two'Access,
                    3 => Reader_Three'Access,
                    4 => Reader_Four'Access);

  -- Now we create the tasks that will use these structures.
  -- the consumers, writer and producer task must be created in this order
  Consumer1 :  Consumer_Pkg.Consumer_Type(Buffer => Reader_One'Access);
  Consumer2 :  Consumer_Pkg.Consumer_Type(Buffer => Reader_Two'Access);
  Consumer3 :  Consumer_Pkg.Consumer_Type(Buffer => Reader_Three'Access);
  Consumer4 :  Consumer_Pkg.Consumer_Type(Buffer => Reader_Four'Access);
  Writer    : aliased Buffer_Pkg.Writer_Type(Buffer => Buff'Access,
                                             Readers=> Consumer_Wait'Access);
  Producer  : Producer_Pkg.Producer_Type(Buffer => Writer'Access);

end Main_Pkg;
```

Figure 5: An example package creating a buffer Protected Object

5 Analysis

The purpose of this discussion is to investigate the dynamic behaviour of such a system, but a reasonable question is if such a self-referential system can be built. The package in Figure five shows that indeed it can, albeit with a bit of work with aliased objects and access to objects.

Of interest to this analysis is the dynamic behaviour of this system. Now each reader delays and then calls it's dedicated suspension protected object, or can call the suspension protected object alone. It is not released from that suspension object until the producer writes new data. Once new data is written the producer leaves Buff, but still in its dedicated protected object releases each reader then leaves. Each consumer is released from its suspension object, calls Buffer.Read and consumes its data.

There is no risk of priority inversion in this example. All consumers are blocked on their own dedicated waiting object until explicitly released. Once released, however, they have the high priority of their protected object and do not release it to call the Buff protected object. Other solutions which permit the producer or consumer tasks to exit the Buffer protected object between calls permit priority inversion.

This solution is also free from deadlock and livelock. There can be no deadlock because there are no cycles in the calls to protected objects. There is no liveock because each consumer is blocked on its protected object until explicitly released, a release which occurs immediately after the producer writes new data.

The solution is not protected against overconsumption of cpu. If the total system load forces consumers to be late in arriving at their read buffer, they can proceed with a read, but the possibility exists that one could be so late as to miss a read completely and be released just before the producer writes new data. In such a situation load shedding is assumed and changes to the protocol required but we cannot assume a solution to such a problem here.

It will be noted that the Ravenscar-compliant examples given are considerably more complex in source code than the original version. Full Ada is a rich language with high levels of abstraction to support many concepts, such as selection and queuing in protected objects. This permits us to write our algorithms fairly compactly, but the cost is residual code for unused features and no control over others (such as selection order from multiple queues). This makes qualification of such code nearly impossible in high integrity systems.

When a simplified system such as Ravenscar is used, we must build our own library of routines in source code to implement the necessary paradigms. This adds some complexity to the source code because we must explicitly code algorithms which were once assumed. The point is, however, that we work to place such algorithms in modules, and we only include the ones necessary for the application. This additional complexity in the source code is acceptable for code qualification because one can reason at the source code level about exact behaviours; something that could not be done with the full runtime present.

It is also the belief of the author that real time designers can become comfortable with using such design patterns, and will create them as needed once they become used to the new paradigms.

6 Conclusion

This example has shown how one common real-time paradigm can be effectively implemented in Ravenscar using Protected objects. This design eliminates risk of priority and permits tight scheduling of the component tasks.

The cost is that scheduling decisions must be carefully crafted through collections of protected objects, but judicious arrangement will hide the implementation of such constructs from application code and result in clean, efficient multitasking designs.

In order for the Ravenscar Tasking Profile to be effectively used and accepted by the community for which it was intended, other common paradigms used by these communities should be analysed and design patterns developed that use the profile to advantage. It would be useful for IRTAW 11 to spend some time in this endeavour.

7 Bibliography

[Burns 1999] Burns J. “The Ravenscar Tasking Profile”., Ada Letters., December 1999.

[AONIX Current] Object-Ada and Raven descriptions, current documentation, Aonix Corp.

[IRTAW 1997] “Proceedings of the 8th International Real Time Ada Workshop”, Ada Letters, 1997.

[IRTAW 1999] “Proceedings of the 9th International Real Time Ada Workshop”, Ada Letters, 1999.

[IRTAW 2000] “Proceedings of the 10th International Real Time Ada Workshop”, Ada Letters, June 2000.

[IS 8652:1995] “The Ada Programming Language Reference Manual”, ISO/IEC 8652:1995

[LV 2000] “Iproving Predicatbility in Embedded Real Time Systems”, Lewis B, Vestal S. SEI Technical Report CMU/SEI-2000-SR-011

[Michell 2000] Michell, S. “Position Paper: Completing the Ravenscar Profile”, Proceedings of the 10th International Real Time Ada Workshop, Ada Letters, June 2000.