# Implementing a High-Integrity Executive using Ravenscar

Neil Audsley, Alan Burns and Andy Wellings

Real-Time Systems Research Group

Department of Computer Science, University of York, UK

## Abstract

*This paper discusses the use of the Ravenscar profile in the implementation of an APEX-compliant high-integrity executive. In general, the Ravenscar profile was found to have enough expressive power to implement a large proportion of the executive. However, the lack of dynamic priorities and single item queues were problematic.*

## 1 Introduction

In the domain of future civil aviation (as specified by the ARINC Integrated Modular Avionics (IMA) standards [1]), it is proposed that multi-process applications (written in a variety of languages) would be supported by a run-time executive. The interface to this executive (kernel) is called the APplication EXecutive (APEX) interface [2]. The goal of the work presented in this paper is to implement an APEX-compliant high-integrity kernel using a subset of the Ada 95 language. The subset used is a sequential subset (such as that typified by SPARK [3]) along with the Ravenscar [4] tasking subset.

## 2 APEX

The primary objective of the APEX specification is to define the interface between an IMA application and an underlying executive (operating system) controlling the computing resource. This interface is language-independent although currently there are only bindings to Ada (83) and C. Applications written according to this interface are notionally portable across different hardware resources and different executive implementations.

Central to IMA (and therefore APEX) is the concept of partitions that are allocated to core processor modules. A partition is assigned its own memory space and, consequently, cannot be accessed directly from other partitions. A partition is also allocated a specifiable fixed share of the underlying computing resources (in particular processor time). This share is guaranteed by the executive irrespective of the behaviour of other partitions on the core processor module. Hence, IMA and APEX provide spatial and temporal partitioning.

The main facilities specified by APEX can be divided into those that support the sharing of partitions within a core processor module (including inter-partition communication mechanisms) and those that support the sharing of processes within a partition. These will be called the Module Operating System (MOS) and the Partition Operating System (POS) respectively.

The objects required by a partition are specified at system build time. These objects include the number of processes contained in the partition, the number of intra-partition communication and synchronization mechanisms (semaphores, buffers, blackboards, events etc.) and the number of inter-partition communication mechanisms (sampling ports and queuing ports). The objects are created during the partition initialisation phase. No further resources can be created once a partition enters its normal mode of operation. Once a partition has been initialised, it is the responsibility of the MOS to schedule the partition (and to provide other time management functions) and to implement the inter-partition communication mechanisms. The goal of the POS is to support the execution of multiple processes running within a partition including intra-partition communication and process scheduling (and other time management functions). The POS also passes any inter-partition communication calls from the application to the MOS.

### 2.1 The Need to Subset APEX

To analyse a multi-process application so that its worst-case timing behaviour can be ascertained requires the use of a restricted computational model. These restrictions are also likely to be required by any certification activity as they result in a reduction in the dynamic behaviour of the application. To be able to predict that all timing requirements will be satisfied, requires an analysable scheduling protocol which will dictate how resources are used at runtime and will support analysis techniques that allow worst-case behaviour to be predicted.

One effective scheduling protocol is fixed priority preemptive dispatching with ceiling priorities used on all

shared resources. APEX attempts to support this method of scheduling but it does not directly allow ceiling priorities to be defined (and hence applications can suffer excessive priority inversion). An effective analysis approach for fixed priority scheduling is to calculate the worst-case response time of each process and then to compare this with the process's deadline [4].

The restricted computational model necessary to give predictable behaviour usually sees an application as consisting of a fixed set of processes. The following restrictions apply:

- Each process has a single invocation (release) event.
- Periodic processes have a time trigger (i.e. the invocation event comes from a clock).
- Sporadic processes have an invocation event that either comes from the environment (e.g. as an interrupt), or from another application process within the same partition, or from another distinct application partition.
- Processes should minimise any other blocking activity during their execution.

In general, most of APEX can be supported by a simple kernel and most services can be analysed. However, the following major restrictions are necessary to facilitate analysis and implementation:

- No use of dynamic priorities by the application — to simplify the schedulability analysis.
- No suspending another process — to simplify the implementation and certification.
- No stopping another process — to simplify the implementation and certification.
- No use of Semaphores — as it requires dynamic priorities to implement the ceiling protocols.
- Single process only waiting on a Queue, Buffer or Event — to simplify the implementation and certification.

Periodic processes are directly supported by APEX. They have well-defined periods and a means of recognising when a deadline has been missed. Sporadic tasks are less well provided for. Even with the restrictions outlined above there are still a number of candidates for providing the invocation event for sporadic processes:

- Use of `resume` after the process has executed a `suspend_self`.
- Use of a queuing port — the arrival of data freeing a blocked reader.
- Use of a buffer — the arrival of data freeing a blocked reader.
- Use of an event —the signal of the event freeing a waiting process

Although hard sporadic processes can have deadlines, there is no means (in APEX) of tying the invocation event to the release of the process. Nor can a deadline be inhibited by a sporadic while it is awaiting its invocation event. There is no easy way of circumventing these problems. One possibility discussed, we understand, by members of the APEX definition team is to extend the definition of the four potentially suspending operations listed above. An extra 'replenish' boolean flag is added to each interface call. If the flag is set to 'false' (which would be the default in the Ada definition) then no action is taken in respect of this flag. However if it set to 'true' then the existing deadline alarm is cancelled and a new one is set **only** when the process is released for execution. Therefore, while the process is blocked it does not have a deadline. When the release operation is executed it is atomic with a replenish action on the sporadic process. Deadline expiry can then be treated according to whether the process is hard or soft.

APEX allows a (priority or FIFO) queuing discipline to be associated with a particular object (event, buffer, and queue) to be set dynamically when the object is created. This is at odds with Ada which requires it to be set on a per partition level. However, as Ravenscar only allows one task to be queued on an object there is no queuing discipline as such. Hence, it is necessary either to re-implement queues in the POS or to place a restriction on APEX that only a single process can wait on a particular object. This latter approach is the one preferred as it eases the implementation of the POS.

# 3 Implementation of an APEX-Compliant Kernel

This section considers how a kernel can be implemented that supports the above subset and facilitates certification.

## *3.1 Overview of Approach*

Prime requirements of the kernel are that:

- it must support both Ada 95 and C applications;
- it must facilitate certification;
- it should support temporal and spatial firewalls between APEX partitions.

One of the main issues concerns the role of Ada tasking at the application level (when the application is written in Ada) and within the executive. The first requirement above dictates that there must not be a strong link between Ada tasks at the application and the APEX notion of process. If there were, all C applications would need Ada wrappers in order to access APEX processes. This would be contrary to allowing portability between different implementations of APEX. In addition, it imposes greater restrictions on support tools (e.g. compilers, linkers), as they must be compatible.

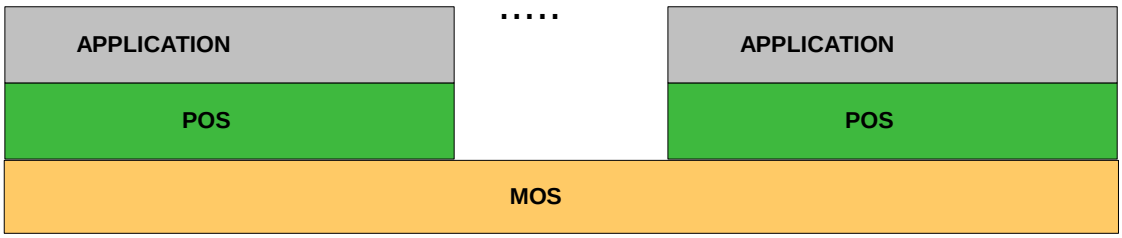



**Figure 3.1: Kernel Architecture**

Consequently, any use of Ada tasking must either be disallowed or be totally transparent to the kernel. In the latter case, either a multitask Ada program runs as a single APEX process (in which case the run-time support system is transparent to APEX), or the Ada run-time support system must be implemented via calls to the APEX executive. In the remainder of this paper, it is assumed that the application is written in purely sequential Ada 95.

The outline architecture of the kernel is given in Figure 3.1. In general, the POS is responsible for process management within a partition and the MOS is responsible for partition management. The POS provides an APEX API to the application. The implementation of the POS interfaces to the MOS for access to low-level (i.e. hardware) devices. The MOS should not be aware of the internal process structure of an application. Furthermore, the spatial and temporal firewall requirements between partitions suggests that either:

- each application runs in a separate address space with the MOS and POS sharing a further distinct address space; or
- each application runs in a separate address space with a copy of the POS, and the MOS occupies a further distinct address space.

Given that the processes within a partition all run at the same criticality and that they share memory, all process management support and IPC support should be as efficient as possible. This would imply that the second approach above is the most appropriate.

## 3.2 Certification Issues

The architecture above helps from a certification perspective, as it minimises the part of the kernel responsible for partitioning and resource control, i.e. the MOS, from the more complex POS, including the APEX implementation. To facilitate certification, the kernel must be implemented in a subset of Ada that is accepted by certification authorities. In the past that has been a sequential subset called SPARK. This should be entirely appropriate for the implementation of the MOS (noting some processor specific assembler may be required).

Whilst the POS could be implemented in SPARK, we note that Ravenscar has been considered acceptable for use within safety-critical systems. Consequently, although Ada tasking may be forbidden at the application level, it can be used within the POS.

Summarising, a mapping between APEX processes and Ada tasks provides a route by which both APEX applications and the kernel can be certified. Given the existence of "safe" run-time support systems for Ada (e.g. Aonixs' Raven), use of Ada tasking within the executive should be encouraged.

## *3.3 APEX Implementation in Ada95*

The above discussion suggests a software architecture for an APEX-compliant executive implemented in Ada 95 as illustrated in Figure 3.2. The figure has the same overall architecture as Figure 3.1. All calls to APEX services are procedure calls into the POS. Each APEX process is mapped to an Ada task by the APEX `Create_Process` service request (i.e. the Ada task is actually resident in the POS, not within the application).

To illustrate the structure given in Figure 3.2, consider the implementation of the process management support. The goal here is to map APEX processes onto Ada task in such a way that they can be used from a C or an Ada application. Furthermore, this is to be done in such a way that no dynamic creation of tasks at run-time occurs. Given that currently APEX requires a fixed number of processes per partition, this can be achieved by declaring an array of tasks and matching an element of the array to each process when it is created. There is one aspect of APEX that makes this approach problematic. APEX when it creates a process passes the process' attributes as a parameter. These attributes include the base priority of the process. Unfortunately, all the tasks in the array will be created at system start-up and the base priority of a task is associated with a pragma attached to the task specification. Thus to change this priority would require Ada's dynamic priority facility. This is not part of the Ravenscar profile.

Given the static nature of each partition, the fact that APEX requires all resources to be pre-allocated and all process names to be known by the executive, it seems strange that most of the process attributes are not known by the executive. If this were the case then it would be possible to initialise each element of the task array with the correct base priority. Further support for this approach is also given by the argument that each process should not be able to manipulate its own priority as this may undermine the schedulability analysis which has been performed to guarantee process deadlines.

The approach adopted here is that the application processes' attributes are checked against the executive's view and if they differ, a configuration error is flagged.

# 4 APEX and the Ravenscar Ada 95 Run-time

This section addresses the issues of the implementation of APEX using a Ravenscar run-time as a basis for the POS. It is necessary to understand issues that are raised by this strategy early so that the actual design and implementation: is consistent with the Ravenscar subset; is consistent with the APEX restrictions proposed; is amenable to timing analysis; and forms a reasonable basis for potential certification.

## *4.1 Overview of Implementation*

In Figure 4.1 the overall software structure is given. This is essentially identical to that in Figure 3.1 except that further decomposition of the POS is shown, together with decomposition of the MOS. The essential elements of the architecture are:

(1) Application (sequential) Ada 95 or C (noting that the concurrency is provided by APEX);

(2) APEX layer, implemented in Ada 95 (Ravenscar subset);

(3) Ravenscar run-time;

(4) APEX Support Functionality (ASF) which provides additional facilities for the APEX layer, over and above those in Ravenscar;

(5) Board Support Package (BSP), providing key hardware dependent services for both Ravenscar and the APEX layer. This is subdivided into:

- BSP(Ada) – the Ada specific part of the BSP that configures the Ravenscar run-time for a particular application (e.g. task-control blocks);
- BSP(HW) – the part of the BSP that provides low-level interfaces to the hardware (i.e. device drivers).

Note that there is a single MOS (per processor) which provides partitioning for each application / POS instance. The MOS is executed in Supervisor mode (we are using a PowerPC), with partitions in User mode, each allocated a separate virtual memory space (i.e. separate MMU entry) – noting that the MOS will have the whole of physical memory available to it, if required. Within each partition, Ravenscar provides task context switches (without using the MMU) by switching task control blocks.

It is intended to use as standard a Ravenscar implementation as much as possible (minimal changes are discussed later). However, it is noted that typically the BSP (as supplied with a standard Ravenscar implementation) will need considerable changes to split into two parts (BSP(Ada) and BSP(HW)) – noting that a context switch facility will be required in *both* parts of the BSP, one to switch between Ada tasks, the other to switch between partition contexts. Also, changes need to be made to support APEX and the execution of the Ravenscar run-time within a partition. The issue of timing support is discussed further below.

## 4.2 POS Timing Facilities

One major area of concern in the POS is in the area of provision of timing facilities. The APEX part of the POS requires timing support in the following areas:

(1) for periodic release of processes;

(2) for (hard) deadlines of processes;

(3) for timeouts on various services.

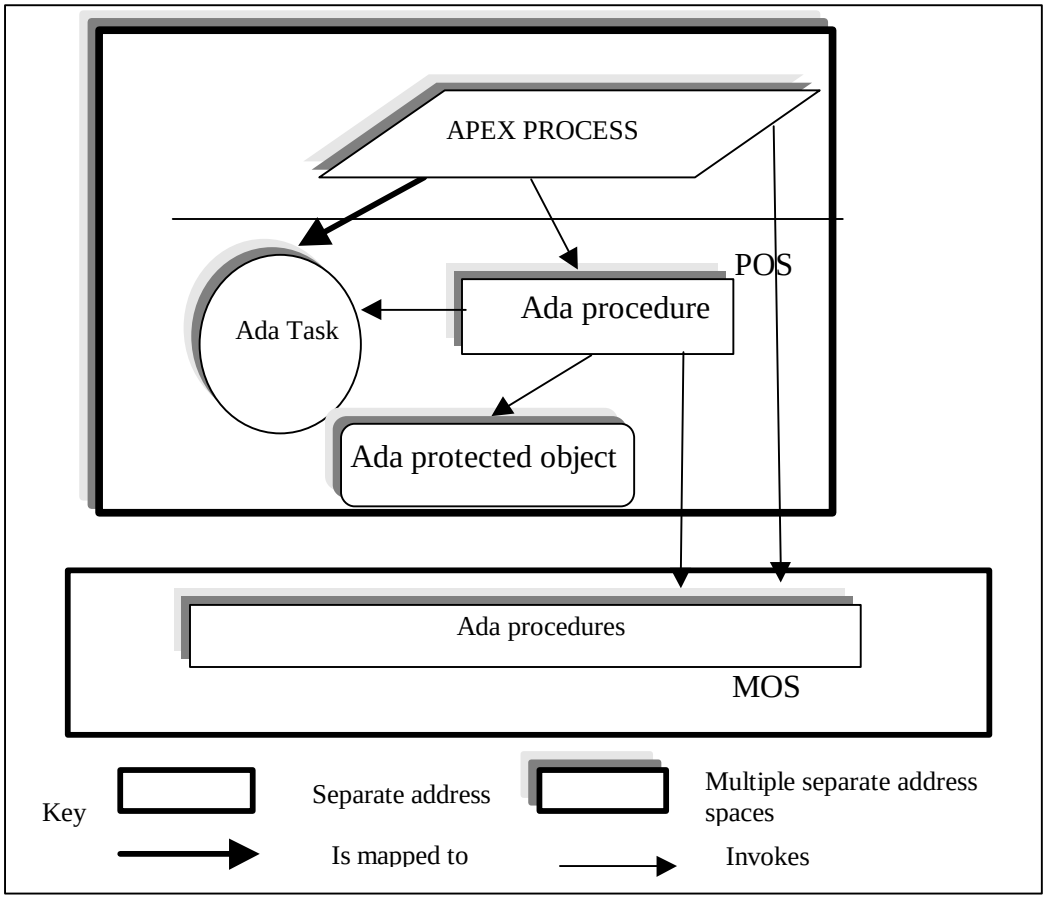
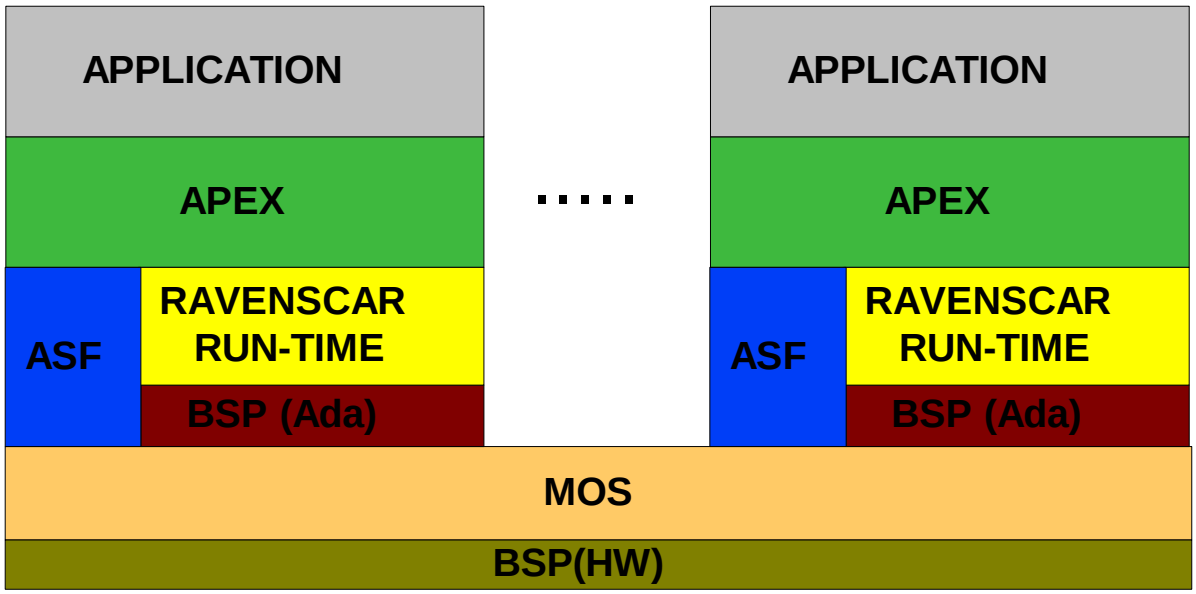



**Figure 3.2: APEX Software Architecture**

**Figure 4.1: Architecture Overview**



When implementing upon a full Ada 95 run-time, a natural model is to utilise the "delay until" statement to implement (1). Unfortunately, given that Ravenscar is running in a virtual machine, this timing call is unavailable as any timing must be done by the MOS.

Time facility (2) can be achieved using a "select then abort" approach and facility (3) by a timed entry call. However, these are not available within Ravenscar, which does not support the Ada "select" statement.

One solution may be to fabricate deadline detection and timeouts using shadow tasks (for each task, we have extra task for detecting its deadline and any timeouts that it may set). However, this is heavy with overheads. Hence an alternative solution is used , utilising an ASF (i.e. not Ravenscar) based timing facility .

The approach is to provide timing facilities within the ASF to support APEX timing requirements. This service implements all timing facilities within a partition, noting that the actual hardware timers are controlled by the MOS. It is envisaged that the MOS communicates timing events to the ASF rather than clock ticks, where the timing events are set by the APEX / ASF (there should be no need for the Ravenscar run-time to set timing events directly).

The obvious implementation for the ASF timer facilities is via a protected object containing a timer queue, with appropriate procedures for setting period delays, setting and canceling both deadlines and timeouts. These request timing events from the MOS, always informing the MOS of the earliest (i.e. closest) time at which it wants an event. The MOS invokes a "timer task" when the event occurs which calls into the appropriate protected object.

With this approach, however, all objects associated with timeouts will have ceiling priorities set at the highest priority, equivalent to that of the timing task, given that this task needs access to all those objects (i.e. protected objects). This problem may ease if the creation of the object declares whether it allows a timeout or not. Thus, two forms of the protected object type would exist - one with and one without timeouts for each particular object. Clearly, this has benefits from a timing analysis point of view, but complicates the design. Perhaps the benefits of having timeouts within the application need to be carefully considered, since if they were not required, then the problem would be eradicated.

# 5 Conclusions

Traditionally kernels for high-integrity systems support single sequential applications. The maturity of timing analysis techniques has initiated a move to restricted concurrent applications. The Ravenscar profile is the first Ada concurrency subset to achieve widespread support. This paper has shown that the Ravenscar profile can be used to implement a subset of the APEX executive. This offers a route for certifying such a kernel and the ability to invoke the kernel from C applications.

Hosting APEX on top of a Ravenscar implementation requires the BSP to be modified so that it operates in a virtual machine and does not handle interrupts directly. Supporting APEX timeouts is cumbersome because the Ravenscar subset of Ada does not support them and only a single task can be queued on a protected entry. Hence, the POS has to be augmented with extra facilities to provide timing and queuing support. These facilities can be implemented in a Ravenscar-compliant manner.

The other significant restriction of Ravenscar is the lack of dynamic priorities. This makes it difficult to support fully the APEX process creation interface and makes it impossible to assign ceiling priorities to APEX entities such as semaphores. Given the other restrictions of Ravenscar, it may be worth considering if the addition of dynamic priorities could be accomplished with little extra overhead.

# Acknowledgements

This work has been undertaken as part of the SHIMA project supported by the UK Department of Trade and Industry. We would like to thank the project's other partners Stewart Hughes, Smiths Industries and Bell Helicopter Textron for their contribution to the ideas expressed in this paper. In particular, Steve Ellis and Kevin Walker from Smiths Industries Aerospace have had a significant impact on the work presented here. We would also like to thank Terry Froggatt for his comments.

# References

[1] ARINC AEE Committee, *Design Guidelines for Integrated Modular Avionics,* 1991

[2] ARINC AEE Committee, *Avionics Application Software Standard*, 1997

[3] Barnes, J.G.P, *High Integrity Ada: The SPARK Approach*, Addison Wesley, 1997

[4] Baker T and Vardanega T, *Session Summary: Tasking Profiles*, Proceedings of IRTAW8, Ada Letters, XVII(5), pp 5-7, 1997

[5] Audsley N.C et al, *Applying New Scheduling Theory to Static Priority Pre-emptive Sceduling, Software Engineering Journal, 8(5), pp 284-292, 1993*