UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

INTEGRATION OF GENERIC OPERATING SYSTEMS IN PARTITIONED ARCHITECTURES

João Pedro Gonçalves Crespo Craveiro

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2009

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

INTEGRATION OF GENERIC OPERATING SYSTEMS IN PARTITIONED ARCHITECTURES

João Pedro Gonçalves Crespo Craveiro

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor José Manuel de Sousa de Matos Rufino

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2009

This work was partially funded by:

ESA/ITI - European Space Agency Innovation Triangular Initiative
(through ESTEC Contract 21217/07/NL/CB — Project AIR-II)

FCT - Fundação para a Ciência e Tecnologia
(through the Multiannual Funding Programme)

Acknowledgments

During the course of the work which led to the present dissertation, including the curricular part of the Masters programme, there are many people to which I am most thankful for all the different kinds of support they gave me. May my gratitude towards them remain forever patent in these lines.

First, I would like to thank my supervisor, Prof. José Rufino, for his orientation, support, inspiration, dedication, and for always striving to give his students the opportunities and conditions to work.

A word of appreciation goes also to the Department of Informatics and the LaSIGE, which have always been superb places to study and work.

Due thanks also to Prof. Carlos Almeida from IST–UTL, Rui Covelo and Pedro Venda, for the collaboration in the paper published in the proceedings of AICCSA 2009. The collaboration and comments of the people involved in the AIR activities were also very insightful in the various stages of this work. They are: José Neves, Edgar Pascoal, Sérgio Santos, Tobias Schoofs, and Cássia Tatibana (from Skysoft Portugal), and James Windsor (from ESA/ESTEC). I would also like to thank Dr. Joel Sherrill at OAR Corporation and Manuel Coutinho at EDISOFT, for the helpful e-mail exchanges regarding RTEMS support.

My coleagues and friends at FCUL and/or LaSIGE, some of which I have had the pleasure to work with, have always been a source of different degrees of support, inspiration, and interesting discussions during meals and breaks. In no special order, and not exhaustively, I would like to thank Diogo Albuquerque, Romeu Carvalho, Maria João Leal, Ricardo Mascarenhas, Tiago Reis, and Joaquim Rosa, for how pleasant it has been to connect and/or work with them.

Last, but not least (this line never gets old, does it?), the biggest "Thank you" of the world goes for my family. My mother Mafalda, my sister Joana, my godmother Fátima and my girlfriend Catarina have always been there for me and helped me keep seeing the light at the end of the tunnel(s). It is not too much to say that I couldn't do this without them.

Ao meu pai.

Abstract

The Integrated Modular Avionics (IMA) specification defines a partitioned environment hosting multiple avionics functions of different criticalities on a shared computing platform. ARINC 653, one of the specifications related to the IMA concept, defines a standard interface between the software applications and the underlying operating system. Both these specifications come from the world of civil aviation, but they are getting interest from space industry partners, who have identified common requirements to those of aeronautic applications.

Within the scope of this interest, the AIR architecture was defined, under a contract from the European Space Agency (ESA). AIR provides temporal and spatial segregation, and foresees the use of different operating systems in each partition. Temporal segregation is achieved through the fixed cyclic scheduling of computing resources to partitions.

The present work extends the foreseen partition operating system (POS) heterogeneity to generic non-real-time operating systems. This was motivated by documented difficulties in porting applications to RTOSs, and by the notion that proper integration of a non-real-time POS will not compromise the timeliness of critical real-time functions. For this purpose, Linux is used as a case study. An embedded variant of Linux is built and evaluated regarding its adequacy as a POS in the AIR architecture. To guarantee safe integration, a solution based on the Linux paravirtualization interface, paravirt-ops, is proposed.

In the course of these activities, the AIR architecture definition was also subject to improvements. The most significant one, motivated by the intended increased POS heterogeneity, was the introduction of a new component, the AIR Partition OS Adaptation Layer (PAL). The AIR PAL provides greater POS-independence to the major components of the AIR architecture, easing their independent certification efforts. Other improvements provide enhanced timeliness mechanisms, such as mode-based schedules and process deadline violation monitoring.

Keywords: Aerospace applications, ARINC 653, IMA, Linux, operating systems, real-time.

Resumo

A especificação Integrated Modular Avionics (IMA) define um ambiente compartimentado com funções de aviónica de diferentes criticalidades a coexistir numa plataforma computacional. A especificação relacionada ARINC 653 define uma interface padrão entre as aplicações e o sistema operativo subjacente. Ambas as especificações provêm do mundo da aviónica, mas estão a ganhar o interesse de parceiros da indústria espacial, que identificaram requisitos em comum entre as aplicações aeronáuticas e espaciais.

No âmbito deste interesse, foi definida a arquitectura AIR, sob contrato da Agência Espacial Europeia (ESA). Esta arquitectura fornece segregação temporal e espacial, e prevê o uso de diferentes sistemas operativos em cada partição. A segregação temporal é obtida através do escalonamento fixo e cíclico dos recursos às partições.

Este trabalho estende a heterogeneidade prevista entre os sistemas operativos das partições (POS). Tal foi motivado pelas dificuldades documentadas em portar aplicações para sistemas operativos de tempo-real, e pela noção de que a integração apropriada de um POS não-tempo-real não comprometerá a pontualidade das funções críticas de tempo-real. Para este efeito, o Linux foi utilizado como caso de estudo. Uma variante embedida de Linux é construída e avaliada quanto à sua adequação como POS na arquitectura AIR. Para garantir uma integração segura, é proposta uma solução baseada na interface de paravirtualização do Linux, paravirt-ops.

No decurso destas actividades, foram também feitas melhorias à definição da arquitectura AIR. O mais significante, motivado pelo pretendido aumento da heterogeneidade entre POSs, foi a introdução de um novo componente, AIR Partition OS Adaptation Layer (PAL). Este componente proporciona aos principais componentes da arquitectura AIR maior independência face ao POS, facilitando os esforços para a sua certificação independente. Outros melhoramentos fornecem mecanismos avançados de pontualidade, como *mode-based schedules* e monitorização de incumprimento de metas temporais de processos.

Palavras-chave: Aplicações aeroespaciais, ARINC 653, IMA, Linux, sistemas operativos, tempo-real.

Resumo alargado [1]

Os sistemas para aplicações espaciais do futuro requerem arquitecturas computacionais inovadoras, que permitam a reutilização de componentes entre diferentes missões espaciais. Neste contexto, a especificação Integrated Modular Avionics (IMA), originalmente definida para aplicações aeronáuticas, surgiu como um desafio às arquitecturas federadas, nas quais cada função de aviónica teria os seus recursos de processamento dedicados (e frequentemente separados fisicamente). As arquitecturas IMA, com funções aviónicas com diferentes graus de criticalidade a coexistir num ambiente compartimentado (separadas em unidades lógicas de contenção denominadas partições), permitem optimizar a utilização e realocação de recursos.

A especificação ARINC 653 é um bloco fundamental da filosofia IMA. Esta especificação observa os conceitos de segregação temporal e espacial, e define uma interface genérica de serviço, denominada APEX (Application Executive), entre o software aplicacional e o sistema operativo da plataforma computacional. A segregação temporal concerne a garantia de que as actividades de uma partição não comprometem a pontualidade (ou seja, o cumprimento de metas temporais) das funções a serem executadas em outras partições. Neste contexto, a segregação temporal é atingida através do escalonamento fixo, cíclico das partições. Por seu lado, a segregação espacial permite impedir que uma aplicação a executar numa partição aceda a zonas de memória pertencentes a outras partições.

Após ter identificado bastantes pontos em comum em termos de requisitos com a indústria aviónica, a indústria espacial — e a Agência Espacial Europeia (ESA) em particular — manifestou o seu interesse na adopção dos conceitos das especificações IMA e ARINC 653 para os sistemas computacionais a bordo das suas missões.

No seguimento deste interesse, foi desenvolvida a arquitectura AIR — ARINC 653 In Space RTOS. Inicialmente uma prova de conceito da adaptação do sistema operativo de tempo-real RTEMS para os requisitos da especificação ARINC 653, a contribuição dos projectos AIR traduz-se numa arquitectura que fornece a reque-

[1]Em cumprimento do disposto no Artigo 27.º, n.º 3, da Deliberação n.º 1506/2006 (*Regulamento de Estudos Pós-Graduados da Universidade de Lisboa*), de 30 de Outubro

rida segregação temporal e espacial, prevendo a utilização de diferentes sistemas operativos entre as partições.

Na tecnologia AIR, o componente transversal a todas as partições e responsável pelo aprovisionamento das principais propriedades do sistema denomina-se AIR PMK (Partition Management Kernel). O AIR PMK assegura a inicialização do sistema, o escalonamento e despacho das partições, o suporte à comunicação entre partições, a gestão de uma noção de tempo comum a todas as partições, e a abstracção face à plataforma de hardware; este componente é, assim, central para as garantias de segregação temporal e espacial. As interacções das aplicações das partições com o sistema são efectuadas através da interface APEX definida na especificação ARINC 653. Na arquitectura AIR, esta interface é concretizada na forma de um componente que visa ser flexível e portável em função dos sistemas operativos das partições. A tecnologia AIR engloba ainda um componente de supervisão, responsável por tomar acções de tratamento de erros aos diferentes níveis da arquitectura. Tanto quanto possível, as acções do Health Monitoring tentam restringir a propagação do erro ao seu domínio de ocorrência, distinguido entre erros ao nível do processo, da partição e do sistema.

A motivação para o trabalho descrito na presente dissertação resulta de várias observações. A primeira é a de que os sistemas aos quais a arquitectura AIR se destina podem beneficiar de aplicações disponíveis em sistemas operativos genéricos, e que não se encontram nos sistemas operativos de tempo-real tradicionais (p. ex., suporte a linguagens interpretadas). Portar estas aplicações para um dos sistemas operativos de tempo-real que se esteja a utilizar numa partição é um processo que pode ser moroso, e que é definitivamente propenso a erros, conforme corrobora a literatura científica. Assim, existe benefício em estender a heterogeneidade entre os sistemas operativos das partições, já prevista pela arquitectura AIR, ao domínio dos sistemas operativos genéricos, mesmo que estes não observem requisitos de tempo-real. Dado que a segregação temporal (e, consequentemente, a pontualidade das tarefas com requisitos estritos de tempo-real) é assegurada pelo escalonamento fixo e cíclico da capacidade de processamento entre as partições, a devida integração de um sistema operativo não-tempo-real numa partição (para execução de tarefas com nível de criticalidade baixo ou nulo) não afectará a pontualidade das tarefas de tempo-real. Dado que o objectivo das partições com sistemas operativos genéricos será executar aplicações existentes sem necessidade de as portar para uma nova interface, só lhes será fornecido um subconjunto de serviços da interface APEX suficiente para suportar actividades de gestão e comunicação entre partições. O primeiro e principal caso de estudo da integração de sistemas operativos genéricos englobará os sistemas baseados no núcleo de código livre/aberto Linux.

Nesta aproximação, as partições com sistemas operativos genéricos receberão, tal como as restantes (que contêm sistemas operativos de tempo-real), uma janela temporal fixa e garantida no escalonamento das partições. Este comportamento distingue-se de anteriores aproximações à coexistência de processos Linux com tarefas de tempo-real (como o RTLinux, o RTAI, ou o xLuna), em que os processos Linux apenas são escalonados quando não exista nenhuma tarefa de tempo-real elegível para execução.

Durante o decurso do trabalho desta dissertação, foram desenvolvidos melhoramentos à arquitectura AIR, em parte directamente relacionados com a heterogeneidade entre os sistemas operativos nas partições (abreviadamente POS, de *Partition Operating System*). Na fase preliminar deste trabalho, foi identificada a necessidade de um componente de adaptação para a partição que iria conter um sistema operativo genérico; este componente iria situar-se na arquitectura abaixo do núcleo do sistema operativo, entre este e o AIR PMK. Uma investigação mais profunda revelou contudo que este componente teria uma função mais abrangente, sendo também benéfico para as partições com sistemas operativos de tempo-real. A consolidação destas observações resultou na integração deste componente, o AIR PAL (POS Adaptation Layer), como uma significativa contribuição para a tecnologia AIR. Em cada partição, uma instância apropriada do AIR PAL encapsula as especificidades do respectivo sistema operativo; tal permite que outros componentes da arquitectura (com ênfase no AIR PMK) sejam mais independentes do sistema operativo de cada partição e menos propensos a modificações, beneficiando assim possíveis processos de certificação dos componentes da arquitectura. O AIR PAL possibilita, através da separação de responsabilidades, a optimização do processo de desenvolvimento de sistemas — potenciando a reutilização de componentes de software entre diferentes misssões, sem necessidade de repetir o processo da sua certificação na totalidade.

Outras contribuições para a definição da arquitectura AIR concernem a melhoria das características de pontualidade. A primeira é a funcionalidade de *mode-based schedules*, um serviço opcional da especificação ARINC 653 que permite a definição estática de múltiplas tabelas de escalonamento das partições; as partições devidamente autorizadas podem alternar entre estas escalas, quer para adaptar o funcionamento do sistema a diferentes etapas da missão (p. ex., em vôo, aproximação ao solo, ou exploração), quer para concretizar mecanismos de tolerância a faltas. O outro mecanismo introduzido relacionado com a pontualidade do sistema consiste na supervisão de violações de prazos de execução por parte dos processos; este mecanismo foi integrado na sequência da rotina de interrupção de relógio, de uma forma que estabelece eficientemente um compromisso entre a prontidão exigida no contexto de uma rotina de interrupção

e o atraso máximo na detecção destas falhas temporais (na pior hipótese, uma violação de um prazo de execução é detectado — e reportado aos serviços de Health Monitoring do sistema — no início da próxima janela temporal de execução da respectiva partição). A instância do componente AIR PAL em cada partição fornece o mecanismo e as estruturas de dados para verificação do cumprimento das metas temporais pelos respectivos processos.

Dada a coexistência com outros sistemas operativos na mesma plataforma, faz total sentido, no desenvolvimento de uma partição Linux, aplicar técnicas adequadas a sistemas com escassez de recursos — os sistemas embebidos. Esta dissertação descreve o processo de construção de uma variante embebida de Linux, atestando a sua adequação para integração na arquitectura AIR. São ainda analisados os requisitos de segurança na operação (*safety*) que esta integração deve observar, e é proposta a adaptação das interfaces de paravirtualização (disponíveis nas mais recentes versões do núcleo Linux) para cumprimento desses requisitos e adaptação às plataformas de processamento tipicamente empregues em missões espaciais (baseadas em processadores SPARC LEON). A arquitectura AIR é demonstrada através de um protótipo baseado em RTEMS, com ênfase nas melhorias arquitecturais introduzidas no decorrer deste trabalho; a concretização de um protótipo com o núcleo Linux numa das partições requer trabalho de engenharia que se estimou não ser compatível com a duração do projecto desta dissertação, sendo motivo para trabalho futuro.

Futuros desenvolvimentos incluem ainda a extensão do conceito de integração em arquitecturas compartimentadas a outros sistemas operativos, como o Windows (na forma do Windows Research Kernel), a provisão de ferramentas para apoio à integração de sistemas baseados na arquitectura AIR (combinando a análise do impacto mútuo entre os dois níveis de escalonamento com a geração automatizada de tabelas de escalonamento), e a definição de extensões à arquitectura para beneficiar do paralelismo fornecido por plataformas com processadores *multicore*.

Contents

List of Figures

Chapter 1

Introduction

Space systems of the future demand for innovative computer architectures, enabling component reuse among the different space missions. In this context, the Integrated Modular Avionics (IMA) specification [1], originally defined for aeronautic applications, appeared as a challenge to the federated architectures, in which each avionics function would have its dedicated (and often physically separated) hardware resources. IMA architectures, with avionics functions of different criticality levels coexisting in a partitioned environment, allow for a more optimal resource utilization and reallocation [2].

The ARINC 653 specification [3, 4] is a fundamental block of IMA, observing the concepts of time and space partitioning, and defining a general-purpose service interface, known as APEX (Application Executive), between the application software and the operating system (OS) of on-board computer platforms.

The interest of the European Space Agency (ESA) in the adoption of these concepts for aerospace applications [5, 6] has led to the definition of the AIR (ARINC 653 in Space RTOS) architecture, which preserves the hardware and real-time operating system (RTOS) independence defined within the scope of ARINC 653, while foreseeing the use of different RTOSs through the partitions [7, 8].

Motivated by the documented difficulties in porting some general-purpose applications to RTOS [9, 10], this thesis describes the study of the process of integrating non-real-time generic operating systems, like (embedded) Linux, as partition operating systems in the AIR architecture [11]. In AIR, these non-critical subsystems also receive a guaranteed time window in the fixed cyclic schedule; this differs from previous approaches to the coexistence of critical and non-critical processes, like RTLinux [12], RTAI [13], and xLuna [14].

The study of the integration of generic operating systems in the AIR architecture also spawned improvements to the latter's definition[15]. A new component, the AIR Partition OS Adaptation Layer (PAL), was introduced, implementing a homogenous methodology for subsequent addition of support to new partition operating systems, either RTOSs or generic non-real-time operating systems. Other enhancements made to the AIR architecture during the course of this thesis's work add robustness to timeliness adaptation and supervision, with features such as mode-based schedules and process deadline violation monitoring.

1.1 Motivation

The present work was motivated by the following observations:

- the systems for which the AIR architecture is tailored may well benefit from applications available on generic operating systems, and not on off-the-shelf RTOSs;
- since temporal segregation (and, consequently, the timeliness of hard real-time tasks) is secured by the fixed, cyclic time slicing of computing resources among partitions, proper integration of a non-real-time operating system inside a partition (for non-critical tasks) should not affect the timeliness of hard real-time tasks;[1]
- both real-time and non-real-time operating systems require architectural support to work as partition operating systems; isolating such support in a smaller, dedicated component allows for a more homogenous process of adding support to new operating systems, while maintaining the verification, validation and/or certification status of the remaining architectural components.

1.2 Contributions

The main contributions of the work described in this thesis comprise:

(i) an improved definition of the AIR Technology, which provides an architecture for time/space partitioned aerospace applications, enrichened by:

   (a) flexible integration of both real-time and non-real-time partition operating systems, with minimal impact on the verification, validation and certification processes of the architectural components;

   (b) enhanced timeliness adaptation and supervision mechanisms, such as mode-based schedules and process deadline violation monitoring;

[1]Partition scheduling is described in detail in Section 3.1.1.

(ii) an embedded Linux solution viable for integration as a partition operating system in the AIR architecture;

(iii) a proposed methodology for safe integration of Linux in the AIR architecture, without violating the architecture's strong segregation properties.

1.3 Institutional context

The present work took place at the Large-Scale Informatics Systems Laboratory (LaSIGE–FCUL), a research unit of the Informatics Department (DI) of the University of Lisbon, Faculty of Sciences. It was developed within the scope of the AIR-II (ARINC 653 In Space RTOS — Industrial Initiative) project, which fits in the Timeliness and Adaptation in Dependable Systems research line of the Navigators group.

AIR-II consists of a consortium, sponsored by the European Space Agency (ESA), comprising LaSIGE–FCUL, Skysoft Portugal, and Thales Alenia Space (a France-based service and system provider). As a full-time junior researcher, the author of this thesis was a member of the LaSIGE–FCUL AIR-II team, participating in the whole extent of the team's activities: identification of problems, proposal and discussion of solutions, experimental work, project meetings (both internal and involving the remaining partners), production of project deliverables, and dissemination of results (through the publication and presentation of scientific papers at international conferences).

1.4 Publications

With the goal of validating and disseminating the present ideas, the work of this thesis generated the following refereed publications:

1. **J. Craveiro**, J. Rufino, C. Almeida, R. Covelo, and P. Venda, "Embedded Linux in a partitioned architecture for aerospace applications," in *Proceedings of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2009)*, Rabat, Morocco, May 2009, pp. 132–138.[2] [11]

2. J. Rufino, **J. Craveiro**, T. Schoofs, C. Tatibana, and J. Windsor, "AIR Technology: a step towards ARINC 653 in space," in *Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference*. Istanbul, Turkey: EUROSPACE, May 2009. [15]

[2]Regular paper acceptance ratio: 32%

3. **J. Craveiro**, J. Rufino, T. Schoofs, and J. Windsor, "Flexible operating system integration in partitioned aerospace systems," in *Actas do INForum 2009, Simpósio de Informática*, Lisbon, Portugal, Sep. 2009, accepted for publication. [16]

and also the following reports:

4. J. Rufino and **J. Craveiro**, "AIR Design Consolidation, PMK – Partition Management Kernel," FCUL, AIR-II Deliverable WP 1.1, 2009, confidential document. [17]

5. **J. Craveiro**, J. Rufino, T. Schoofs, and J. Windsor, "Robustness, flexibility and separation of concerns in ARINC 653-based aerospace systems," AIR-II Technical Report RT-09-02, 2009. [18]

1.5 Document outline

The remainder of this thesis is structured as follows:

Chapter 2 Literature review of basic underlying concepts (real-time, operating systems, IMA, ARINC 653) and related work.

Chapter 3 Description of the AIR architecture upon which the present work intends to integrate generic operating systems.

Chapter 4 Developments made to the AIR architecture during the course of the present work.

Chapter 5 Identification of the problem of integrating generic operating systems in the AIR architecture, and respective solutions based on (embedded) Linux as a case of study.

Chapter 6 Concluding remarks and insight for future developments.

Chapter 2

Related work

This chapter introduces fundamental concepts, like real-time systems, Integrated Modular Avionics (IMA), and the ARINC 653 specification. Furthermore, previous related work by other researchers is reviewed, concerning other approaches to the use of Linux in real-time systems and aerospace applications.

2.1 Real-time systems

A *real-time system* is that whose progression is specified in terms of *timeliness* requirements dictated by the environment. In other words, a real-time system is that whose computations' correctness is defined both in terms of the logical results and the time at which they are provided [19, 20, 21].

The definition of a real-time system in itself does not specifically define a dependence between the time at which the results of a computation are provided and the correctness of the computation (i.e., it does not completely define what "timeliness" means). There are different *classes* of real-time systems, differing in how demanding their definitions of "timeliness" are: hard real-time, soft real-time, and mission-critical.

Hard real-time systems are those where the environment dictates that no timing failures can occur. Thus, these systems have a *time/utility function* where the utility (and, therefore, the correctness) of a computation drops to zero immediately at the latest desirable instant for result production — the *deadline*.

On the other hand, in *soft real-time systems* occasional timing failures are accepted as a consequence of the kind of impact they may have in the environment (e.g., a online video streaming application may be defined as having only soft-real time requirements, since occasional timing failures result, at most, in a poorer viewing experience). The time/utility function of such system is one that progressively decreases towards zero after the deadline is reached.

Finally, there are the so-called *mission-critical real-time systems*, where timing failures are avoided, but accounted for. In other words, a mission-critical system is prepared to avoid timing failures, but at the same time deal with occasional timing failures as exceptional events (thus mitigating their propagation and consequences to the environment in which the system executes).

Contrary to an easy misconception [19], the goal of real-time systems is not achieved through the path of performance, but through that of *determinism*. Thus, most importantly than augmenting computing power and resources, real-time must deal with the rational use of those resources so as to meet temporal requirements. This gives extreme relevance to the problem of *resource scheduling*, which spans over different facets, like CPU scheduling (also known as *process scheduling*), and input/output (I/O) scheduling. This thesis focuses on an architecture which relies on a two-level hierarchical cyclic scheduling of resources to achieve timeliness (cf. Section 2.3.1 and Chapter 3).

2.2 Integrated Modular Avionics (IMA)

Federated avionics are a legacy kind of architecture which makes use of distributed avionics functions packaged as self-contained units: Line Replaceable Units (LRU) and Line Replaceable Modules (LRM) [2]. An avionics system can be comprised of multiple LRUs or LRMs, potentially built by different contractors. What distinguishes LRUs from LRMs is that, while the former are potentially built according to independent specifications, the latter consummate a philosophy in which the use of a common specification is defended [22]. With each avionics function having its own dedicated (and sometimes physically apart) computer resources, which cannot be reallocated at runtime, inefficient resource utilization is a potential drawback from the inherent independence of faults [23, 24].

As a challenge to the traditional federated avionics system architecture, the *Integrated Modular Avionics (IMA)* [1] concept emerged. IMA architectures employ a high-integrity, partitioned environment that hosts multiple avionics functions of different criticalities on a shared computing platform. IMA addresses the needs of modern systems, such as optimizing the allocation of computing resources, reducing size, weight and power consumption (a set of common needs in the area of avionics, which is commonly represented by the acronym SWaP), and consolidation development efforts (releasing the developer from focusing on the target platform, in favor of focusing on the software and easier development and certification processes) [2]. Figure 2.1 portrays a basic example of the layered architecture of a IMA module.

A newer trend in the aeronautic industry, combining the advantages from both the legacy federated avionics and the Integrated Modular Avionics is the

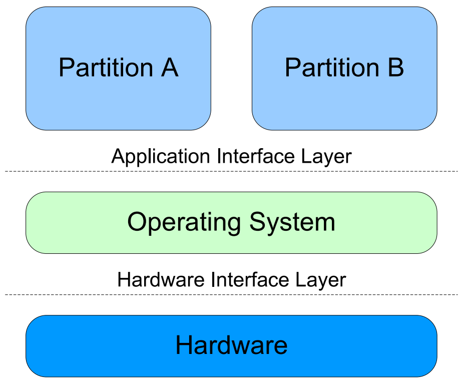


Figure 2.1: Basic architecture of a IMA computing module

Distributed Integrated Modular Avionics (DIMA) concept [25].

2.3 ARINC 653

The *ARINC 653 specification* [3, 4], adopted by the Airlines Electronic Engineering Committee in 1996, is a fundamental block from the Integrated Modular Avionics (IMA) definition [1], where the partitioning concept emerges for protection and functional separation between applications, usually for fault containment and ease of validation, verification, and certification [3, 26]. Examples of its application in the civil aviation industry include the operating systems shipped in the Airbus A380 and Boeing 787 commercial aircrafts. The ARINC 653 specification defines a standard interface between the software applications and the underlying operating system, known as application executive (APEX) interface.

The architecture of a standard ARINC 653 system is sketched in Figure 2.2. At the application software layer, each application is executed in a confined context, dubbed *partition* in ARINC 653 terminology [3]. The application software layer may include system partitions intended to manage interactions with specific hardware devices.

Application partitions consist in general of one or more processes and can only use the services provided by a logical application executive (APEX) interface, as defined in the ARINC 653 specification [3]. System partitions may use also specific functions provided by the core software layer (e.g. hardware interfacing and device drivers), being allowed to bypass the standard APEX interface. The main parameters caracterizing each partition are its criticality level, its period, and its duration [23]. The notions of period and duration are related to the employment of temporal partitioning of computing resources, which will be detailed in Section 2.3.1.

The execution environment provided by the OS kernel module must furnish a

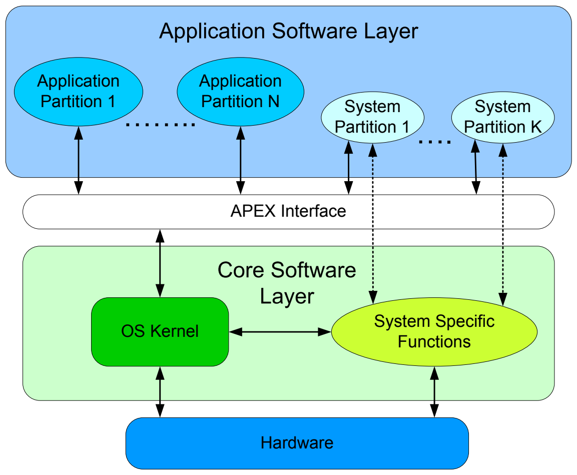


Figure 2.2: Standard ARINC 653 architecture

relevant set of operating system services, such as process scheduling and management, time and clock management, and inter-process synchronization and communication.

2.3.1 Time and space partitioning

Time partitioning consists of the time-sliced allocation of computing resources to hosted applications, achieved in ARINC 653 through a fixed, cyclic scheduling of partitions over a major time frame (MTF) [3, 23].[1] This way, strong temporal segregation is achieved, in which activities inside each partition do not affect the timeliness of activities executing inside the remaining partitions in the system. Processes inside a partition are scheduled according to a priority preemptive algorithm [3].

For a well designed system, the duration of the MTF will correspond to the least common multiple of the partitions' periods (so that it is a multiple of each individual partition's period). The *period* of a partition is the interval at which computing resources are assigned to it; the amount of time, per period, during which each partition owns computing resources is called its *duration*.

Space partitioning concerns preventing applications from having write access to memory zones outside those belonging to its partition. In ARINC 653, this is implicitly ensured by the concept of partitions; the specification makes no considerations on how the operating system achieves space partitioning, it only assumes and requires it [3, 7].

[1] This is the basic partition scheduling behaviour, optionally extended with multiple schedules (cf. Section 2.3.4).

2.3.2 Health Monitoring

The Health Monitoring (HM) functions consist in a set of mechanisms to monitor system resources and application components. The HM helps to isolate faults and to prevent failures from propagating. Within the scope of the ARINC 653 standard specification the HM functions are defined for process, partition and system levels [3].

2.3.3 ARINC 653 Service Interface

The ARINC 653 service requests define the application executive APEX interface layer provided to the application software developer and the facilities the core software layer shall supply. A set of services is mandatory for strict compliance with the ARINC 653 standard [3], grouped into the following major categories:

Partition management

Partition management comprises services for obtaining the current partition's status, and for setting the partition's operating mode (NORMAL, IDLE, COLD_START, or WARM_START). NORMAL is the operational mode of a partition, when process scheduling is active, while IDLE means that the partition is not executing any processes. Both COLD_START and WARM_START indicate an initialization phase is in progress.

Process management

Process management services include obtaining process information (identifier, status), creating processes, modifying priorities, controlling the state of processes (suspend/resume, stop/start) and enabling/disabling process preemption at the partition level.

Time management

Time management services implement timed waits, periodic waits, deadline postponing (replenishment), and obtainment of the current value of the system clock.

Intrapartition communication

Intrapartition communication encompasses interprocess communication and process synchronization. Interprocess communication is possible via message buffers and blackboards, and process synchronization benefits from services to support semaphores and events.

Interpartition communication

Interpartition communication services provide the creation and read/write access to two different abstractions: sampling ports (where each occurrence of a message overwrites the previous one) and queuing ports (where a finite number of messages is stored in FIFO order).

Health monitoring

Health monitoring services provide means for partitions to signal any detected erroneous behaviors to the health monitoring functions, create error handler processes (which have a highest, non-modifiable, priority, and are not preemptible), and invoke the handler process defined for a given error.

2.3.4 ARINC 653 Extended Services

The Part 2 of the ARINC 653 specification [4] adds, to the aforementioned mandatory services, optional services or extensions to the required services. These include the Multiple Module Schedules services, which are the basis of an improvement in the AIR architecture called *Mode-based Schedules*. This is described in depth in Section 4.2.1.

2.4 IMA and ARINC 653 in space

Having identified similarity with the requirements of the avionics applications, the space industry has developed an interest in IMA and its associated concepts, like time and space partitioning, for space applications. A TSP working group, comprising elements from the European Space Agency, the French space agency (CNES, Centre National d'Études Spatiales), Astrium and Thales Alenia Space (two civil and defense space systems and services providers), has been established provide use case descriptions, establish requirements assess the impact on hardware and software, and identify issues that require further prototyping or feasibility assessment [6].

Applying the IMA philosophy to space provides an easier integration of software modules, allows independent validation of critical functions, aids fault containment, and facilitates the implementation of security modules sharing onboard resources. Space agencies, such as ESA, and industrial partners have established user requirements of applying IMA to space, and identified no fundamental and technological feasibility impediments [27].

2.5 Linux and real-time

The initial approach for introducing hard real-time processes in Linux-based operating systems was given by [12] in the *RTLinux* design approach, in which real-time behavior is secured through a low-level specific-purpose microkernel inserted between the hardware infrastructure and the Linux operating system kernel, as pictured in Figure 2.3. The Linux kernel and applications run as the idle task of the real-time microkernel.

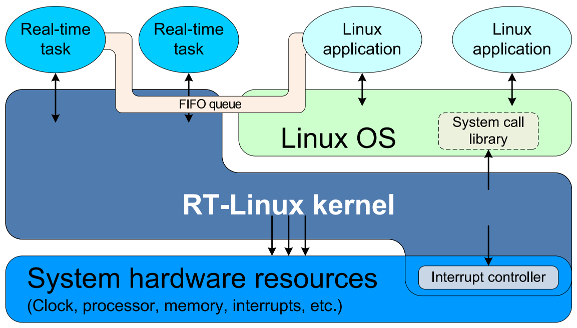
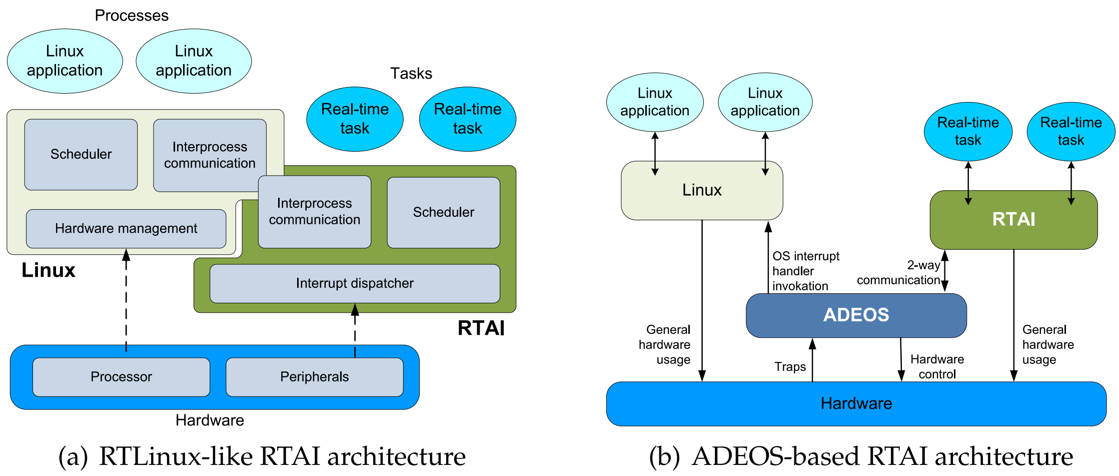


Figure 2.3: RTLinux architecture (adapted from [21])

The *Real-Time Application Interface for Linux (RTAI)* [13] initially employed the RTLinux design concept, as portrayed in Figure 2.4(a), adding features absent from the latter at the time (like floating-point support). However, patent issues led to the later substitution of the RTLinux hardware abstraction layer (RTHAL) for ADEOS (Adaptive Domain Environment for Operating Systems) [28, 29], a resource virtualization layer which poses itself as a kernel module, taking over the innermost protection ring to intercept traps and interrupts. The ADEOS-based architecture of RTAI is pictured in Figure 2.4(b).



(a) RTLinux-like RTAI architecture (b) ADEOS-based RTAI architecture

Figure 2.4: The two historical approaches to RTAI

Interrupt handling by RTAI/ADEOS is based on an interrupt pipe, propagating interrupts through the different domains (cf. Figure 2.5). Both the concept of domains and this interrupt pipe approach were inspired by the SPACE approach to operating system implementation [29, 30].

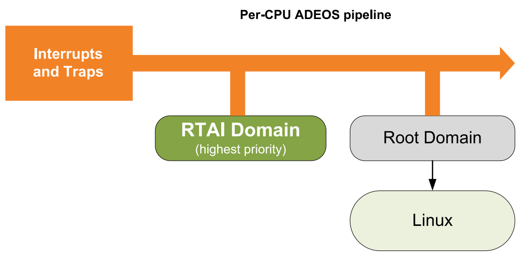


Figure 2.5: Interrupt pipe in ADEOS-based RTAI

A similar approach is followed in the *xLuna* operating system [14], this time around supported on the RTEMS (the Real-Time Executive for Multiprocessor Systems) kernel [31]. xLuna was developed by Critical Software under a contract with ESA's European Space Research and Technology Centre (ESTEC). A xLuna microkernel mediates the interactions between the hardware and the operating system components (RTEMS and Linux). Linux runs as the lowest priority task. The xLuna has been targeted to run on the SPARC LEON processor, extensively used in aerospace systems. Figure 2.6 depicts the architecture of the xLuna kernel.

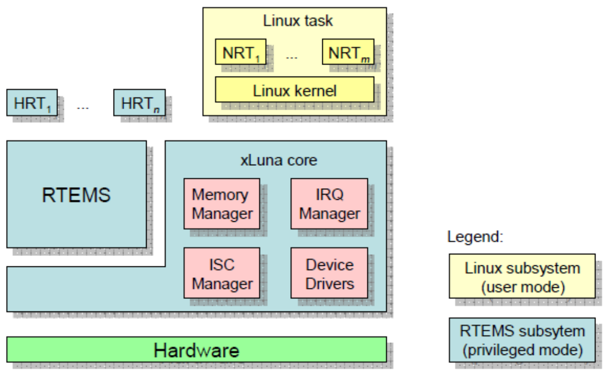


Figure 2.6: xLuna architecture

In all the aforementioned approaches, the Linux kernel and, consequently, the applications running on top of it have no guaranteed processing time. Although this does not compromise the requirements of real-time applications, it may hinder the minimum desirable performance of non-real-time Linux applications. In AIR (cf. Chapter 3), all partitions, including those running non-critical activities such as Linux processes, have a guaranteed execution time window in the scheduling of computing resources to partitions.

In the architecture on which the work of this thesis focuses, the approach to the integration of operating systems in a partitioned environment is that of allowing the same operating system to run, unmodified, both on its own and as a partition operating system.

2.6 Linux in aerospace applications

As regards the use of Linux in avionics applications, the Avionics and Simulation Department of Airbus has conducted research on the viability of migrating a soft real-time system from a POSIX off-the-shelf RTOS to a standard Linux kernel (coupled with BusyBox). This research found no technical obstacle or incompatibility which could compromise the adoption of Linux, while nevertheless identifying some points where the complexity of optimizing system resource usage should be reduced in favor of achieving the deterministic behavior required by critical embedded systems [32].

In the space industry, ESA is widely interested in adopting platform-independent [33] and open source [27] operating system solutions, like RTEMS [31] and Linux. This manifests itself in financial and technical support for projects like xLuna [14] (cf. Section 2.5) and AIR (on which this thesis builds upon), and in the efforts of migrating ground segment data systems from Sun Solaris to Linux [33].

2.7 Integrating Linux in partitioned architectures

AIR (ARINC 653 In Space RTOS) [7, 8] is the definition of a partitioned architecture for aerospace applications, derived from the the interest of the European Space Agency (ESA) on the concepts of the aviation-related IMA [1] and ARINC 653 [3] specifications. Initially a proof of concept for the adaptation of RTEMS [31] to offer the ARINC 653 application executive interface and functionality, the design of the AIR architecture evolved to foresee the employment of different real-time operating systems along the partitions. The AIR architecture resulted from ESA-sponsored initiatives featuring collaboration between the Faculty of Sciences of the University of Lisbon and Skysoft Portugal.

XtratuM [34], developed by the Universidad Politécnica de Valencia in a contract with CNES, is a paravirtualizing hypervisor aiming to run multiple operating systems in a robustly partitioned fashion, on the SPARC V8 architecture (LEON processors). Linux has been ported to run as a partition operating system on top of XtratuM, but such portings are only available for the Intel x86 architecture. Despite having had its Application Programming Interface (API) and internal operations adapted to resemble the ARINC 653 specification [3], XtratuM does not intend to comply with ARINC 653 [35].

This thesis builds upon the operating system heterogeneity inherent to the AIR architecture, extending it to generic non-real-time operating systems, such as Linux (which is used as a case study).

2.8 Summary

This chapter presented concepts, literature and related work on the basic concepts fundamental to the full understanding of the work performed in the present thesis: real-time systems, Integrated Modular Avionics (IMA), ARINC 653, the adoption of IMA/ARINC 653 concepts in space, and Linux on real-time and aerospace systems.

In the following chapter, a partitioned architecture based on the ARINC 653 principles — the AIR architecture — is presented in detail.

Chapter 3

AIR: ARINC 653 in Space RTOS

The technological interest of ESA in the concepts of IMA and ARINC 653 [3, 5, 6, 27] led to the development of a proof of concept [7, 8] and a demonstration of feasibility of use, within the scope of the ESA innovation triangular initiative (project AIR). The "AIR-II: ARINC 653 Interface in Space RTOS - Industrial Initiative" activities continue the work done in AIR, with the goal of becoming closer to a real system by improving and completing the key ideas identified.

This chapter describes the AIR architecture, upon which this work proposes to integrate generic operating systems, as it was defined when this thesis's work began. Improvements made since then, which constitute contributions of this work inserted in the AIR-II activities, are described in Chapter 4.

3.1 System architecture

The design of the AIR architecture in essence preserves the hardware and RTOS independence defined within the scope of the ARINC 653 specification [3, 8, 7]. The AIR system architecture is pictured in Figure 3.1. The different components will now be presented in detail.

3.1.1 Partition Management Kernel (PMK)

The component, transversal to all partitions, which is responsible for providing the strong properties of the system, is the AIR Partition Management Kernel (PMK). The AIR PMK is a simple microkernel that efficiently handles:

- *Partition scheduling*, selecting at given times which partition owns system resources, namely the processing infrastructure. The AIR Partition Scheduler secures temporal segregation using a single fixed cyclic scheduler.

- *Partition dispatching*, saving the execution context of the running partition and restoring the execution context for the heir partition. The *AIR Partition*

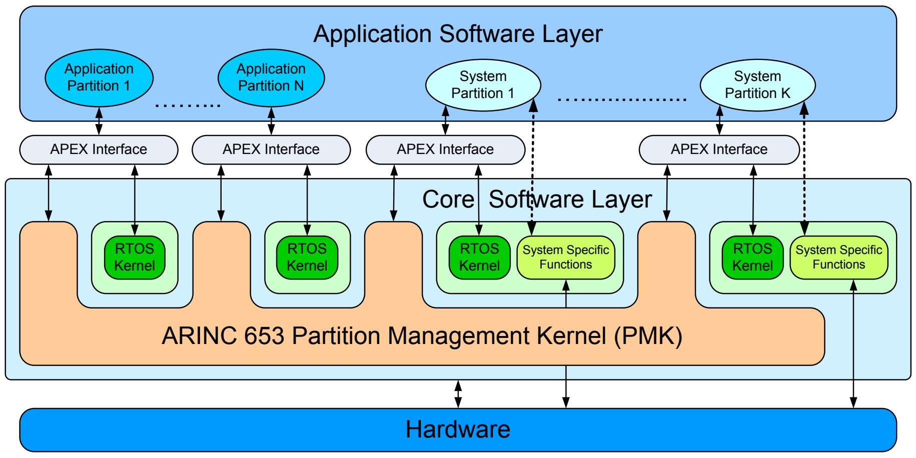


Figure 3.1: Overview of the AIR multi-executive core system architecture

Dispatcher secures the management of all provisions required to guarantee spatial segregation.

- *Interpartition communication*, allowing the exchange of information between different partitions without violating spatial segregation constraints.

Partition scheduling

System clock tick interrupts are caught by the AIR Hardware Abstraction Layer (HAL), which in turn passes each one to the AIR PMK, namely to the PMK Time Manager internal component. This component increments the clock tick count, and passes the interrupt to the PMK Partition Scheduler. The latter verifies the Partition Scheduling Table (PST) to determine if we are in the presence of a partition preemption point at the current tick (and, if so, what is the *heir partition*). The PST lists partition preemption points instants relatively to the beginning of the major time frame.

Partition dispatching

The processing of the system clock tick interrupt continues at the PMK Partition Dispatcher component, where (in the presence of a partition preemption point) the context of the running partition is saved, and the context of the heir partition is restored. It is also the PMK Partition Dispatcher that calculates the number of clock ticks elapsed during the most recent preemption of the heir partition, which will be ultimately be announced via a modified system clock tick interrupt service routine (ISR) on the native partition operating system.

The described functioning of the AIR partition scheduling and dispatching mechanisms is schematized in Figure 3.2. The flowchart there contained particularly details how clock tick accounting is kept for both the (previously) running partition and the heir partition.

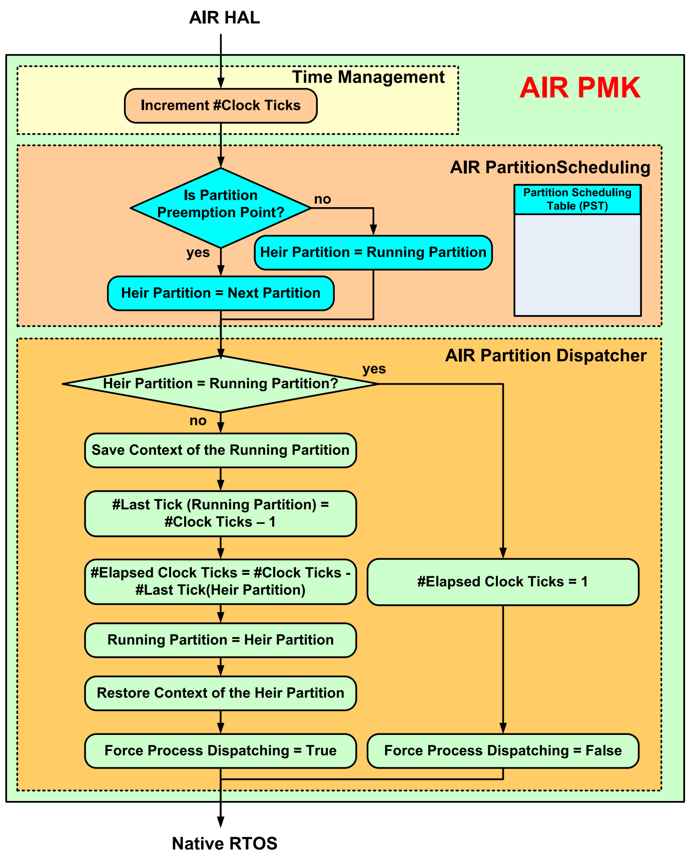

AIR HAL
Time Management
AIR PMK
Increment #Clock Ticks
AIR PartitionScheduling
Is Partition Preemption Point?
no
yes
Heir Partition = Running Partition
Heir Partition = Next Partition
Partition Scheduling Table (PST)
AIR Partition Dispatcher
Heir Partition = Running Partition?
yes
no
Save Context of the Running Partition
#Last Tick (Running Partition) = #Clock Ticks – 1
#Elapsed Clock Ticks = #Clock Ticks - #Last Tick(Heir Partition)
#Elapsed Clock Ticks = 1
Running Partition = Heir Partition
Restore Context of the Heir Partition
Force Process Dispatching = True
Force Process Dispatching = False
Native RTOS

Figure 3.2: System clock tick processing at the AIR PMK level

Interpartition communication

Interpartition communication was introduced in AIR-II and its relation with spatial segregation implies the use of specific executive interface services encapsulating and providing the transfer of data from one partition to another without violating spatial segregation constraints [3].

The interpartition communication abstractions required by the ARINC 653, sampling ports and queuing ports, model each partition's way to communicate (send **or** receive messages) through a communication channel.

The core AIR interpartition communication mechanisms are integrated at the APEX (core) level. This means that each partition's instance of the APEX holds the necessary mechanisms and structures to support the ports that partition may use during system execution (which are entirely defined at system configuration time, as per ARINC 653). For sampling ports, this consists of a buffer to store one message; on a given partition's source port, each message sent by an application overwrites any previous message there might be, and remains in the port until the channel transmits it (or until overwritten, whichever happens first). In the case of a destination port, messages are successively overwritten as new messages are delivered by the channel, thus allowing applications in the partition to always read the latest message. In the case of queuing ports, there is space for a configured number of messages, and they are never overwritten: messages in a source queuing port will be stored in the queue until they are transmitted, and messages in a destination port will remain queued until they are received by the application.

The AIR PMK shall manage the memory protection mechanisms, to guarantee that each partition's ports are not accessible by any other partition, and the memory-to-memory copy procedures needed to implement the local communication channels. Although the AIR architecture design does not currently address interpartition communication to that degree, the ARINC 653 specification foresees that these channels can connect two different nodes [3].

The described interpartition communication mechanisms are exemplified in Figure 3.3, with the following partition/ports combination:

- Partition X: one source queuing port with a 4-message buffer;
- Partition Y: one destination queuing port with a 3-message buffer, and one destination sampling port;
- Partition Z: one source sampling port.

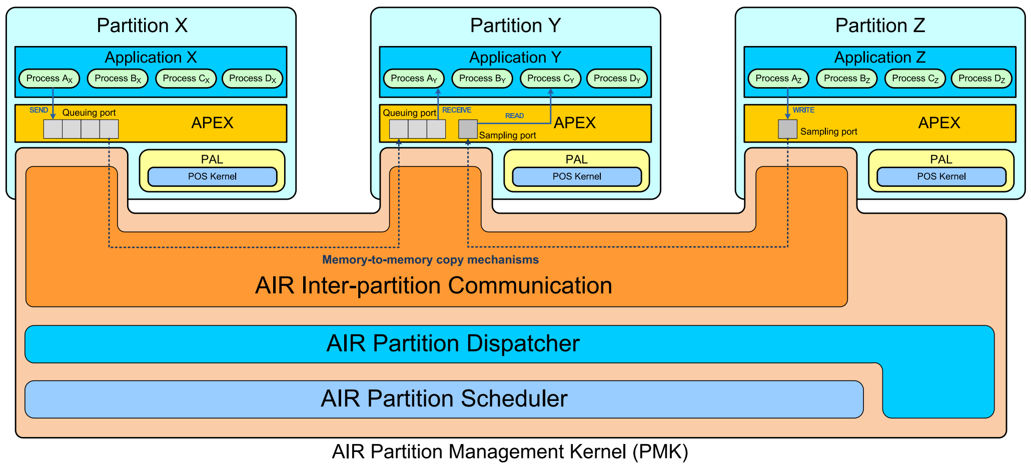


Figure 3.3: APEX (core), AIR PMK components, and their relation with interpartition communication

3.1.2 Application Executive (APEX) Interface

Another fundamental component concerns the *APEX interface*, defining for each partition in the system a set of services in strict conformity with the ARINC 653 specification. The AIR APEX was improved in AIR-II, and consists of two components: the APEX Core Layer and the APEX Layer Interface. The relationship between such two components can be seen in Figure 3.4. The APEX core Layer implements the advanced notion of *Portable APEX* intended to ensure portability between the different OS supported by AIR [36]. It exploits the POSIX application programming interface that is currently available on most (RT)OS [37]. An optimized implementation may invoke directly the native (RT)OS service primitives. In the AIR activities, the APEX was developed by the Skysoft partner, and received input from the activities of another project where the APEX was present — ARINC 653 Simulator for Modular Space Based Applications (AMOBA) [25, 38].

On top of the APEX core layer, the partition and process management services, the intra- and interpartition communication services and the health monitoring services are built. The partition management, interpartition communication and health monitoring services rely additionally on the PMK service interface. In this respect, the PMK provides the partition-wise handling of memory protection descriptors.

The APEX also coordinates when required the interactions with the AIR Health Monitor, e.g. upon the detection of an error.

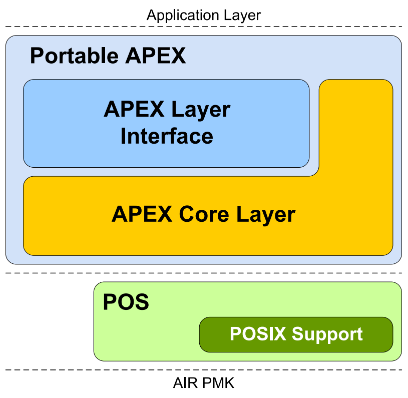


Figure 3.4: Internal architecture of the APEX Interface

3.1.3 AIR Health Monitoring

The AIR Health Monitor introduced in AIR-II is responsible for handling hardware and software errors (like deadlines missed, memory protection violations, bounds violation or hardware failures). As much as possible, it will isolate the error propagation within its domain of occurrence: process level errors will cause an application error handler to be invoked, while partition level errors trigger a response action defined by the partition Health Monitor table in the ARINC 653 configuration. The response action may be shutting down the entire partition, reinitializing the partition again or simply ignoring the error. Partition errors may be also raised as a consequence of process level errors that cannot be handled by the application error handler. Errors detected at system level may lead the entire system to be stopped or reinitialized.

The design of AIR allows Health Monitoring handlers to simply replace existing handlers or to be added to existing ones in pre- and/or post-processing modes.

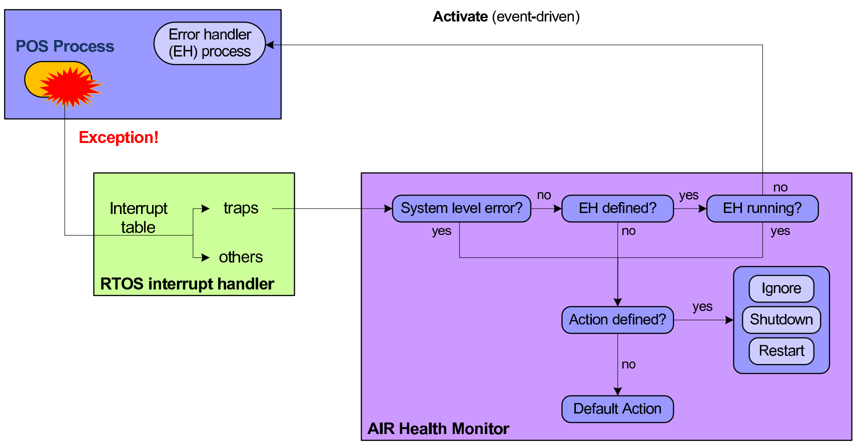


Figure 3.5: AIR-II Health Monitoring Mechanisms

3.2 Robust time and space partitioning

Time and space partitioning (TSP) is a fundamental concept in the software architecture used in the shared computing platform of an IMA-based system [6]. ARINC 653, being a significant block IMA, revolves around the notion of TSP, and the AIR architecture definition provides its implementation of this required notion. The following sections describe the robust implementation of time and space partitioning present in the AIR architecture.

3.2.1 Strict temporal segregation

The ARINC 653 standard specification [3] restricts the processing time assigned to each partition, in conformity with given configuration parameters. The scheduling of partitions defined by the ARINC 653 standard is strictly deterministic over time. Each partition has a fixed temporal window in which it has control over the computational platform. Each partition is scheduled on a fixed, cyclic basis. This allows the AIR architecture to cope with hard real-time requirements and, in a given sense, opens room for the temporal composability of applications.

To ensure flexibility and modularity, instead of modifying the RTOS scheduler to extend it to the partitioning concept, the approach followed in the AIR architecture uses one instance of the native RTOS scheduler for process scheduling inside each partition. No fundamental modification is needed to the functionality of the RTOS process scheduler for its integration in the AIR system. Such a *two-level hierarchical scheduler* approach secures partition and process scheduler decoupling, thus allowing the use of different operating systems in different partitions (e.g. RTEMS [31], eCos [39, 40], . . .).

3.2.2 Spatial segregation

Robust partitioning comprises the protection of each partition's memory addressing space, to be provided by specific memory protection mechanisms usually implemented in a hardware memory management unit (MMU). It requires also a functional protection concerning the management of privilege levels and restrictions to the execution of privileged instructions. A basic set of such mechanisms do exist in the Intel IA-32 architecture (widely used in everyday applications) and, to a given extent, in the SPARC LEON processor core, crucial to the European space industry.

Space partitioning abstraction layer

A highly modular design approach has also been followed in the support of AIR spatial segregation. Spatial segregation requirements, specified in ARINC 653 configuration files with the assistance of development tools support, are described in run-time through a high-level processor independent abstraction layer. A set of descriptors is provided per partition, primarily corresponding to the several levels of execution of an activity (e.g. application, POS kernel and AIR PMK) and to its different memory sections (e.g. code, data and stack), as illustrated in the diagram of Figure 3.6.

In AIR Technology, the definition of the high-level abstract space partitioning takes into account the semantics expected by user-level application programming. At each partition, the application environment inherits the execution model of the corresponding POS and/or its language run-time environment. This is true for system partitions and may be applied also to application partitions, using only the standard APEX interface.

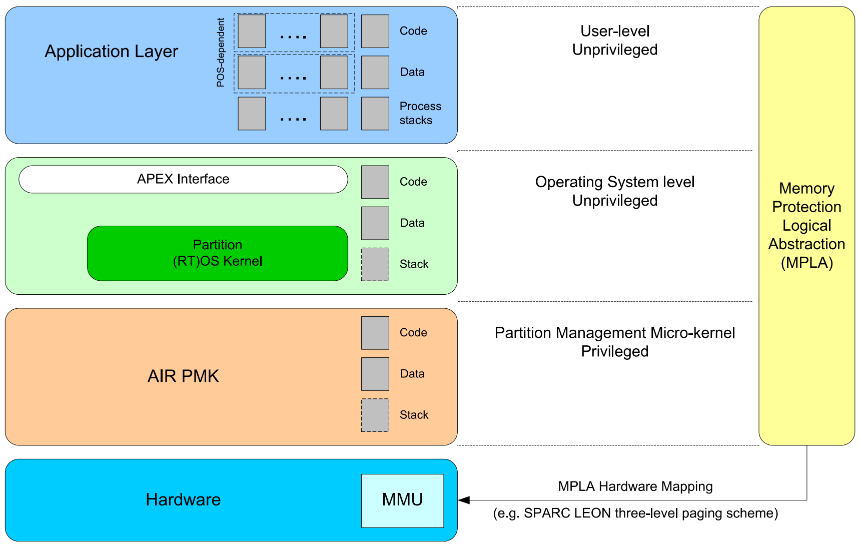


Figure 3.6: AIR-II Spatial Segregation Scheme

Memory protection mechanisms

The high-level abstract space partitioning description needs to be mapped in run-time to the specific processor memory protection mechanisms, possibly exploiting the availability of a memory management unit (MMU). Possible examples of such mapping are: the fence registers of the ATMEL AT697E SPARC V8 LEON2 processor or the Gaisler SPARC V8 LEON3 three-level page-based MMU core.

Mapping of high-level abstract partitioning also includes the management of privilege levels: only the AIR PMK is executed in privileged mode (cf. Figure 3.6). The lack of multiple protection rings, such as it exists in the Intel IA-32 processor architecture, may be mitigated in the SPARC V8 architecture by granting access to a given level only during the execution of services belonging to that level (or lower ones). This may be achieved by activating the corresponding memory protection descriptors upon call of a service primitive, and deactivating them when service execution ends.

3.3 Summary

This chapter has described the essential of the AIR Technology. The AIR technology aims to provide the developers and the integrators of space on-board software with a time- and space-partitioned environment that is standard and in conformity with the ARINC 653 specification [3]. The AIR solution is hardware- and OS-independent. The AIR design allows in principle the versatile integration of both open-source and commercial operating system kernels. Current development makes use of the Real-Time Executive for Multiprocessor Systems (RTEMS) [31] as a significant representative of RTOS kernels [8]. RTEMS is a real-time multitasking kernel qualified for use in space on-board software developments.

The following chapters describe the original contribute of this thesis to the AIR Technology definition. The next chapter will present the enhancements made to the system architecture, in part directly related to the goal of integrating generic non-real-time operating systems in the partitions, but also including timeliness adaptation and monitoring features.

Chapter 4

Improving the AIR Technology

This chapter describes evolutions made to the AIR architecture, which benefited, to a high degree, from motivation and lessons learned from the preliminary stages of approaching the problem of integrating generic operating system in a partitioned architecture.

4.1 The AIR POS Adaptation Layer (PAL)

After the preliminary studies on the integration of generic operating systems in AIR [11], it became evident that an additional layer (between the partition operating system — POS — and the remaining system architecture components) was required for the integration of Linux, while also being apparent that partitions hosting real-time operating systems would also benefit (albeit possibly in different ways) from an identical layer.

On a first approach, this was dubbed the paravirtualization layer, and would reside directly under the POS, between the latter and the AIR PMK. Further developments capitalized on the observations that:

(i) to aid the integration of different POSs, such layer should, not only provide paravirtualization of the POS, but also bring added POS-independence characteristics to the architectural components (such as the AIR PMK); this also confirmed that this component should be present in all partitions;

(ii) the purposes of POS adaptation of this layer implied, not only interactions with the AIR PMK, but also with the APEX, thus it should reside *around* the POS (not *below*).

The second approach to this new layer resulted in the integration as a new component in the AIR architecture, the *AIR POS Adaptation Layer (PAL)*. This integration is pictured in Figure 4.1, which represents the current and improved

AIR system architecture, resulting from activities developed in the course of this thesis's work.

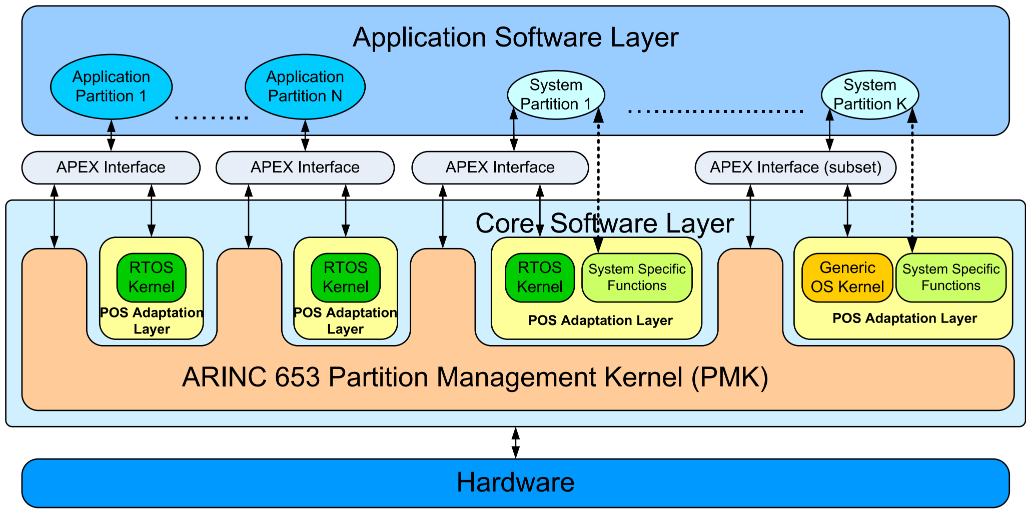


Figure 4.1: Overview of the improved AIR architecture

The integration of the Portable APEX (cf. Figure 3.4) was slightly modified, in function of the introduction of the AIR PAL component. The new approach is pictured in Figure 4.2. The APEX will now take advantage of mappings or services provided by the AIR PAL, as detailed in Section 4.1.1.

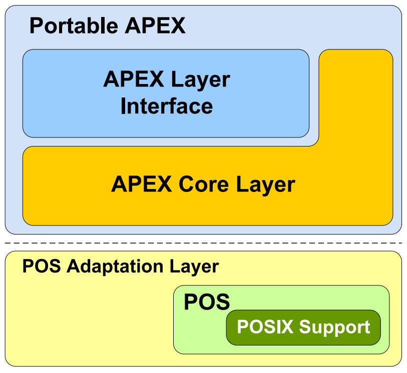


Figure 4.2: Architecture of the re-designed APEX Interface

The improvements enabled by the introduction of the AIR PAL component can be divided in three categories: *(i)* architectural properties; *(ii)* engineering of AIR components, and; *(iii)* leaner development processes, stemming from separation of concerns. These improvements will now be detailed.

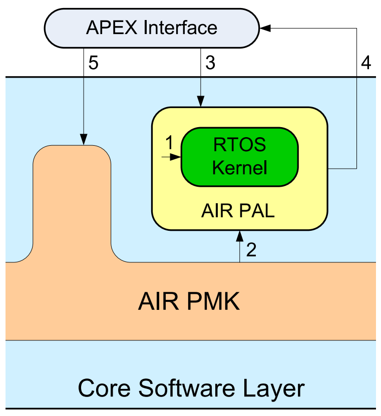


Figure 4.3: AIR Partition Adaptation Layer (PAL)

4.1.1 Architectural properties

Although foreseeing the use of different operating systems on each partition, and providing applications with a POS-independent interface (the APEX), the original AIR PMK design [8], as presented in Chapter 3, did, by omission, lend itself to too much of a burden on the matter of adaptation to a new POS. Whenever support to a new POS was to be added, the AIR PMK would have to be modified. Although changes will often be slight and focused, they would hinder previous or ongoing validation and/or certification efforts on the AIR PMK.

By wrapping each partition's operating system kernel inside an adaptation layer (the AIR PAL), the AIR PMK can act upon the POS in a way that is agnostic of the latter, when necessary. Upon the need to add support to a different POS, the AIR PMK remains unaltered, with support being coded by developing an adequate PAL — from scratch, or from an existing one. It is still necessary to code the integration of a new POS, but this time around no modification to the AIR PMK is needed (and, thus, its verification, validation and/or certification status remains the same).

This was the key idea in introducing the AIR PAL component, where the interaction with the AIR PMK does concern (cf. Figure 4.3):

1. the execution of POS initialization procedures;
2. interfacing with AIR PMK components, including: the partition scheduler and dispatcher; the low-level hardware abstraction layer, managing the access to raw computer platform resources; effective mechanisms to support interrupt-driven or polled-mode input/output actions and its relation with partition scheduling.

However, the PAL also benefits the design of other AIR components, such as the Portable APEX and the AIR Health Monitor (HM), as follows:

3. mapping of APEX system calls onto the services provided by the POS or onto specific services built into the AIR architecture, for example, actions for the management of the time-budget of the processes, which may include: timeliness control and time-related functions; monitoring of hard real-time process deadline and its verification; notification that a process has completed its execution and entered a wait state; advanced time-budget transfers from processes in hard real-time partitions to processes in non real-time partitions;

4. support to interactions with the AIR HM component, usually triggered by the raising of an exception [15].

For completeness sake, Figure 4.3 also specifies the interactions between the APEX and the AIR PMK components that do not directly involve PAL calls, such as those concerning:

5. partition management and mapping of APEX interpartition communication service requests into memory writing/reading accesses and, if necessary, memory-to-memory copy actions, without violating space segregation constraints [15, 3].

4.1.2 Component engineering

Besides consolidating the POS-independence of the AIR PMK, Portable APEX and AIR HM components, the AIR PAL can also make up for some non-optimal or inappropriate behavior of the native POS implementation of some function. Also, by providing these surrogate functions — intended to be called in spite of the native ones — instead of creating patches to be applied to the RTOS's source code, we extend the lifetime of the support to a given RTOS through more subsequent versions of the latter. The reason for this is that the patches' mapping onto their target relies on source code file names and line numbers, whereas the AIR PAL relies on function prototypes and behaviors.

The improvements on architectural properties and component engineering are closely related, and constitute the basis of a *flexible integration of partition operating systems*. We now illustrate, through a use case scenario, the architectural and engineering problems that the AIR PAL solves, and how.

Use case scenario: partition scheduling and dispatching

In the AIR PMK design, as described in Section 3.1.1, partition scheduling and dispatching culminates in the announcement to the heir partition of the number of clock ticks elapsed since it was last preempted.

Calling the native POS system clock tick announcement routine for this purpose poses two problems. The first one concerns architectural features, namely the AIR PMK's POS-independence: different POSs may offer calls with differing prototypes for this purpose, forcing the PMK to be aware of the POS to call the right function. The other problem is that different POS may also offer calls with differing or inappropriate behaviors; specifically, one must ensure that the joint announcement of the clock ticks elapsed since the last preemption of the heir partition is an atomic action. This is a fundamental condition to ensure robust temporal partitioning.

Addressing these issues, a PAL implementation for any POS offers a surrogate clock tick announcement routine, which prototype should follow a defined specification. Depending on how each POS natively implements the routine we are encapsulating, the PAL implementation can range from acting as a simple alias, to being a complete rewrite of the native code. This use case scenario is illustrated in Figure 4.4, using the RTEMS [31] as an example; for RTEMS, the AIR PAL implementation would consist only of iterating the appropriate number of times on the native RTEMS procedure for (single) clock tick announcement (rtems_clock_tick).

4.1.3 Separation of concerns

Another reason against patching the POS so as to obtain the integration and intended behavior for running on the AIR architecture is that it would break the desired *separation of concerns*, thus undermining an otherwise streamlined development process.

When application developers are working on an application to run on a given partition, they should have to be concerned only with functional implementation. System partition developers will nevertheless typically use and need to know of OS-specific interfaces, but even these should not be concerned on how the underlying POS is adapted to the AIR architecture. Thus, the operational burden of this adaptation should reside on the side of the core software layer components' developers/maintainers. This is another one of the main reasons why having the PAL provide new implementations to other components (PMK, APEX, HM) by wrapping the POS is a more elegant approach than modifying (patching) the latter.

In turn, on the core software layer side, centering the adaptation to every new POS on the AIR PMK is not desired, because it would require further constant revalidation work (with certification in mind), while at the same time diverting the focus of the AIR maintainers from what should be the main concerns of the PMK — ensuring robust temporal and spatial segregation.

By consolidating the separation of concerns in the AIR architecture, the devel-

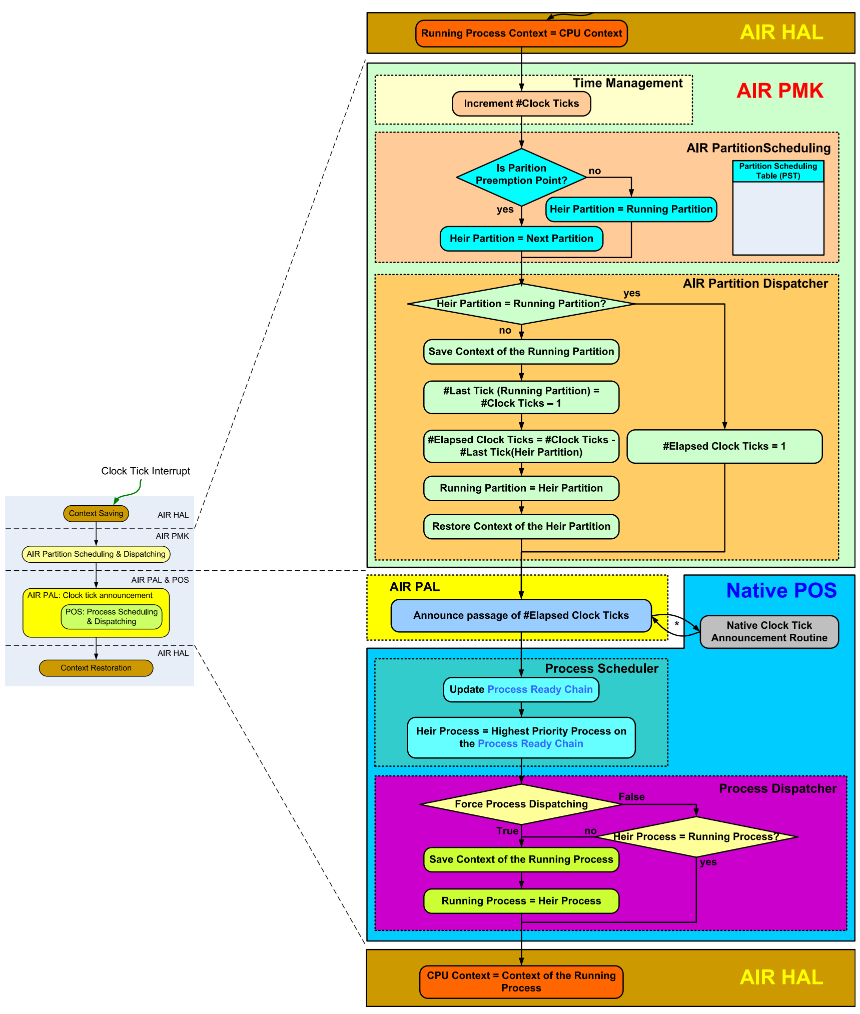

Clock Tick Interrupt
Context Saving
AIR HAL
AIR PMK
AIR Partition Scheduling & Dispatching
AIR PAL & POS
AIR PAL: Clock tick announcement
POS: Process Scheduling & Dispatching
AIR HAL
Context Restoration
Running Process Context = CPU Context
AIR HAL
Time Management
Increment #Clock Ticks
AIR PMK
AIR PartitionScheduling
Is Parition Preemption Point?
no
yes
Heir Partition = Running Partition
Heir Partition = Next Partition
Partition Scheduling Table (PST)
AIR Partition Dispatcher
Heir Partition = Running Partition?
yes
no
Save Context of the Running Partition
#Last Tick (Running Partition) = #Clock Ticks – 1
#Elapsed Clock Ticks = #Clock Ticks - #Last Tick(Heir Partition)
#Elapsed Clock Ticks = 1
Running Partition = Heir Partition
Restore Context of the Heir Partition
AIR PAL
Announce passage of #Elapsed Clock Ticks
*
Native POS
Native Clock Tick Announcement Routine
Process Scheduler
Update Process Ready Chain
Heir Process = Highest Priority Process on the Process Ready Chain
Process Dispatcher
Force Process Dispatching
False
True
no
Heir Process = Running Process?
yes
Save Context of the Running Process
Running Process = Heir Process
CPU Context = Context of the Running Process
AIR HAL

Figure 4.4: Use case scenario of AIR PAL in the PMK partition scheduling and dispatch scheme

opment workflow can rely much more on reusable components — also known as *building blocks*. This methodology is widely praised by the space industry [27]. For instance, as soon as an AIR PAL for a given POS is available, it can be used by different partition application developers. On another aspect, system integrators are given the partition applications in the form of reusable objects, allowing them great flexibility on how to integrate them in different modules.

This leads to leaner software development processes, with less overhead spent on interactions between different stakeholders (partition application developers, system integrators, etc.).

Figure 4.5 illustrates the entire development process, with emphasis on the different roles involved. The interactions between different intervenients are essentially one-way and evolve incrementally towards the full integrated system. By building the system from independent object files, modifications to one part of the system will not affect the remaining parts, whose object files will remain

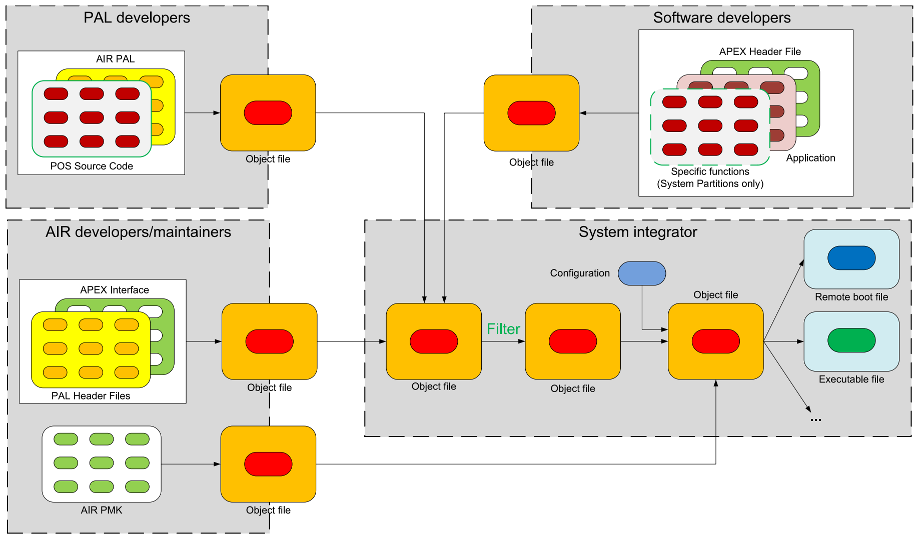


Figure 4.5: Optimized development process, enabled by separation of concerns

unmodified, thus not requiring new validation efforts.

4.2 Enhanced timeliness mechanisms

In the course of the present work, the design of AIR PMK was enhanced so as to incorporate:

- simple, yet highly effective, *mode-based scheduling* mechanisms, to enable switching among multiple partition schedules and execute appropriate actions the first time a partition is dispatched after a schedule change; this is useful both to implement fault tolerance mechanisms and to optimize partition scheduling for different modes of operation or phases of a mission [4];
- a POS-independent optimized scheme, inserted at the AIR PMK level before the partition process scheduler, aiming to verify the fulfillment of hard real-time *process deadlines*; only the earliest deadline is verified at each system clock tick, and violations of deadlines are reported to the health monitoring mechanisms.

4.2.1 Mode-based schedules

The ARINC 653 specification [3] defines a static scheduling of partitions, cyclically obeying to a Partition Scheduling Table (PST) defined offline, at system integration time. This is certainly very restricting in terms of configuration and fault tolerance.

ARINC 653 Part 2 [4], which defines optional extended services, introduces the notion of multiple mode-based schedules. Acknowledging how restrictive a single schedule can be for certain scenarios, the basic mandatory scheme is extended to allow multiple schedules to be defined in the configuration table. At execution time, authorized partitions may request switching between these schedules. Examples of the usefulness of such approach include the adaptation of schedules to different modes/phases (initialization, operation, etc.) and the accommodation of component failures (e.g., assigning a critical program running in a failed processor to another one).

For this purpose, the configuration table is extended in two ways:

1. definition of multiple schedules, with different major time frames, partitions, and respective periods and execution time windows;

2. inclusion of restart actions (ScheduleChangeAction) to be performed, on a per-partition and per-schedule basis, when the schedule is changed.

Support for this functionality includes the provision of following new APEX services.

SET_MODULE_SCHEDULE sets the schedule that will start executing at the top of the next major time frame. It must be invoked by an authorized partition, and have the identifier of an existing schedule as its only parameter. The immediate result is only that of setting the NEXT_SCHEDULE field in the schedule status to the indicated schedule identifier.

At the start of the next major time frame, the following steps are performed to make the schedule switch effective:

1. CURRENT_SCHEDULE is set to NEXT_SCHEDULE (which is the schedule identifier provided in the previous SET_MODULE_SCHEDULE call);

2. partition start is initiated for each partition that was not started during a previous schedule.

Also, each partition in the new schedule running in NORMAL mode will have to be restarted according to its ScheduleChangeAction.[1] This action will take place the first time each partition is scheduled/dispatched after the schedule switch.

Support for the SET_MODULE_SCHEDULE service makes up for virtually the whole of the changes made to support mode-based schedules. These changes

[1]One possibility, IGNORE, is for the partition not to be restarted. Partitions not running in NORMAL mode will not be restarted, regardless of the ScheduleChangeAction defined for them for the newly installed schedule.

concentrate mostly on the AIR Partition Scheduler, and are illustrated in Figure 4.6. The AIR Partition Dispatcher only needs to be modified to execute the pending change action after dispatching each partition for the first time after the schedule switch; the reasoning for *when* schedule change actions ought to be executed is also implemented on the AIR Partition Scheduler.

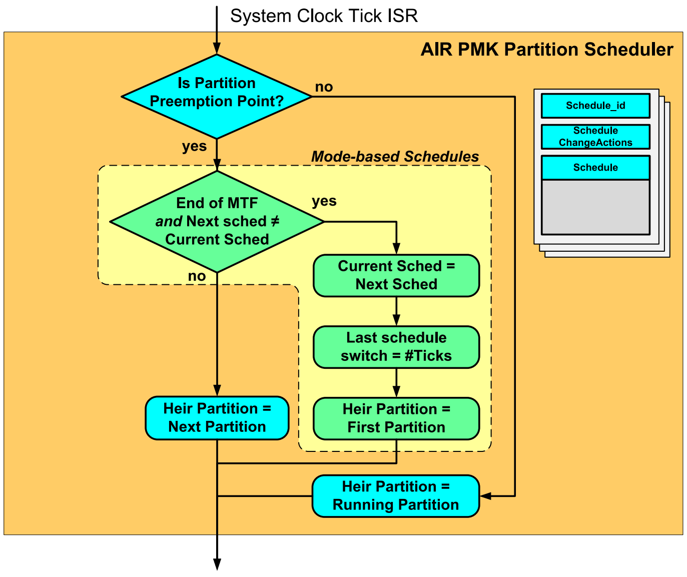

System Clock Tick ISR
AIR PMK Partition Scheduler
Is Partition Preemption Point?
no
yes
Mode-based Schedules
End of MTF *and* Next sched ≠ Current Sched
yes
no
Current Sched = Next Sched
Last schedule switch = #Ticks
Heir Partition = Next Partition
Heir Partition = First Partition
Heir Partition = Running Partition
Schedule_id
Schedule ChangeActions
Schedule

Figure 4.6: AIR Partition Scheduler with support for mode-based schedules

GET_MODULE_SCHEDULE_STATUS allows obtaining the current schedule status information, which is defined in ARINC 653 Part 2 [4] as comprising:

- the time of the last schedule switch (0 if none ever occurred);
- the identifier of the current schedule (CURRENT_SCHEDULE);
- the identifier of the next schedule (NEXT_SCHEDULE), which will be the same as the current schedule if no schedule change is pending for the end of the present major time frame).

GET_MODULE_SCHEDULE_ID returns the identifier of the schedule with a given name (if there is one).

Design and engineering issues

Since the AIR Partition Scheduler and Dispatcher code is invoked at every system clock tick, its code needs to be as efficient as possible. In the AIR implementation, in the best case, only two computations are performed: incrementing the number of clock ticks by one, and verifying if we are in the presence of a partition preemption point (which, in this best case, will turn out false; this will typically be the most frequent situation throughout the system functioning).

To incorporate the mode-based schedules functionality, the Partition Scheduler computations had to become slightly more complex; verifications of the presence of a partition preemption point or the end of a MTF need to rely on the number of clock ticks elapsed since the last schedule switch, and not solely the number of clock ticks since system initialization. The pseudocode for the implementation of the AIR Partition Scheduler, with support for mode-based schedules, is provided in Appendix A (Listing A.2).

The Partition Dispatcher's (cf. Listing A.3) only modification regarding mode-based schedules is the invocation of pending schedule change actions. Part 2 of the ARINC 653 specification [4] does not clearly state whether schedule change actions should be performed immediately after effectively changing schedule (i.e., at the beginning of the first MTF under the new schedule, for all partitions) or performed for each partition as it is dispatched for the first time after the schedule switch. It is nevertheless our understanding that the latter approach is more compliant with the fulfillment of temporal segregation requirements, since each partition's schedule change actions (which may include restarting the partition) will only affect its own execution time window.

4.2.2 Process deadline violation monitoring

Upon the dispatching of a partition, after having announced the passage of clock ticks to the heir partition, it might be the case that one or more deadlines of processes in that partition have been missed while it has been inactive. This can happen in the presence of operational faults, or when uncertainty regarding a process's worst case execution time (WCET) leads to the partition execution time windows being under-dimensioned. Other factors related to faulty system planning (like a periodic process exceeding its deadline because the partition windows have a larger period than the said process and/or the partition period is not a multiple of the process's period) could, in principle, also cause deadline violations; however, such issues can easily be predicted beforehand and avoided using simulation and/or schedulability analysis tools.

In the context of Health Monitoring (HM), ARINC 653 classifies process deadline violation as a process level error (an error that impacts one or more processes in the partition, or the entire partition) [3]. Possible recovery actions in the event of such an error are:

- ignoring the error (logging it, but taking no action);

- logging the error a certain number of times before acting upon it;
- stopping the faulty process and reinitializing it from the entry address;
- stopping the faulty process and starting another process;
- stopping the faulty process, assuming that the partition will detect this and recover;
- restarting the partition (COLD_START or WARM_START);
- stopping the partition (IDLE).

The detection of deadline violations should be performed right after the clock tick update, and before invoking the process scheduler. To optimize this process, only the earliest deadline (in principle) is checked; following deadlines may subsequently verified until one has not been missed. This can be computationally optimized with the help of an appropriate data structure with the deadlines in ascending order, allowing for $\mathcal{O}(1)$ retrieval of the earliest deadline. This is extremely relevant given deadline verification is performed inside the system clock interrupt service routine. Furthermore, this methodology is optimal with respect to deadline violation detection latency.

Regardless of the data structure, this information about processes statuses and deadlines is maintained in such a way that it is conveniently kept updated by the relevant APEX primitives. The APEX services:

- START, which makes a process become able to be executed, by initializing all its attributes, resetting the runtime stack, and placing it in the ready state;
- DELAYED_START, which makes the same initializations as START, but places the process in the waiting state (the process will become ready when the requested delay is expired);
- PERIODIC_WAIT, which suspends the execution of the requesting (periodic) process until the next release point[2];
- REPLENISH, through which a process requests its deadline time to be postponed, and;
- SET_PARTITION_MODE, which corresponds to requesting a partition shutdown or restart;

[2]A release point of a process is defined in general as the instant the process becomes ready for execution. For a periodic process the consecutive release points will be separated by the respective period.

will need to insert or update the due processes' deadlines, while:

- STOP, which renders a given process ineligible for process resources (it is placed in the dormant state) and;
- STOP_SELF, through which a process places itself in the dormant state;

need to remove the due processes' deadline information from the control data structures.

The AIR PAL component provides private interfaces for these APEX services to register/update and unregister deadlines [17]. The appropriate data structures containing this information will be kept at each partition's AIR PAL; this is the most logical implementation, from the engineering, integrity and spatial segregation points of view. An example of how the APEX and the AIR PAL for one given partition integrate to provide this functionality is provided in Figure 4.7.

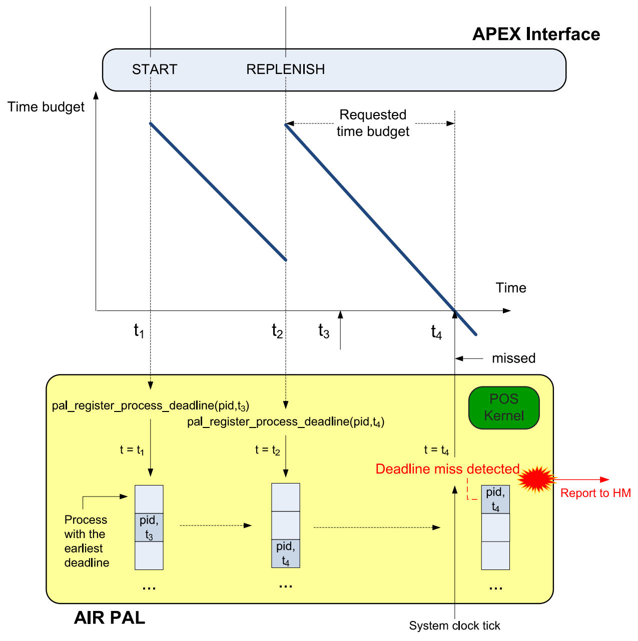


Figure 4.7: Integration of the APEX Interface and the AIR PAL to provide process deadline violation detection and reporting

When a process is started, via the START APEX service, its deadline time is set to instant t_3 (obtained by adding the process's time capacity to the current instant), and this value is registered via the AIR PAL-provided interface. Upon a replenishment request (REPLENISH service), a new deadline time, t_4, is calculated (by adding the requested budget time to the current instant). The interface provided by AIR PAL to register a process deadline is again called, to update the information for this process; if necessary, the node containing this information will have to be moved to keep the structure sorted by ascending deadline time order.

When instant t_4 is reached without the process having finished its execution, a deadline miss has occurred, which is detected and should be reported to the health monitoring mechanisms through appropriate private interfaces [17].

Figure 4.8 illustrates the further modification to the surrogate clock tick announcement routine provided by the AIR PAL, so as to verify the earliest deadline(s) and report any violations to the health monitoring.

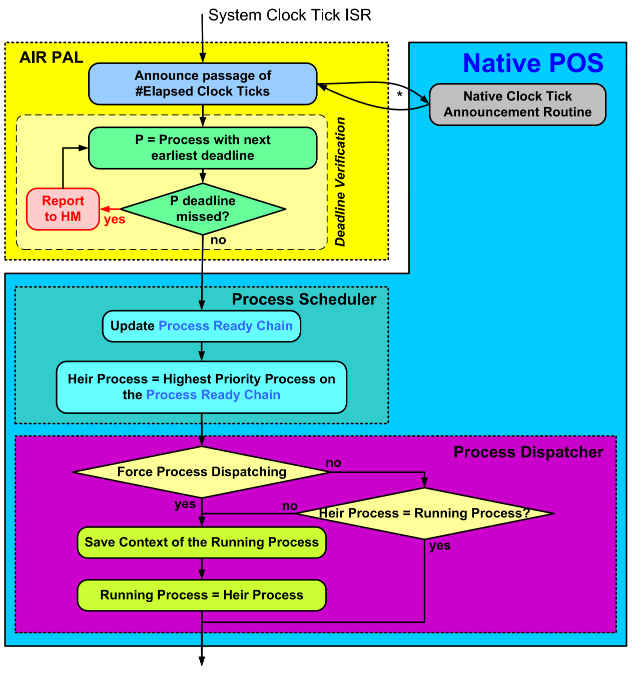

System Clock Tick ISR
AIR PAL
Announce passage of #Elapsed Clock Ticks
Native POS
*
Native Clock Tick Announcement Routine
P = Process with next earliest deadline
Deadline Verification
Report to HM
yes
P deadline missed?
no
Process Scheduler
Update Process Ready Chain
Heir Process = Highest Priority Process on the Process Ready Chain
Process Dispatcher
Force Process Dispatching
no
yes
no
Heir Process = Running Process?
yes
Save Context of the Running Process
Running Process = Heir Process

Figure 4.8: Modifications on the surrogate clock tick announcement routine to accommodate deadline verification features

Design and engineering issues

As described, to keep the computational complexity of the process deadline violation monitoring to a minimum, the information concerning process deadlines is kept, at each partition's AIR PAL component, ordered by deadline, and only the earliest deadline is verified by default; this deadline is retrieved in constant time ($\mathcal{O}(1)$). Only in the presence of deadline violation will more deadline be checked, in ascending order until reaching one that has not been violated.

Current AIR prototype engineering makes use of a linked list.

In the deadline verification process (which happens at every system clock tick during the partition's execution time window), a violation is detected, and after reporting the occurrence to HM the deadline is removed from the control structure. Since we already have a pointer to the node to be removed, the complexity of the deadline removal from the linked list will effectively be $\mathcal{O}(1)$ (as opposed to the generic $\mathcal{O}(n)$ complexity yielded by linked lists), requiring the manipulation of merely one pointer.

A point where the use of a self-balancing binary search tree [41] would theoretically outperform a linked list concern the act of inserting, removing or updating nodes, materialized in the register/unregister deadline interfaces provided to the APEX (cf. Listing A.4) — $\mathcal{O}(\log n)$ vs. $\mathcal{O}(n)$. Nevertheless, since these operations do not happen inside the system clock tick interrupt service routine (ISR), but rather on a partition's execution time window), and also the number of processes accounted for deadline verification will be typically small, such asymptotic advantage will not correlate to effective and/or significant profit, and certainly not compensate for the more critical downside to operations running during an ISR.

4.3 AIR space partitioning

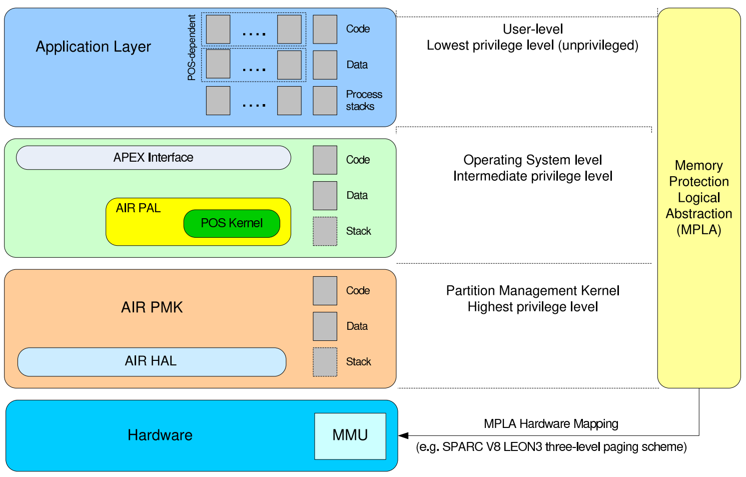
Also in the course of this thesis, albeit not directly related to the main focus of the present work, the spatial segregation mechanisms (cf. Section 3.2.2) supporting the AIR architecture's space partitioning capabilities were consolidated. A brief overview of such mechanisms [16] will now be given; these are described in further detail in [17].

As mentioned in Section 3.2.2, the approach adopted in AIR to support spatial partitioning was highly modular by design, featuring a high-level processor independent abstraction layer [15]. The diagram in Figure 4.9 schematizes this design, by bringing Figure 3.6 up to date with the newest definition of the AIR architecture.

The necessary mapping into MMU-specific mechanisms depends on the resources available on each processing platform foreseen for AIR applications. The most versatile mapping assumes the use of a memory segmentation model, such as it exists in the Intel IA-32 architecture, where a one-to-one mapping between the high-level abstract spatial partitioning description and low-level memory management descriptors is possible.

A one-to-one mapping is not possible if a paging translation model is being used in the MMU since a memory descriptor is required by page frame. This also implies some restrictions with respect to the number and size of partitions, as illustrated by the data inscribed in Figure 4.9.[3] An optimal design approach is

[3]It is worth mentioning that this thesis follows a notation in conformity with the IEC 60027-2 standard in respect to the usage of prefixes for binary multiples [42].



Memory Management Unit (MMU) Mapping Results

Processing Platform	*MMU Address Translation Model*	*Primary MMU Descriptors per Partition*	*Number of Partitions*	*Partition Size*
IA-32	segmentation	1	variable	variable
	paging	1	1024	4 MiB
SPARC V8	paging	1	256	16 MiB

Figure 4.9: AIR Spatial Partitioning and Operating System Integration

assumed, where the action of changing the status of a partition (active/inactive) requires no more than the update of a single primary MMU descriptor per partition. The data inscribed in Figure 4.9, for IA-32 and SPARC V8 RISC processing architectures, is in conformity with the requirements found in typical avionics and aerospace applications in respect to number and size of partitions.

The provision of these mapping functions is under the scope of overall partition management as provided by the APEX layer and by some specific AIR PMK components. For example, the mapping into processor specific descriptors needs to be updated in run-time when a partition switch occurs. This has to be coordinated by AIR PMK specific components, in this case by the AIR PMK Partition Dispatcher.

4.4 Summary

This chapter described evolutions made to the AIR architecture in the course of this thesis' work.

The contribution with most impact on the definition of the AIR architecture is a new component, the AIR POS Adaptation Layer (PAL), which stemmed from the initial motivation and lessons learned from the preliminary studies of this thesis' approach to the integration of generic operating systems in AIR. An appropriate instance of the AIR PAL wraps each partition's operating system, encapsulating its specifics; this allows other architectural components (with emphasis on the AIR PMK) to be more POS-independent and less prone to modifications, thus benefiting possible certification initiatives on this component. The AIR PAL also enables a more streamlined development process through separation of concerns.

Further contributions made to the AIR architecture concerned the improvement of timeliness enhancement mechanisms. The first one was mode-based schedules, an optional service of the ARINC 653 specification which allows the static definition of multiple partition scheduling tables; authorized partitions are able to switch between these different schedules, either to adapt the functioning of the system to different stages of the mission (e.g., airborne, landing, or exploration) or to implement fault tolerance mechanisms.

The other timeliness-related mechanism introduced was the monitoring and reporting of deadline violations by ARINC 653 hard real-time processes[4]; this mechanism was hooked in the sequence of the clock tick interrupt service routine (ISR), in way that efficiently compromises between responsiveness (required in the context of an ISR) and the promptness of deadline miss detection (violations are detected, in the worst case, at the beginning of the respective partition's next execution time window). Each partition's AIR PAL provides the mechanism and data structures for the violation of deadlines by the respective processes.

Further enhancements were also achieved, related to spatial segregation mechanisms. These mechanisms were consolidated, by establishing a mapping between the identification of partitions in the system and the first level of the three-level page-based memory management present on the SPARC LEON processor core [17].

[4] An ARINC 653 process is that which follows the ARINC 653 definition of a process, utilizing the service interface specified in [3]. Although that is not the primary goal of these deadline violation monitoring mechanisms, partitions with generic POSs can also utilize them to measure how their non-real-time or soft real-time tasks are performing. This requires applications to be modified to accommodate this, or the addition of a monitoring process.

Chapter 5

Integration of generic operating systems

This chapter approaches the problem of integrating generic operating systems as partition operating systems (POS) in the AIR architecture, using (embedded) Linux as a case study. The development of an embedded variant of a Linux-based operating system is documented, along with the analysis of its viability as an AIR POS. Then, a solution for guaranteeing that this integration does not compromise the already established strong properties of the AIR architecture is proposed; this solution is based on the paravirtualization interface present in the Linux kernel, paravirt-ops.

5.1 Relevance of the problem

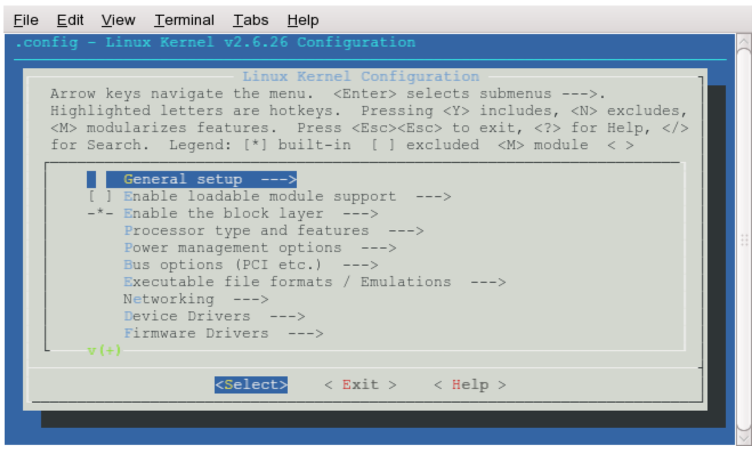
Porting applications to one of the RTOS one might be using (RTEMS [31], eCos [39, 40], VxWorks [43], etc.) can be a complicated task, and definitely not an error-free one [9, 10]. Furthermore, certain hardware interfaces may be necessary that are not supported by the given RTOS. This also applies to the aerospace applications that the AIR architecture targets. An example is a space probe for planetary observation, within which a hardware interface with a camera is needed, and whose pictures need to go through some post-processing by a widely available application that has not been ported to the RTOS.

To address this portability issue, we are evaluating the approach of having one partition of such a system run a general-purpose operating system based on the Linux kernel, for which community efforts continuously develop applications and device drivers. In the AIR architecture, such soft real-time and/or non-real-time applications using the standard Linux interface always receive a guaranteed (albeit shared) execution time window. Such guarantee is not provided by earlier approaches combining real-time and Linux operation in the same execution platform [12, 13, 14].

In this scenario, existent applications for GNU/Linux can be used or created without the further effort of having to port them to a particular RTOS and/or programming interface, like the APEX [3]. Another significant advantage is that the benefits of scripting languages widely used in the Linux world can be brought into scene, something which would otherwise depend on a port of the interpreter to a particular RTOS.

5.2 Linux state of the art

Linux is an open source operating system kernel available free of charge and maintained by developers from all the world. The source code is accessible for everyone and people are encouraged to contribute with their own code. For this reason, the Linux kernel is extremely portable between computer architectures and supports a massive variety of hardware devices and device drivers. And there's always space for more. An increased modularity along with a visual kernel configuration tool (illustrated in Figure 5.1) allow one to easily select the smallest set of features required for each system, avoiding unnecessary code. For systems with limited resources, or for very specific applications such as those found in aerospace, this may be very important. Additionally, the added support for a wider variety of hardware devices, computer architectures and the improved build tools help enhance the pace of development of the kernel itself. This flexibility makes Linux, and specially Linux 2.6, a good choice for embedded systems design, and for the provision of (soft) real-time guarantees.



```
.config - Linux Kernel v2.6.26 Configuration

                    Linux Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

          General setup  --->
      [ ] Enable loadable module support  --->
      -*- Enable the block layer  --->
          Processor type and features  --->
          Power management options  --->
          Bus options (PCI etc.)  --->
          Executable file formats / Emulations  --->
          Networking  --->
          Device Drivers  --->
          Firmware Drivers  --->
      v(+)

              <Select>    < Exit >    < Help >
```

Figure 5.1: Linux kernel configuration tool

5.2.1 Process scheduling

Linux kernel 2.6 allows the preemption of kernel tasks, i.e. user applications are no longer locked until the end of all pending system calls before they can continue executing. This significantly reduces the latency of user applications and increases the overall system responsiveness.

The original Linux 2.6 $\mathcal{O}(1)$ scheduler [44] was further improved into the *Completely Fair Scheduler (CFS)*, which wields $\mathcal{O}(1)$ complexity for choosing a task and $\mathcal{O}(\log n)$ for rescheduling a task after it has executed and becomes ready again. This is accomplished by the substitution of runqueues for a red–black tree with an auxiliary pointer to the leftmost element. The CFS also uses nanosecond granularity accounting (provided by the introduction of *high-resolution timers*), abandoning the notion of time slices and the need for specific process interactivity heuristics, while still allowing to tune the scheduler to cope with different workload patterns [45].

5.2.2 Input/output (I/O) scheduling

Another novelty introduced in the 2.6 line of the Linux kernel was the *Completely Fair Queueing (CFQ) I/O scheduler*, which applies concepts from network scheduling to disk scheduling: it maintains I/O queues per process, and attempts to distribute the available I/O bandwidth equally among all I/O requests. CFQ presents similar bandwidth results than the previously used Anticipatory Scheduler (AS), but lower latency, and has replaced AS as the default in the Linux kernel from version 2.6.18 onwards (although Red Hat, two years earlier, had included in RHEL 4 a 2.6.9 kernel configured with CFQ as default). While this is suitable for suitable for soft responsiveness requirements, real-time systems may further benefit from another I/O scheduler available in the Linux kernel: the *Deadline I/O scheduler*, which aims to guarantee that an I/O request begins to be served at a specified time. The Deadline I/O scheduler performs inferiorly than CFQ in balancing transactions across multiple drives or file systems, which is, by principle, a less common to encounter in embedded systems [46].

5.2.3 Real-time capabilities

Versions starting with 2.6.16 also saw components of the PREEMPT_RT patchset being merged into the kernel mainline. This includes userspace priority inheritance and high-resolution timers [47]. Remaining features, towards the goal of full kernel preemptibility and real-time capabilities in the Linux kernel, are still available as patches, there being a wide agreement among the Linux community that these will soon be merged into the kernel mainline [48].

Although the AIR architecture targets hard real-time systems, the study of integrating Linux in AIR did not take direct advantage or measure the profits from the increasingly near real-time capabilities of Linux. The reason for this lies in that the Linux partition will run non-critical tasks, and the hard real-time guarantees of the respective remaining partitions in the system will be guaranteed by the architecture's strong temporal segregation properties (independently of the internal real-time capabilities of the Linux partition).

5.3 Embedded Linux

There are a few Linux distributions available for embedded systems. However, some are commercial or targeted at a specific type of device, and others simply have too much unnecessary features or already have their own kernel modifications. Having a Linux kernel built from a standard, unpatched source tree, exactly as distributed by the developers, is extremely important. The absence of customized patches ensures easier upgradability and less compatibility issues between different versions.

Therefore, we analyze how to build a specific version of a Linux-based operating system targeted for embedded and aerospace systems and applications. The main vectors for achieving an effective balance between functionality and available resources are: 1) configuration of the Linux kernel; 2) inclusion of functional features; 3) use of a smaller system library; and 4) provision of the standard Unix command interface in a more resource-efficient way. However, instead of designing a solution totally from scratch, we follow a design-by-reuse approach, and use as much open source and widely available tools as possible; this philosophy is highly encouraged by ESA for its projects [27].

The foundations of such process have been addressed in the literature [49, 50]. Next, we describe its application, assess its effectiveness, and discuss its relevance to embedded and aerospace systems and applications.

5.3.1 Configuring the Linux kernel

The first step to produce a small kernel image exploits the configurability of the Linux kernel. The configuration of the Linux kernel is performed via a menu-driven graphical interface. In a first approach, superfluous features and device drivers were removed.

The diagram in Figure 5.2 illustrates the overall size difference between the Linux kernel image included in a generic distribution, and the image produced for this embedded Linux solution.

Generic	Embedded
2150 KiB	830 KiB

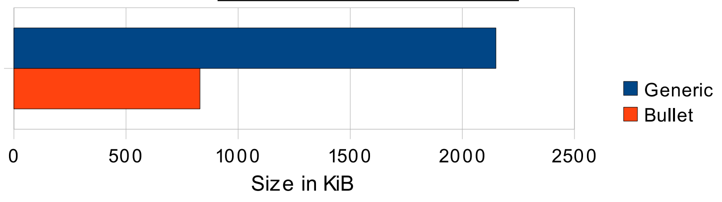


Figure 5.2: Size comparison between a kernel in a generic Linux distribution and the embedded Linux kernel

Two specific issues are worth mentioning. One is that, besides the kernel image, a standard Linux distribution ships a set of loadable kernel modules that can amount to 50 MiB, which were not accounted for in Figure 5.2, to make the comparison fairer. The other one is that the size gain illustrated in Figure 5.2 is both a combination of feature selection and using only built-in features; this gain will now be analyzed in a more fine-grained fashion.

5.3.2 Building in functional features

The reason why no items were included as modules is that in an embedded solution they must be always present in memory. Therefore, the design choice was to build in such functionalities into the kernel.

Figure 5.3 highlights the exact gain obtainable by removing the loadable module support and building features in into the kernel, instead of providing them through modules (with no condition differences otherwise). The data presented were obtained by adding, to the optimized Linux kernel, some extra functionalities (USB, Ethernet, WLAN, TCP/IP networking, PCMCIA, and Ext3 filesystem support) and the device drivers thereto associated. Some of these functions may not be present in the real systems for which this solution is targeted. They were here included for comparative measurement purposes only.

While adding features and device drivers through a built-in approach only results on an enlarged kernel image, the modular approach aggravates the total size in two fronts. On the one hand, the kernel image is enlarged to include the built-in support for loadable kernel modules. On the other hand, the kernel image must be accompanied by a set of several modules, which bear a noticeable overhead over the alternative of building those modules' features in into the kernel image.

	Image			**Modules**	**Total**
	Embedded kernel	***Module support***	***Built-in features***		
Modular	830 KiB	225 KiB	—	1568 KiB	2623 KiB
Built-in	830 KiB	—	644 KiB	—	1474 KiB

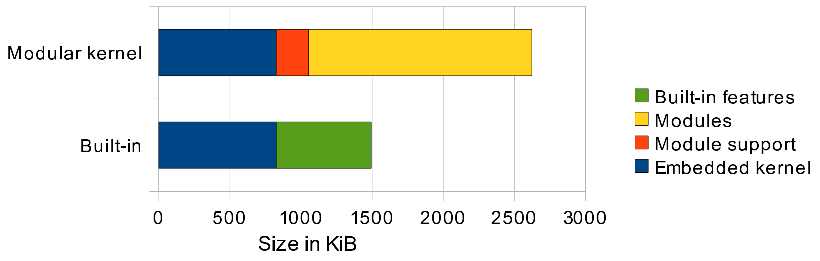


Figure 5.3: Size comparison between the embedded Linux kernel modular and built-in approaches to the inclusion of the same features

5.3.3 Small system library

One of the most important components of a Unix-like system is the *system library*. The system library provides application programmers a comprehensive set of services.

The most used system library is the *GNU C library (glibc)*.[1] This library is targeted for generic systems, exhibiting excessive functionality (for embedded systems), a non-optimized implementation, and a large object size.

An alternative design option is *uClibc*,[2] a C library specially developed for embedded systems. It features almost all GNU libc functionality, while exhibiting a small object size appropriate for systems with low memory resources. The uClibc developers have accomplished this by reimplementing it with size optimizations in mind, and by modularizing some functionalities, allowing the configuration of the uClibc library and its adaptation to the requirements of the target system.

There are alternative small footprint C libraries available, such as *newlib*,[3] and *diet libc*.[4] uClibc was chosen for its maturity and for how well other tools used (BusyBox, Buildroot) integrate with it. Nevertheless, future experiments might compare this with newlib, which has become an increasingly interesting alternative — it is, for instance, the C library included in RTEMS [31].

The use of uClibc allowed the present embedded Linux solution to keep a small size, when comparing against the use of a standard GNU libc. Figure 5.4 illustrates the immediate advantages brought by uClibc, showing the sizes taken up by both the GNU C library and uClibc.

[1]http://www.gnu.org/software/libc/
[2]http://www.uclibc.org/
[3]http://sourceware.org/newlib/
[4]http://www.fefe.de/dietlibc/

glibc	**uClibc**
2474 KiB	368 KiB

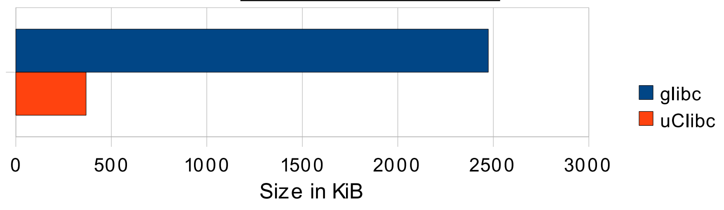


Figure 5.4: Size comparison between the GNU C Library (glibc) and the uClibc

5.3.4 Linux utilities and tools

A complete Linux-based operating system needs some well-known command-line utilities and tools. Even using shared libraries, standard GNU tools can use a lot of space, which is a real problem when dealing with resources shortage. To efficiently provide this functionality, we use *BusyBox*,[5] which is a set of those utilities and tools bundled together with a shell in a single executable. This approach alone reduces memory size requirements. Furthermore, the developers of BusyBox have rewritten these tools to be smaller than their original counterparts. This was accomplished by code optimizations and by the absence of some of the features, although maintaining the most important functions. Discarding unneeded sections from intermediate object files before generating the BusyBox executable allows a slight additional size gain.

BusyBox was chosen also because it is highly modular and configurable. It provides a wide array of commands (e.g. core utilities like dd, network utilities like ifconfig, or editors like sed) that can be included at this stage, some of which can be fine tuned as to only include a part of the available features.

Figure 5.5 shows the difference in size between a set of tools chosen for this embedded Linux (consisting, mainly, of core utilities), provided as both standalone executables and as only one BusyBox executable. The technical difference between the *BusyBox (unstripped)* and *BusyBox* executables is that the latter was produced by discarding unneeded sections from the former; this process is automatically performed when compiling BusyBox.

5.3.5 Shell

BusyBox provides a few shell options, the most traditional of which is the Almquist Shell (*ash*). Although compatible with the Bourne shell and suitable for low memory systems, *ash* lacks some extras provided by other shells like the ubiquitous Bourne Again Shell (*bash*).

[5] `http://www.busybox.net/`

Standard	BusyBox (unstripped)	BusyBox
1932 KiB	440 KiB	363 KiB

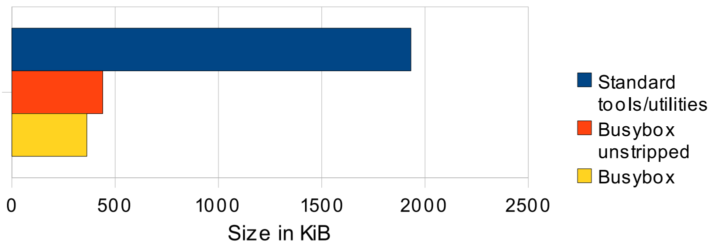


Figure 5.5: Size comparison between a set of GNU utilities and tools provided both as separate executables and as a single BusyBox executable (both stripped and unstripped of unnecessary symbols)

When the use of scripting is needing, one has to evaluate if the functionality provided by *ash* is appropriate. Otherwise, a more appropriate shell can be included, as a standalone executable. This can be automated during the building process (cf. Section 5.3.7, ahead).

5.3.6 Interpreted/scripting languages

The previous design steps of an embedded Linux solution leave out the support for interpreted/scripting languages. The support for interpreted/scripting languages is extremely interesting for a wide set of applications, including some of those in the aerospace domain.

BusyBox does not support any of these interpreters (save for the aforementioned shells), so support must be added as standalone executables. Once again, this can be automated during the building process (Section 5.3.7). Currently, the available packages are: lua, microperl (Perl without OS-specific functions), python, ruby, tcl, and php.

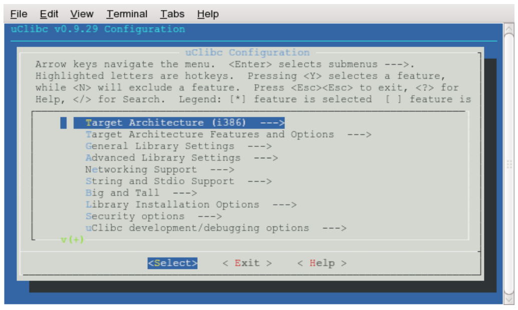
5.3.7 Building process

Buildroot[6] is a tool suite that makes it easy to generate a cross-compilation toolchain and other resources for the target Linux system using the uClibc C library. Buildroot is specially appropriate for embedded systems engineering, being used to facilitate the configuration and build process of the uClibc system library and the BusyBox toolset. It configures builds, and prepares the cross-compiler environment for the later build of the system library and toolset. This cross-compiling environment is necessary because the target architecture may be different from the architecture of the build system.

[6] http://buildroot.uclibc.org/

In this specific case, the kernel was compiled from unpatched sources with a specific configuration for the existing devices and interfaces of the prototype systems (Intel IA-32-based, Ethernet network, and usually no hard-disk drive). The system library and toolset were also configured to be as small as possible, while maintaining all the important functionalities.

The presented embedded Linux solution was built with Linux kernel 2.6.26, uClibc 0.9.29 and BusyBox 1.11.13. It is extremely customizable, inheriting its main components' flexibility and modularity. Linux kernel, uClibc and BusyBox configuration files are generated using the respective visual configuration tools (illustrated in Figures 5.1, 5.6(a), and 5.6(b)), and integrated into the Buildroot source tree. Buildroot is further fine tuned through the use of a similar configuration tool (Figure 5.6(c)).



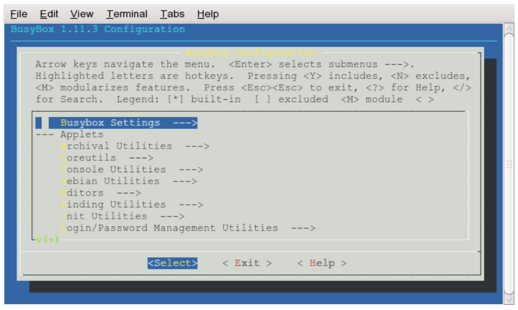
```
uClibc v0.9.29 Configuration

                         uClibc Configuration
   Arrow keys navigate the menu.  <Enter> selects submenus --->.
   Highlighted letters are hotkeys.  Pressing <Y> selectes a feature,
   while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
   Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is

            Target Architecture (i386)  --->
            Target Architecture Features and Options  --->
            General Library Settings  --->
            Advanced Library Settings  --->
            Networking Support  --->
            String and Stdio Support  --->
            Big and Tall  --->
            Library Installation Options  --->
            Security options  --->
            uClibc development/debugging options  --->
        v(+)

                  <Select>    < Exit >    < Help >
```

(a) uClibc



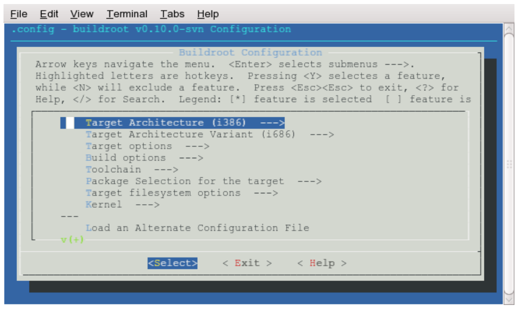
```
BusyBox 1.11.3 Configuration

                         Busybox Configuration
   Arrow keys navigate the menu.  <Enter> selects submenus --->.
   Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
   <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
   for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

        Busybox Settings  --->
   --- Applets
        Archival Utilities  --->
        Coreutils  --->
        Console Utilities  --->
        Debian Utilities  --->
        Editors  --->
        Finding Utilities  --->
        Init Utilities  --->
        Login/Password Management Utilities  --->
   v(+)

                  <Select>    < Exit >    < Help >
```

(b) BusyBox



```
.config - buildroot v0.10.0-svn Configuration

                       Buildroot Configuration
   Arrow keys navigate the menu.  <Enter> selects submenus --->.
   Highlighted letters are hotkeys.  Pressing <Y> selectes a feature,
   while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
   Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is

            Target Architecture (i386)  --->
            Target Architecture Variant (i686)  --->
            Target options  --->
            Build options  --->
            Toolchain  --->
            Package Selection for the target  --->
            Target filesystem options  --->
            Kernel  --->
        ---
            Load an Alternate Configuration File
        v(+)

                  <Select>    < Exit >    < Help >
```

(c) Buildroot

Figure 5.6: Visual configuration tools exploited in the embedded Linux build process

By putting together a specially configured Linux kernel 2.6 for embedded system prototyping, a restricted uClibc system library (e.g. excluding large file support), and a selected set of system tools (including the Almquist Shell, several core utilities, a few archival utilities, and no network utilities or Ext2 filesystem-related programs), it became possible to build an entire Linux operating system that can fit in as little as 1.5 MiB. This does not include any additional shell or language interpreter options as standalone executables.

5.3.8 Overall results analysis

The size gain of embedded Linux can be analyzed by comparing each of its components individually with the equivalent in a desktop distribution. Typically, a desktop distribution is built with a standard or lightly patched kernel compiled with a modular approach; a modular Linux Kernel is composed of an image plus a set of files that correspond to different modules. Modules are loaded into memory only if considered necessary by the system or the user. A typical modular kernel has an image size slightly above 2 MiB and a set of modules with about 50 MiB. The system library is a fully featured GNU libc 2.X (libc 6) along with many other smaller less generic libraries. The system tools in a typical desktop distribution are compiled against the GNU C library and occupy a big slice of storage space. Globally, Figure 5.7 summarises the analysis of size in two distinct situations: a generic Linux distribution and the analyzed embedded Linux solution. For comparison sake, we only present the size occupied, in a standard Linux distribution, by the same system utilities/tools and libraries included in the embedded variant.

Kernel		**System library**		**System library**		**Total**	
Generic	***Embedded***	***glibc***	***uClibc***	***Generic***	***BusyBox***	***Generic***	***Embedded***
2150 KiB	830 KiB	2474 KiB	368 KiB	1932 KiB	363 KiB	6556 KiB[a]	1561 KiB

[a]Plus a set of modules amounting to 50 MiB

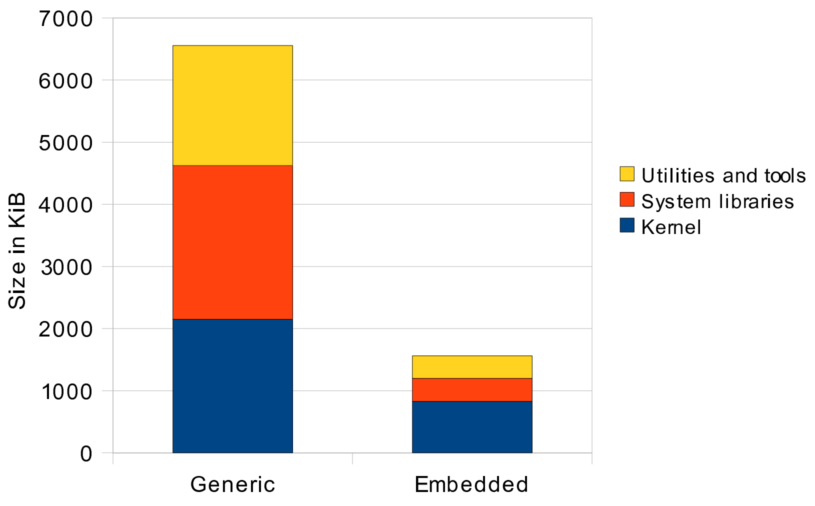


Figure 5.7: Overall size comparison between embedded Linux and a typical Linux distribution

The results obtained with this embedded Linux approach open room for its integration in the AIR architecture. The embedded solution provided allows for an implementation of Linux-based systems and applications which will always be entirely present in physical memory. No virtual memory mechanisms are required, meaning no particular memory protection scheme is needed for compliance to the ARINC 653 specification and integration in the AIR architecture.

5.4 Integration in the AIR architecture

The integration of Linux in the AIR architecture poses issues regarding the integration of Linux as partition OS focus on guaranteeing that it does not contaminate the robust temporal and spatial partitioning of the AIR architecture. Temporal partitioning is ensured, as standard, by the cyclic fixed scheduling of partitions, provided that the Linux partition can not disable or divert interrupts at the hardware (processor) level. We will want the Linux kernel to be notified of clock ticks, like other partition operating systems, only when its partition is active. Thus, interrupts will be totally controlled and handled by the AIR PMK, bypassing the Linux interrupt infrastructure [51].

To guarantee this, and since most processor architectures are not fully virtualizable (i.e., not all sensitive instructions are also privileged instructions), we can not merely run Linux in an unprivileged mode (usermode) and rely on having sensitive instructions generate a trap [52, 53]. A good candidate to solve this issue is the employment of paravirtualization [54].

5.4.1 Paravirtualization in the Linux kernel

The paravirt-ops paravirtualization interface, which enables multiple hypervisors to hook directly into the Linux kernel, has been merged into the main Linux kernel starting with version 2.6.21, along with the support for VMWare's Virtual Machine Interface (VMI). VMI is the open specification of an interface for the paravirtualized guest OS kernel to communicate with the hypervisor [55], which takes advantage of hooks onto the paravirt-ops interface. Many popular Linux distributions shipping with Linux 2.6.21 have the paravirt-ops and VMI configuration options enabled; this means that the same kernel will run both on native hardware and on top of a VMI-enabled hypervisor without requiring recompilation (with negligible performance overhead when running on native hardware [56]).

Figure 5.8 illustrates the process in which a VMI-enabled Linux kernel is booted, and either runs natively or on top of a hypervisor. Early during the boot process, the VMI initialization code probes for a ROM module through which the hypervisor's VMI layer is to be published to the paravirtualized operating system. If such a module is found, the VMI initialization code dynamically patches the kernel, so as to inject the necessary calls to the hypervisor's VMI layer; if not, the kernel continues to run as normal, natively on top of the hardware [55].

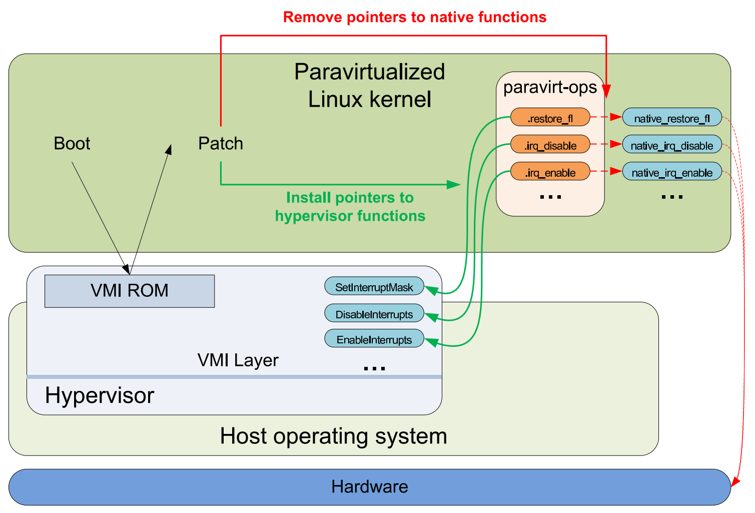


Figure 5.8: Boot process of a paravirtualized Linux kernel on top of a paravirt-ops/VMI-compliant hypervisor

5.4.2 AIR Linux partition: AIR PAL design and integration

When transposing this to the reality of the AIR architecture, the AIR PAL will provide the relevant functions of the VMI layer to the partition operating system, interacting with AIR PMK when required, as illustrated in Figure 5.9. Examples of the VMI functions to be provided by the AIR PAL include virtualization of: *(i)* interrupt management; *(ii)* input/output (I/O) calls; *(iii)* memory and I/O space protection mechanisms; *(iv)* privilege level management.

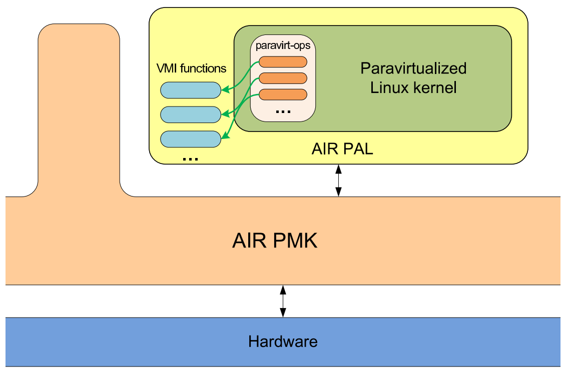


Figure 5.9: Concepts of paravirtualization in the AIR architecture

Further integration issues have already been identified. One of them concerns the implementation, at the AIR PAL level, of a surrogate clock tick announcement routine (cf. Section 4.1.2) that maps onto the complex timekeeping architecture of Linux [51].

Providing the access to interpartition communication mechanisms to the Linux partition is also a feature that needs to be addressed. On a first and direct approach, applications in the Linux partition can be adapted to use the APEX interface primitives to send and receive messages to/from queuing ports, and/or write and read messages to/from sampling ports. This does, nevertheless, obligate Linux application developers to divert from the usual programming model, thus slightly undermining the seminal motivation for integrating generic operating systems onto the AIR architecture. A more interesting approach would consist of providing pseudodevices (e.g. `/dev/apexqp1` for a queuing port, or `/dev/apexsp2` for a sampling port), through which Linux processes could access interpartition communication facilities using more traditional primitives (`open`, `read`, `write`, `send`, `recv`).

The combination of Linux's process-based memory protection mechanisms with the AIR partition-based spatial segregation design must also be addressed carefully in future work. More specifically, we must ensure that, when a Linux process tries to access a memory zone that, not only is outside its address space, but is also outside *its partition's* addressing space, the AIR Health Monitoring mechanisms are adequately notified of this error.

Finally, there is further investigation and engineering work to do related to the generation and build process of the Linux partition system image. Current prototyping activities using RTEMS [31] comprise having each POS in the form of an object file; at integration time, the POS object files are linked together with the core software layer components object files to produce the final system image (cf. Figure 4.5). Upon the engineering of an AIR prototype with a Linux partition, either the Linux system image production process or the AIR build process will have to be conveniently adjusted to allow the production of a final single executable file.

These design considerations apply both to Intel IA-32 and SPARC V8 processor architectures.

5.4.3 AIR application platforms

The space applications to which the AIR technology is to be applied typically employ SPARC V8 RISC processors, like LEON 2 and LEON 3, so the concepts of paravirt-ops and VMI, which are Intel IA-32 and Intel 64-centric by design, have to be transposed to the reality of this architecture. The current state of the art is nevertheless interesting for proof of concept prototyping purposes, and to apply to ground-segment applications, where the Intel architectures are present. As of Linux kernel 2.6.30, paravirt-ops and VMI support is implemented for both Intel IA-32 and Intel 64 architectures.

5.5 Summary

This chapter describes the problem of integrating generic operating systems on the AIR architecture, and the developments achieved on its solution using Linux as a case study. An embedded flavor of Linux is analyzed in terms of its viability for this purpose, with its genesis described step by step and in detail; the main pillars of the development of such a solution are the configuration of the Linux kernel, the utilization of a reduced C library, and the efficient provision of common utilities and tools as an optimized single executable file.

The paravirtualization interface provided by the Linux kernel (paravirt-ops) is explained and proposed as a solution for the safety issues inherent to the integration of Linux as a POS; namely, one must guarantee that interrupts are completely are maintained. Further design and engineering issues — regarding timekeeping, interpartition communication, memory protection and system integration — have been identified for future developments.

Chapter 6

Conclusion

This thesis addressed the problem of integrating generic operating systems onto AIR, an ARINC 653-based partitioned architecture for aerospace applications featuring strong temporal and spatial segregation, and allowing the use of different partition operating systems (POS). The architecture is composed of multiple components, of which this thesis focuses most on the AIR Partition Management Kernel (PMK), which is responsible, for instance, for partition scheduling and dispatching, and concentrates most mechanisms to guarantee temporal and spatial segregation.

The contributions of this work encompass improvements made to the AIR architecture, which were described throughout this document, and comprise:

(i) a new component, the AIR POS Adaptation Layer (PAL), which allows for a stable POS-independent AIR PMK, a homogenous process of integrating new POSs (both real-time and non-real-time operating systems) and enables better a better process development workflow through separation of concerns;

(ii) advanced timeliness adaptation and monitoring mechanisms, like mode-based schedules (allowing to change, in execution time, between multiple predefined partition scheduling tables, so as to adapt system functioning to different phases of operation) and process deadline violation monitoring;

(iii) the study of Linux as a candidate for a generic non-real-time POS in AIR; this included the genesis and evaluation of an embedded flavor of Linux for this purpose, and the proposal of paravirtualization mechanisms to guarantee that the integration of Linux does not pose safety issues, by compromising temporal and/or spatial segregation.

The described embedded Linux yielded encouraging results as a POS candidate: a fully-operational embedded Linux version, complete with a system library and common utilities, could fit in as little as 1.5 MiB; this is very important in the context of a POS for AIR, since there will be multiple OS kernels coexisting in the same platform, and the typical absence of persistent storage (such as a hard disk drive) means that they will all reside permanently in memory.

6.1 Future work directions

The implementation of a complete AIR prototype demonstrator with the Linux kernel running in one of the partitions requires engineering work that was deemed incompatible with the duration of this thesis's project, and is subject to follow-up work. The next and ambitious step may include the approach to other generic non-real-time operating systems, such as Windows (in the form of the Windows Research Kernel [57]).

Other interesting issues pertaining to the AIR architecture still to be addressed include consolidating and extending temporal and spatial segregation support features, by including a structured approach integrating these concepts with the need to operate a computing platform with interfaces to the real world, in the form of sensors and actuators (e.g. satellite attitude control). From the perspective of spatial segregation, this implies extending memory protection mechanisms to the domain of each partition's use of input/output, possibly resorting to dedicated communication channels between partitions, or to the advanced notion of pseudopartition extensions. At the temporal segregation level, this might implicate consolidating the time execution windows assigned to each partition with a global event scheduling perspective.

Further developments already planned as expansions of this line of work revolve around problems related with scheduling, and schedulability analysis.

Multicore processors are paving their way into the realm of embedded systems [58]. The utilization of multicore architectures in IMA-related platforms has, though, not been addressed in detail. Real-time scheduling in multicore platforms has been approached in literature in some of its facets [59, 60, 61]; there are still though open questions [62]. Also, to the best of our knowledge, real-time scheduling in multicore platforms has not been associated with the IMA-inherent two- level scheduling scheme.

In this perspective, it is interesting to apply the existing concepts and any advances meanwhile obtained on schedulability analysis to platforms with multicore CPUs. This includes the profound analysis of the impact of parallelism — both intrapartition parallelism (i. e., between processes) and between partitions. The approach to intrapartition parallelism aims to understand why ARINC 653 may mandate that processes in a partition shall not be distributed among processors, and if the reasons also apply to distribution among cores in one processor.

Concerning parallelism between partitions, two approaches to the scheduling of the latter will be studied: *(i)* static (extending the system configuration mechanisms, to allow explicit definition of where parallelism between partitions will occur), and; *(ii)* semi-dynamic (extending the configuration mechanisms, to allow the expression of restrictions and dependencies that will guide the activity of a dynamic partition scheduler with support for parallelism between partitions). This evolution implies not only schedulability considerations, but also a great concern with overall robustness, safety and security.

Appendix A

Pseudocode snippets

A.1 Mode-based schedules

A.1.1 Data structures and global variables

```
/* Data structures */

typedef struct
{
  int number;
  int      last_tick;
  void *  tick_announce;
  Context context;
} Partition;

typedef struct
{
  int              tick;
  Partition*  partition;
} PreemptionPoint;

typedef struct
{
  int mtf;
  int initialSchedule; /* boolean ,true for one and only one schedule */
  PreemptionPoint* table;
  int numberPartitionPreemptionPoints;
  ScheduleChangeAction changeAction[NUMBER_PARTITIONS];
} ModuleSchedule;

typedef struct
{
  int lastSchedSwitch;
  int currentSched;
  int nextSched;
} ScheduleStatus; /* as per ARINC 653 Part 2 */

/* Global variables */
Partition*      _runningPartition;
Partition*      _heirPartition;

ModuleSchedule  schedules[NUMBER_MODULE_SCHEDULES];
ScheduleStatus  schedStatus;

Partition       partitions[NUMBER_PARTITIONS];
int             _table_iterator;


int             _ticks;
int             _elapsedTicks;
```

Listing A.1: Data structures and global variables common to the partition scheduler and dispatcher

A.1.2 Partition scheduler

```
_ticks++;

/* check if we are in the presence of a preemption point */
if( schedules[schedStatus.currentSched].table[_table_iterator].tick ==
    ((_ticks - schedStatus.lastSchedSwitch)
     % schedules[schedStatus.currentSched].mtf) ) {

  /* check if this is the end of MTF and a new schedule is pending */
  if ( ( ( _ticks - schedStatus.lastSchedSwitch ) %
        schedules[schedStatus.currentSched].mtf == 0 ) &&
        ( schedStatus.currentSched != schedStatus.nextSched ) ) {

    /* change to new schedule */
    schedStatus.currentSched = schedStatus.nextSched;

    /* update last schedule switch time */
    schedStatus.lastSchedSwitch = _ticks;

    /* change the heir partition */
    _heirPartition = schedules[schedStatus.currentSched].table[0].partition;

    /* increase table iterator cyclically */
    _table_iterator = (_table_iterator + 1) % schedules[schedStatus.currentSched].
        numberPartitionPreemptionPoints;

  } else {

    /* change the heir partition */
    _heirPartition = schedules[schedStatus.currentSched].table[_table_iterator].
        partition;

    /* increase table iterator cyclically */
    _table_iterator = (_table_iterator + 1) % schedules[schedStatus.currentSched].
        numberPartitionPreemptionPoints;
  }
}
```

Listing A.2: Partition scheduler featuring mode-based schedules

A.1.3 Partition dispatcher

```
if(_heirPartition == _runningPartition) {

    _elapsedTicks           = 1;

} else {

  /* Save Running Partition Context */
  _SaveContext(_runningPartition.context);

  _runningPartition->last_tick = _ticks - 1;

  /* Update the heir partition number of elapsed clock ticks */
  _elapsedTicks = _ticks - _heirPartition->last_tick;

  /* Select running partition */
  _runningPartition = _heirPartition;

  /* Restore Heir Partition Context */
  _RestoreContext(_heirPartition.context);

  /* Trigger pending change actions */

  /* Announce elapsed clock ticks to the POS */
  void (*tick)(int) = _heirPartition->tick_announce;
  (*tick)(_elapsedTicks);
}
```

Listing A.3: Partition dispatcher featuring mode-based schedules

A.2 Process deadline violation monitoring

```
typedef
  struct {
    PROCESS_ID_TYPE      process_id;
    SYSTEM_TIME_TYPE     deadline_time;
    void*  next;
  } PAL_DEADLINE_TYPE;

int pal_register_process_deadline( /*in*/ PROCESS_ID_TYPE process_id,
/*in*/ SYSTEM_TIME_TYPE deadline_time) {
  PAL_DEADLINE_TYPE* dp = NULL;
  PAL_DEADLINE_TYPE* modp = NULL;
  PAL_DEADLINE_TYPE* modprevp = NULL;

  /* search for a deadline already set for process process_id */
  for (dp = pal_deadlines; dp != NULL; dp = (PAL_DEADLINE_TYPE*)(dp->next)) {
    if (dp->process_id == process_id) {
      modp = dp;
      break;
    }
    modprevp = dp;
  }

  if (modp == NULL) {

    /* no deadline already set: insert */
    PAL_DEADLINE_TYPE* newd = /* obtain pointer to structure */;
    newd->process_id = PROCESS_ID;
    newd->deadline_time = DEADLINE_TIME;
    newd->next = NULL;

    /* empty list */
    if (pal_deadlines == NULL) {
      pal_deadlines = newd;
      return NO_ERROR;
    }

    PAL_DEADLINE_TYPE* insp = NULL;
    PAL_DEADLINE_TYPE* insprevp = NULL;

    /* search for correct place to insert new deadline */
    for (insp = pal_deadlines, insprevp = NULL;
      insprevp != NULL;
      insp = (PAL_DEADLINE_TYPE*)(insp->next)) {

      if (insp == NULL) {
        /* tail */
        insprevp->next = newd;
        return NO_ERROR;
      }

      if (insp->deadline_time >= DEADLINE_TIME) {
        newd->next = insp;
        insprevp->next = newd;
      }

      insprevp = insp;
    }

  } else {
    /* deadline already set: update and reposition */
    modp->deadline_time = DEADLINE_TIME;

    /* advance the node until it is before a node with a deadline greater or equal */
    while (modp->next != NULL &&
      ((PAL_DEADLINE_TYPE*)(modp->next))->deadline_time < modp->deadline_time) {
      modprevp->next = modp->next;
      modp->next = ((PAL_DEADLINE_TYPE*)modprevp->next)->next;
```

```
        ((PAL_DEADLINE_TYPE*)modprevp->next)->next = modp;
      }
      return NO_ERROR;
    }
}

int pal_unregister_process_deadline( /*in*/ PROCESS_ID_TYPE PROCESS_ID) {
  PAL_DEADLINE_TYPE* delp = NULL;
  PAL_DEADLINE_TYPE* delprevp = NULL;

  /* search for node */
  for (delp = pal_deadlines; delp != NULL; delp = (PAL_DEADLINE_TYPE*)(delp->next)) {
    if (delp->process_id == PROCESS_ID) {
      /* found: delete */
      if (delprevp == NULL) {
        /* deleting head (covers deleting only remaining node) */
        pal_deadlines = delp->next;
      } else {
        /* deleting middle/tail element */
        delprevp->next = delp->next;
      }
      free(delp);
      return NO_ERROR;
    }
    delprevp = delp;
  }

  /* if we reach this point, then process_id did not have a deadline
     to unregister */
  return INVALID_PARAM;
}

int pal_verify_deadlines() {
  PAL_DEADLINE_TYPE* deadp = pal_deadlines;
  int                 reported = 0;

  while (deadp != NULL) {
    if (deadp->deadline_time < pal_get_current_time()) {
      /* report deadline violation by deadp->process_id to HM */

      reported++;
      deadp = deadp->next;
    }
    break;
  }

  return reported;
}
```

Listing A.4: Deadline register, unregister and verification at the AIR PAL level

Abbreviations

AIR	ARINC 653 in Space RTOS
APEX	Application Executive
ARINC	Aeronautical Radio, INCorporated
CNES	Centre National d'Études Spatiales
CPU	Central Processor Unit
ESA	European Space Agency
GCC	GNU Compiler Collection
GDB	The GNU Debugger
GNU	GNU's Not Unix
HAL	Hardware Abstraction Layer
HM	Health Monitoring
IMA	Integrated Modular Avionics
ISR	Interrupt Service Routine
LRM	Line Replaceable Modules
LRU	Line Replaceable Unit
MMU	Memory Management Unit
MTF	Major Time Frame
OS	Operating System
PAL	POS Adaptation Layer
PMK	Partition Management Kernel
POS	Partition Operating System
PST	Partition Scheduling Table
RTAI	Real-Time Application Interface for Linux
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System

SIS	SPARC Instruction Simulator
SPARC	Scalable Processor ARChitecture
TSP	Time and Space Partitioning
VMI	Virtual Machine Interface
WCET	Worst Case Execution Time
WP	Work Package

Bibliography

[1] *Design Guidance for Integrated Modular Avionics*, Airlines Electronic Engineering Committee (AEEC), ARINC Specification 651, 1991.

[2] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," in *Proceedings of the 26th IEEE/AIAA Digital Avionics Systems Conference (DASC 2007)*, Dallas, TX, USA, Oct. 2007.

[3] *Avionics Application Software Standard Interface*, Airlines Electronic Engineering Committee (AEEC), ARINC Specification 653-2 Part 1 (Required Services), Mar. 2006.

[4] *Avionics Application Software Standard Interface*, Airlines Electronic Engineering Committee (AEEC), ARINC Specification 653 Part 2 (Extended Services), Draft 5, Aug. 2006.

[5] J.-L. Terraillon and K. Hjortnaes, "Technical note on on-board software," ESA, European Space Technology Harmonisation, Technical Dossier on Mapping, TOSE-2-DOS-1, Feb. 2003.

[6] TSP Working Group, "Time and space partitioning for space application," presented at the ESA Workshop on Avionics Data, Control and Software Systems (ADCSS), Noordwijk, The Netherlands, Oct. 2008.

[7] N. Diniz and J. Rufino, "ARINC 653 in space," in *Proceedings of the DASIA 2005 "DAta Systems In Aerospace" Conference*. Edinburgh, Scotland: EUROSPACE, Jun. 2005.

[8] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, "ARINC 653 interface in RTEMS," in *Proceedings of the DASIA 2007 "DAta Systems In Aerospace" Conference*. Naples, Italy: EUROSPACE, Jun. 2007.

[9] L. Kinnan, J. Wlad, and P. Rogers, "Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example," in *Proceedings of the 23rd IEEE/AIAA Digital Avionics Systems Conference (DASC 2004)*, vol. 2, Salt Lake City, UT, USA, Oct. 2004, pp. 10.B.1–10.1–8.

[10] L. Kinnan, "Application migration from Linux prototype to deployable IMA platform using ARINC 653 and Open GL," in *Proceedings of the 26th IEEE/AIAA Digital Avionics Systems Conference (DASC 2007)*, Dallas, TX, USA, Oct. 2007, pp. 6.C.2–1–6.C.2–5.

[11] J. Craveiro, J. Rufino, C. Almeida, R. Covelo, and P. Venda, "Embedded Linux in a partitioned architecture for aerospace applications," in *Proceedings of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2009)*, Rabat, Morocco, May 2009, pp. 132–138.

[12] V. Yodaiken and M. Barabanov, "A real-time Linux," in *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, USA, Jan. 1997.

[13] G. Racciu and P. Mantegazza, *RTAI 3.4 User Manual, rev. 0.3*, Oct. 2006.

[14] P. Braga, L. Henriques, B. Carvalho, P. Chevalley, and M. Zulianello, "xLuna - demonstrator on ESA Mars Rover," in *Proceedings of the DASIA 2008 "DAta Systems In Aerospace" Conference*, Palma de Majorca, Spain, May 2008.

[15] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor, "AIR Technology: a step towards ARINC 653 in space," in *Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference*. Istanbul, Turkey: EUROSPACE, May 2009.

[16] J. Craveiro, J. Rufino, T. Schoofs, and J. Windsor, "Flexible operating system integration in partitioned aerospace systems," in *Actas do INForum 2009, Simpósio de Informática*, Lisbon, Portugal, Sep. 2009, accepted for publication.

[17] J. Rufino and J. Craveiro, "AIR Design Consolidation, PMK – Partition Management Kernel," FCUL, AIR-II Deliverable WP 1.1, 2009, confidential document.

[18] J. Craveiro, J. Rufino, T. Schoofs, and J. Windsor, "Robustness, flexibility and separation of concerns in ARINC 653-based aerospace systems," AIR-II Technical Report RT-09-02, 2009.

[19] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, Oct. 1988.

[20] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[21] P. Veríssimo and L. Rodrigues, *Distributed Systems for System Architects*, 1st ed. Boston, MA, USA: Springer, Jan. 2001.

[22] R. Little, "Advanced avionics for military needs," *Computing and Control Engineering Journal*, vol. 2, no. 1, pp. 29–34, Jan. 1991.

[23] N. Audsley and A. Wellings, "Analysing APEX applications," in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS 1996)*, Washington, DC, USA, Dec. 1996, pp. 39–44.

[24] M. A. Sánchez-Puebla and J. Carretero, "A new approach for distributed computing in avionics systems," in *Proceedings of the 1st International Symposium on Information and Communication Technologies (ISICT 2003)*. Dublin, Ireland: Trinity College Dublin, 2003, pp. 579–584.

[25] E. Pascoal, "AMOBA — ARINC 653 simulator for modular space based applications," Department of Informatics, University of Lisbon, 2008, M.Sc. Thesis.

[26] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms and assurance," SRI International, California, USA, NASA Contractor Report CR-1999-209347, Jun. 1999.

[27] J. Miró, "Onboard software technology," presented at the 3rd Portuguese Space Forum, Lisbon, Portugal, Jun. 2009.

[28] K. Yaghmour, "Adaptive Domain Environment for Operating Systems," 2001, unpublished whitepaper.

[29] G. Zhang, L. Chen, and A. Yao, "Study and comparison of the RTHAL-based and ADEOS-based RTAI real-time solutions for Linux," in *Proceedings of the 1st International Multi-Symposiums on Computer and Computational Sciences (IMSCCS 2006)*, vol. 2. Hangzhou, Zhejiang, China: IEEE Computer Society, 2006, pp. 771–775.

[30] D. Probert, J. Bruno, and M. Karaorman, "SPACE: a new approach to operating system abstraction," in *Proceedings of the 1st International Workshop on Object Orientation in Operating Systems (IWOOOS 1991)*, Palo Alto, CA, USA, Oct. 1991, pp. 133–137.

[31] *RTEMS C Users Guide*, Edition 4.8, for RTEMS 4.8 ed., OAR - On-Line Applications Research Corporation, Feb. 2008.

[32] S. Goiffon and P. Gaufillet, "Linux: A multi-purpose executive support for civil avionics applications?" in *IFIP Congress Topical Sessions*, R. Jacquart, Ed. Toulouse, France: Kluwer, 2004, pp. 719–724.

[33] N. Peccia, "Software technology," presented at the 3rd Portuguese Space Forum, Lisbon, Portugal, Jun. 2009.

[34] M. Masmano, I. Ripoll, and A. Crespo, "XtratuM Hypervisor for LEON2: design and implementation overview," I. U. de Automática e Informática Industrial, Universidad Politécnica de Valencia, Tech. Rep., Jan. 2009.

[35] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. J. Metge, "XtratuM: an Open Source Hypervisor for TSP Embedded Systems in Aerospace," in *Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference*. Istanbul, Turkey: EUROSPACE, May 2009.

[36] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *Proceedings of the 27th IEEE/AIAA Digital Avionics Systems Conference (DASC 2008)*, St. Paul, MN, USA, Oct. 2008.

[37] IEEE, *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. New York, NY, USA: IEEE, 1996.

[38] E. Pascoal, J. Rufino, T. Schoofs, and J. Windsor, "AMOBA — ARINC 653 simulator for modular space based applications," in *Proceedings of the DASIA 2008 "DAta Systems In Aerospace" Conference*, Palma de Majorca, Spain, May 2008.

[39] A. Massa, *Embedded Software Development with eCos*. Prentice-Hall, 2002, iSBN 0130354732.

[40] M. Barr and A. Massa, *Programming Embedded Systems (with C and GNU Development Tools)*, 2nd ed. O'Reilly Media, Inc., Oct. 2006.

[41] E. B. Koffman and P. A. T. Wolfgang, *Objects, Abstraction, Data Structures and Design Using Java Version 5.0*. John Wiley and Sons, Inc., 2005.

[42] *IEC 60027-2: Letter symbols to be used in electrical technology – Part 2: telecommunications and electronics*, IEC Std., Aug. 2005.

[43] *VxWorks Kernel Programmer's Guide, 6.2*, Wind River Systems, Inc., 2005.

[44] J. Aas, "Understanding the Linux 2.6.8.1 CPU scheduler," Feb. 2005.

[45] D. Hart, J. Stultz, and T. Ts'o, "Real-time Linux in real time," *IBM SYSTEMS JOURNAL*, vol. 47, no. 2, p. 208, 2008.

[46] D. J. Shakshober, "Choosing an I/O scheduler for Red Hat Enterprise Linux 4 and the 2.6 kernel," *Red Hat Magazine*, no. 8, Jun. 2005. [Online]. Available: http://www.redhat.com/magazine/008jun05/features/schedulers/

[47] T. Gleixner, "The realtime preemption patch (PREEMPT_RT): concepts and mainline integration," presented at the 8th Real-Time Linux Workshop (RTLWS 2006), Lanzhou, Gansu, China, Oct. 2006.

[48] T. Schoofs, S. Santos, C. Tatibana, J. Anjos, J. Rufino, and J. Windsor, "An IMA development environment," in *Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference*. Istanbul, Turkey: EUROSPACE, May 2009.

[49] D. Abbott, *Linux for Embedded and Real-time Applications*, 2nd ed., ser. Embedded Technology. Newnes, 2006.

[50] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[51] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, Inc., Aug. 2008.

[52] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[53] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers, 2005.

[54] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA, Jun. 2002, pp. 195–209.

[55] Z. Amsden, D. Arai, D. Hecht, A. Holler, and P. Subrahmanyam, "VMI: An interface for paravirtualization," in *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, Jul. 2006, pp. 363–378.

[56] (2009, May) Native performance: paravirt vs non-paravirt kernel. VMware, Inc. [Online]. Available: http://www.vmware.com/interfaces/paravirtualization/performance.html

[57] M. Schöbel and A. Polze, "Kernel-mode scheduling server for CPU partitioning: a case study using the Windows Research Kernel," in *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC 2008)*. Fortaleza, Ceará, Brazil: ACM, 2008, pp. 1700–1704.

[58] J.-Y. Mignolet and R. Wuyts, "Embedded multiprocessor systems-on-chip programming," *IEEE Software*, vol. 26, no. 3, pp. 34–41, May/Jun. 2009.

[59] J. Anderson, J. Calandrino, and U. Devi, "Real-time scheduling on multicore platforms," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, San Jose, CA, USA, Apr. 2006, pp. 179–190.

[60] K. Lakshmanan, R. Rajkumar, and J. P. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS 2009)*, Dublin, Ireland, Jul. 2009, accepted for publication. [Online]. Available: http://www.contrib.andrew.cmu.edu/~klakshma/partfpps.pdf

[61] J. Carpenter, S. Funk, P. Holman, J. Anderson, and S. Baruah, *A categorization of real-time multiprocessor scheduling problems and algorithms*. Chapman & Hall/CRC, 2004.

[62] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson, "Soft real-time scheduling on performance asymmetric multicore platforms," in *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2007)*, Bellevue, WA, United States, Apr. 2007, pp. 101–112.