UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AMOBA - ARINC 653 SIMULATOR FOR MODULAR SPACE BASED APPLICATIONS

Edgar Manuel Cândido da Silva Pascoal

Mestrado em Engenharia Informática

2008

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

AMOBA - ARINC 653 SIMULATOR FOR MODULAR SPACE BASED APPLICATIONS

Edgar Manuel Cândido da Silva Pascoal

DISSERTAÇÃO

Projecto orientado pelo Prof. Dr José Manuel de Sousa de Matos Rufino
e co-orientado por Tobias Schoofs

Mestrado em Engenharia Informática

2008

FACULDADE · DE · CIÊNCIAS UNIVERSIDADE · DE · LISBOA

Declaração

Edgar Manuel Cândido da Silva Pascoal, aluno nº 30420 da Faculdade de Ciências da Universidade de Lisboa, declara ceder os seus direitos de cópia sobre o seu Relatório de Projecto em Engenharia Informática, intitulado ”AMOBA – ARINC653 Simulator for Modular Space Based Applications”, realizado no ano lectivo de 2007/2008 à Faculdade de Ciências da Universidade de Lisboa para o efeito de arquivo e consulta nas suas bibliotecas e publicação do mesmo em formato electrónico na Internet.

FCUL, 14 de Julho de 2008

Tobias Schoofs, supervisor do projecto de *Edgar Manuel Cândido da Silva Pascoal*, aluno da Faculdade de Ciências da Universidade de Lisboa, declara concordar com a divulgação do Relatório do Projecto em Engenharia Informática, intitulado ”AMOBA – ARINC653 Simulator for Modular Space Based Applications”.

Lisboa, 14 de Julho de 2008

Abstract

The Integrated Modular Avionics (IMA) main characteristic is sharing of computation resources ensuring a relevant set of safety and real-time guarantees. The ARINC 653 is a well-known standard based on IMA concept. This IMA-based standard has taken a leading role within the aeronautical industry in the development of safety-critical systems. The related cost savings in reduced integration, validation and verification effort has raised interest in the European space industry for developing a spacecraft IMA approach and for the definition of an ARINC 653-for-Space software framework. As part of this process, it is necessary to establish an effective way to test and develop space applications without having access to the final IMA target platform. This dissertation describes the main research key ideas developed during the design and implementation of a multi-platform and modular ARINC 653 simulator, called AMOBA, which shall emulate an execution environment for ARINC 653 space applications.

KEYWORDS:

ARINC653 ; simulator ; space applications ; embedded systems ; real-time operating systems

Resumo

O conceito IMA (Integrated Modular Avionics) introduziu um conjunto de directivas extremamente inovadoras e revolucionarias nos conceitos de *safety*, modularidade, abertura, escalabilidade e independência para sistemas "*on-board*". Os objectivos do conceito IMA são não só obter ganhos financeiros através de processos de desenvolvimento mais simples, reutilização de componentes e recurso a plataformas computacionais padrão mas também ganhos operacionais nas aeronaves ao nível de espaço, consumos energéticos e peso dos sistemas. Tendo as suas bases acentes no conceito IMA, a norma ARINC 653 tem tido um papel preponderante no sector da aeronautica e no ramo dos sistemas criticos. A redução de custos relacionados com a integração, validação e verificação deste tipo de aplicações suscitou o interesse em transferir estes conceitos e tecnologias para outros sectores industriais tais como o espaço. No âmbito desta transferência de tecnologia para o sector espacial surge a actividade AMOBA. O objectivo chave desta actividade é produzir um simulador modular de ARINC 653, que ao mesmo tempo que emula um ambiente ARINC 653, permita aos programadores desenvolver e testar aplicações para o espaço sem necessitar de ter acesso à plataforma final. Esta dissertação pretende descrever o trabalho de pesquisa e desenvolvimento realizado durante a concretização do simulador AMOBA.

PALAVRAS-CHAVE:

ARINC653 ; simulador ; aplicações espaciais ; sistemas embebidos ; sistemas operativos de tempo-real

Table of contents

1. Introduction

The content of this dissertation is result of our research and development activities done during the AMOBA project. This chapter addresses the following issues:

- Motivation to the project.
- AMOBA Project.
- Document overview.

1.1. Motivation

The Integrated Modular Avionics (IMA) is a concept on avionics area that was created with the special purpose to satisfy the avionics industry requirements. Its main feature consists in share the resource components over different applications without fault propagation.

The IMA concept has being applied on several commercial aircrafts, business jets and military planes. The Airplane Information Management System (AIMS) on the Boeing 777 airplane is one of the first IMA implementation on a commercial aircraft. [AIMS 97]

IMA is now implemented on the airborne systems of the two biggest commercial aircrafts Boeing 787 and Airbus A380 [A380-IMA 07]. With this approach it was possible for Boeing to remove 900kg off from the 787 dreamliner. In addition Airbus was also able to reduce to half part of the number of electric equipment needed on board of the A380. [News-IMA 07]

The IMA concept is also applicable to military aircrafts. The Dassault Rafale, F-22 Raptor and F-35 are examples of strike fighters that use IMA philosophy. Another foreseen use of this concept is on the modernization programs that are in progress to upgrade several cargo military planes like KC-135, C-130 and the new Airbus A400M transport.

The F-22 Raptor is a sophisticated combat aircraft that is recognized to have the most advanced avionics and software system on-board. F-22 is the first with a full IMA system integrated where all on board systems work as one.

Some consider that the IMA concept began to be used with the new F-22 Raptor and F-35 fighters and subsequently migrated to the commercial airliners. Alternatively others believe that this concept has been used with less integration, in business jets and regional airliners since the 1980s or 90s. Airplanes such as Dassault Falcon 900, Dassault Falcon 2000, and Dassault Falcon 7X, are business jets which are flying with IMA applications [Dassault-IMA 07].

The following list summarizes some of the IMA systems being used today in avionics industry:

- Airplane Information Management System (AIMS) [AIMS 97] used on Boeing 777
- Common Core System (CCS) used on Boeing 787
- Enhanced Avionics System (EASy) used on Dassault Falcon900, 2000 and 7X [Dassault-IMA 07].
- Modular Data Processing Unit (MDPU) used on Dassault Rafale [Dassault-IMA 07].

The ARINC 653 [A653-Part1, A653-Part2] is a standard based on the IMA concept that specifies a programming interface for a RTOS (Real-Time Operating System), and, in addition, establishes a particular method for partitioning resources over time and memory. At the moment this standard has been established as an important foundation for the development of safety-critical systems in the avionics industry.

At the moment IMA is also the concept candidate for migration to space domain. A considerable effort has been made on the IMA-for-space concept since the IMA's successful implementation on AIMS for Boeing 777 [A653-Space 05, IMA_Space 96]. The success around the AIMS together with several similarities on the development of space on-board software has aroused attention of space community to migrate the IMA concept for space domain [IMA_Space 96, A651, ESA_TechNote 03].

The ARINC 653 standard [A653-Part1, A653-Part2] approach for space purposes will bring new challenges and concepts that have to be explored. A tool is needed that helps on this discovering process and provides a validation, verification and development environment for on-board applications. We called AMOBA to such tool and it aims to emulate an execution environment for space based applications.

Other challenges are associated with the design implementation of AMOBA, in this sense is important that it could be available on several platforms and at the same time be as modular as possible.

Portability shall be an essential issue to consider. It intends to solve the dependency between platforms given AMOBA is intended to be used on as many platforms as possible.

Modularity is also other challenge on the development of AMOBA. It shall promote code organization and an easy way to change modules. In this manner, it would help to find the space requirements for IMA and lead IMA-space concept into the right way.

1.2. AMOBA project

1.2.1. Skysoft

Founded in 1998, with a decade of knowledge in the aeronautics, space and telematics markets, Skysoft Portugal is an enterprise that works for, and in cooperation, with some of the foremost National and International organizations such as the European Space Agency, TMN Portugal, AXA Portugal, SIBS - Sociedade Interbancária de Serviços, Thales Avionics, BAE SYSTEMS Avionics, EADS – Airbus.

Skysoft Portugal works with several operation sectors, implying on different business areas. Based on that, Skysoft operates through three oriented departments:

- Mobility Telematics and Business Solutions (MT&BS) – this business sector is responsible for develop and innovate products such as telematics systems, positioning applications and enterprise systems.
- Space – this division has the responsibility to design and implement products related with ground segment software for space missions, satellite navigation and communication technologies.

- Aeronautics, Security and Defence (ASD) – this department deals with products associated with aircraft avionics software and associated test and simulation tools.

1.2.2. European Space Agency

The European Space Agency (ESA) is an international organisation that counts at the moment with 17 member states. Some of the central ESA's objectives are to promote the development of Europe's space capability and ensure that investment in space continues to deliver benefits to the citizens of Europe and the world.

ESA's main task is to design and implement a European Space programme. These programmes are usually aimed to discover more things about, Earth, its nearest space environment, our Solar System and the Universe. In parallel ESA promotes European industries and establishes relations with space organisations outside Europe.

Since this European agency has an enormous and complex working area, it is strategically organized into several specific operational divisions:

- **European Astronauts Centre** (EAC) – is where the astronauts are trained for future missions.
- **European Space Astronomy Centre** (ESAC) – is the science operation centre for all ESA astronomy and planetary missions together with their scientific archives.
- **European Space Operations Centre** (ESOC) – is the department responsible for controlling ESA satellites and space probes.
- **ESA centre for Earth Observation** (ESRIN) – acts as an interface between ESA and those who use its services.
- **European Space Research and Technology Centre** (ESTEC) – is the main technology development and test centre for spacecraft and space technology.

The European Space Agency is funding and participating with Skysoft in the development of AMOBA.

1.2.3. Project description

AMOBA [AMOBA-Paper 08] stands for ARINC 653 Simulator for Modular Space Based Applications. The project is part of a group of investigation activities on IMA carried out by ESA. IMA is one of the main candidates for a future reference architecture for space systems. A simulator able to run on low-cost hardware and freely available operating systems is considered an appropriate mean to stimulate and intensify research in this area. The main goal is to have a tool able to execute space applications ported to IMA and compliant to the ARINC 653 specification. This tool should execute applications in an environment typical for space systems, like SPARC processor hardware or emulation software; moreover, it should be able to emulate and monitor the execution of hard real-time applications in an appropriate way, hence, a real-time operating system should be targeted.

Main design goals are portability and modularity. AMOBA shall be portable in order to obtain a multi-platform simulation environment for the development of critical software. On the other hand the AMOBA implementation shall be as modular as possible. With a modular approach it should be possible to provide an execution environment to user applications and at the same time allow a progressive integration of components required for the IMA-Space concept being developed.

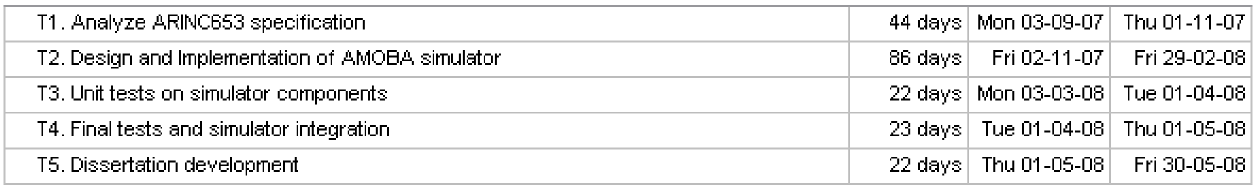
Since that one of the main objectives of AMOBA project is to explore the IMA architecture and adjust it to space requirements on a portable manner, the AMOBA simulator concretization will use the ARINC 653 [A653-Part1, A653-Part2] and Portable Operating System Interface (POSIX) [IEEE1003.1-POSIX] standards.

1.2.4. Project Schedule Plan

The schedule foreseen for AMOBA project is described below. The predicted tasks for this project are structured into 6 subdivided groups as it is shown on the following table.

AMOBA Workpackages
WP0 - Project Management and Quality
WP1 - Study of ARINC 653 Application to the Space Domain
WP2 - AMOBA ARINC 653 Simulator Design
WP3 - AMOBA ARINC 653 Simulator Development
WP4 - AMOBA ARINC 653 Simulator Assessment and Trials
WP5 - AMOBA ARINC 653 Simulator Market Analysis

My work on the AMOBA project involves only the workpackages WP2 and WP3, the tasks covered can be detailed as follows:



T1. Analyze ARINC653 specification	44 days	Mon 03-09-07	Thu 01-11-07
T2. Design and Implementation of AMOBA simulator	86 days	Fri 02-11-07	Fri 29-02-08
T3. Unit tests on simulator components	22 days	Mon 03-03-08	Tue 01-04-08
T4. Final tests and simulator integration	23 days	Tue 01-04-08	Thu 01-05-08
T5. Dissertation development	22 days	Thu 01-05-08	Fri 30-05-08

1.3. Document overview

This document represents the dissertation for master degree in informatics engineering. The purpose of this document is to expose the research findings and conclusions ideas reached during the AMOBA project execution. Within this document will be illustrated issues regarding the actual state of knowledge around technologies concerned on the project. The objectives of the AMOBA project together with architecture and design issues will be also mentioned.

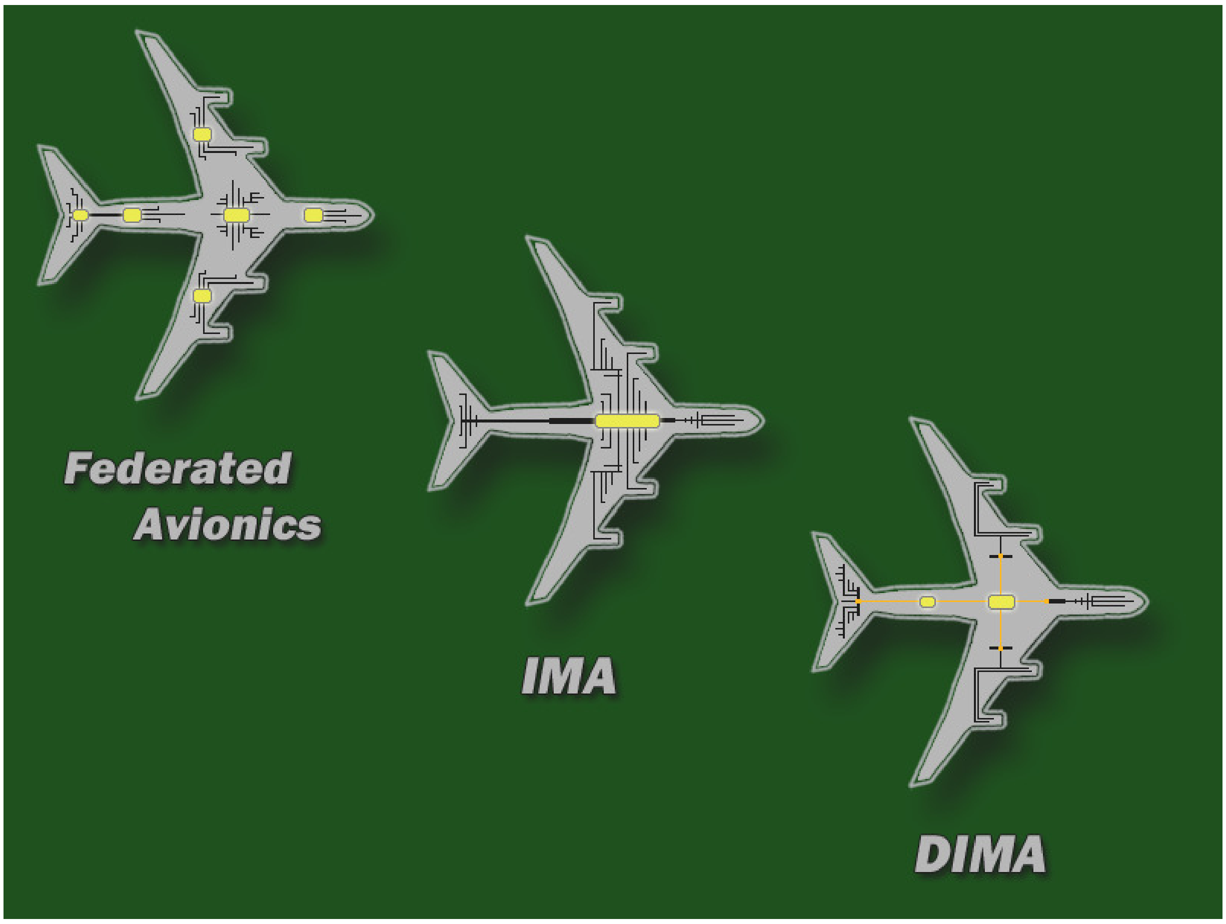
The document is organized as follows:

- Introduction – summary of the document purpose

- State-of-the-art – presents the actual state of knowledge around technologies regarding the project in concern.
- Conception Requirements – describes the general AMOBA needs and the desires expected by space domain
- Simulator Conception – explain issues regarding the simulator conception
- Concluding Remarks and Further Issues – summary about the work done, conclusions and further work.

2. State-of-the-art

2.1. Avionics architectures



Originally at the avionics beginning, around 1970, there was a tight relation between a function and a piece of equipment (component / sub-system) that supported this function. Components implementations were mainly mechanical and highly independent. The term avionics – aviation electronics – appeared and was popularised when electronic devices were introduced and generalised, mainly for military aviation. Today, avionics covers communications, localization, navigation, sensors, displays, flight control, collision avoidance, etc.

The introduction of computer-based control provided more flexibility than pure analogue electronics, inside devices and in interconnection of those devices. But even with this, each piece of avionics equipment was specific: specific processor, specific memory, specific programming language, etc. Sharing the definition and development of these items was a natural evolution, similar to what happened with electronics.

2.1.1. Federated Avionics

Federated avionics is an architecture that follows a “loosely coupled” approach and it is typical of most legacy avionics flying today. In this avionic architecture the Line Replaceable Units (LRUs) are dedicated hardware devices geographically separated which intends to satisfy specific purposes within the avionic system. Usually exists dependencies between LRUs and any functionality addition to system often results on major system redesign, integration test and verification work.

The resource sharing occurs at the last link in the information chain, via the controls and displays. Also, a time-shared multiplex circuit “highway” is used. Several standard data processors are often used to perform a variety of low-bandwidth functions such as navigation, stores management and flight control. The data processors are interconnected by time-division multiplex buses which range in data rate capability from 1 megabit/second (MIL STD 1553) to 20 megabits/second (STANAG 3910).

When moving from a federated to an integrated architecture, certain difficulties need to be overcome. The two main ones are to provide data separation, previously provided by virtue of physical and geographical separation, and to move towards fault management at system level, which is more effective and easier to implement in an integrated system.

2.1.2. Integrated Modular Avionics

The fundamental difference between federated avionics and IMA is essentially related to the approach used on resource management (computing, communication, and I/O).

Presently, most advanced avionics solutions have adopted IMA. As stated in the Integrated Modular Avionics (IMA) development guidance and certification considerations [DO-297 05], IMA is a shared set of flexible, reusable, and interoperable hardware and software resources that, when integrated, form a platform that provides services, designed and verified to a defined set of safety and performance requirements, to host applications performing aircraft functions.

Computing modules are physically grouped into racks, called cabinets, and are interconnected via specific or more generic busses [AFDX 04]. Modules can be specialised to support, e.g., pure data processing, graphical processing, power supply or Input/Output (I/O). This module standardisation facilitates definition and development. But this integration also brings benefits in term of avionics volume, weight, power consumption, maintenance or conditioning. Reduction of the number of spare types and parts also bring cost savings. Obsolescence is better defined and anticipated.

However, in order to respect certification constraints, ease integration and maintenance, integration of software components on the same physical module imposed the definition and development of specific Operating System (OS) that should ensure spatial and temporal segregation that were naturally found with federated solutions. ARINC 653 is the result of a standardisation effort in this area. Nevertheless, certain avionics systems do not use this standard API. Even if Commercial-Off-The-Shelf (COTS) implementations exist, some suppliers prefer to use their own implementation. In spite of these differences, the main principles are present in all solutions.

The OS and the API guarantee a high level of independence between the application software and the hardware execution platform, hence ensuring a reduced impact of obsolescence of electronic components, and an improved upgrade ability and portability.

In spite of its advantage, IMA approach raises some difficulties. Limits of responsibilities must be defined differently.

Definitely, there is a room left for evolution and improvement. For example, the level of services offered by ARINC 653 is still low and could be elevated. Current evolution of general processors, e.g., increase of the number of *cores*, will necessitate new language features where distribution becomes more natural than it is today. If this becomes a reality in Information Technologies (IT) world, it may well have an impact on avionics too.

2.1.3. Distributed Integrated Modular Avionics

Soon the electronic systems and particularly avionics will need to meet higher demands regarding the costs. A new trend is emerging to reduce the on-board components needed and at the same time to meet these tight requirements without renounce to safety, maintainability and operability. This new concept is called Distributed Integrated Modular Avionics (DIMA).

The DIMA philosophy combines the advantages from the traditional, federated avionics concept and those from the Integrated Modular Avionics (IMA), by being a physically distributed but functionally integrated system.

The DIMA concept is based on reusable building blocks with viable and standardised interfaces. Consequently it accomplishes a high degree of flexibility and capability for expansion. This concept also support layered software architecture by providing hardware independent application software services.

The concept makes it possible to allocate processing capability close to sensors and actuators and thereby accomplish a digital network with the following advantages:

- Improved capability to re-configure the architecture.
- Improved capability to add, upgrade, or change functionality.
- Reduced wiring between units.
- Improved transmission capability.

2.2. ARINC653

The ARINC 653 [A653-Part1, A653-Part2] specification is an IMA-based standard, where the partitioning concept emerges as a way to ensure protection and functional separation between applications. The partitioning concept aims enforcing fault containment, preventing fault propagation from one partition to another. It also eases system validation, verification and certification procedures.

The ARINC 653 standard specification describes a partition as being roughly the same as a program in a single application environment (comprising code and data, its configuration attributes and execution context). One or more processes concurrently execute in a partition by sharing access to the processor infrastructure and other hardware resources.

The standard demands the functions resident on a core module to be partitioned with respect to space (memory partitioning) and time (temporal partitioning). A partition is therefore a program unit of the application designed to satisfy these partitioning constraints. A core module is responsible for enforcing partitioning and for the management of the individual partitions. Spatial (or memory) partitioning is achieved through allocation of predetermined areas of memory to each partition. Temporal partitioning ensures each partition uninterrupted access to common resources during predetermined time periods. Partitions are scheduled on a fixed, cyclic basis.

Configuration of all partitions throughout the whole system is expected to be under the control of the system integrator and maintained with configuration tables. The configuration table for the partition schedule will define the system Major Time Frame (MTF), which will be cyclically executed.

Standardisation of the interface will allow the use of application code, hardware, and OS from a wide range of suppliers, encouraging competition and allowing reduction in the costs of system development and ownership. The APEX (ARINC 653 APlication EXecutive) interface provides a common logical environment for the application software.

The following objectives are expected to be achieved with the APEX interface:

- Portability: The APEX interface facilitates portability of the software. It is desirable for the application software developed for a particular aircraft to be ported to other aircraft types with minimal recertification effort. By removing language and hardware dependence, the APEX interface achieves this objective.
- Reusability: The APEX interface allows the production of reusable application code for IMA systems. The APEX interface will reduce the amount of customizing required when a component is reused.
- Modularity: The APEX interface provides the benefits of modularity when developing application software. By removing hardware and software dependencies, the APEX interface reduces the impact on application software from modifications to the overall system.
- Integration of software of multiple criticalities: The APEX interface supports the ability that applications of different levels of criticality co-exist on the same system.

The ARINC 653 standard is subdivided into three parts:

- ARINC 653 Part 1 - Required Services: It defines the minimum set of services that must be provided. It is basically focused on services to provide robust partitioning, inter and intra partition communication and health monitoring [A653-Part1];
- ARINC 653 Part 2 - Extended Services: It defines additional services to increase the interoperability and support of additional Host Application. Could be highlighted the File System, ARINC 664 Ports

(SAP), Log, data format exchange standardization and the support for multi-configuration modes for partition scheduling (useful for start-up, debugging and mode switch) [A653-Part2].

- ARINC 653 Part 3 - Conformity Test Specification: It provides test procedures to verify the conformity of the middleware implementation against ARINC 653 Part 1.

The ARINC 653 standard is still under high volume of revisions. One important revision for ARINC 653 Part 1 is the change to the XML schema configuration format.

For ARINC 653 Part 2, could be highlighted the addition of shared memory and timer services. The ARINC 653 Part 3 effort is to extend the scope of the compliance test to ARINC 653 Part 2.

In both avionics and space industries, the safety concept is of paramount importance. The ARINC 653 standard was developed with the purpose that all safety critical software embedded in a system must follow very strict and demanding rules both in terms of operation and certification.

ARINC 653 is a standard based on IMA architecture that specifies mutually the interface, and the behaviour of the API services. This specification is intended for use in a partitioned environment, specifically, the partitioning of computer resources over spatial and temporal segregation methods.

The general architecture of a standard ARINC 653 system is illustrated in Figure 1. It comprises the application software layer, with each application executing in a dedicated partition, and a given set of system partitions. The system partitions are optional components and are intended to manage the interactions with specific hardware devices. Appropriate support from the core software layer (e.g. hardware interfacing and device drivers) is required.

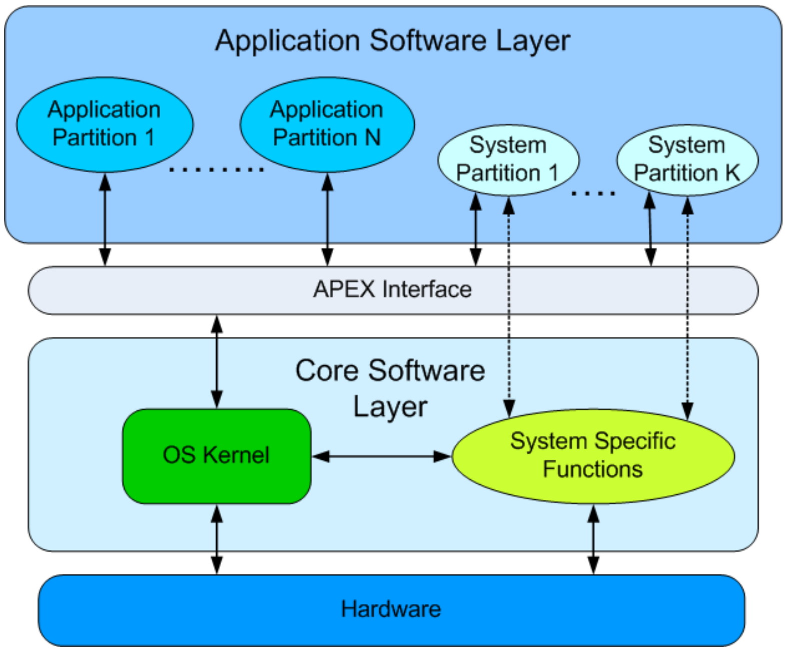


Figure 1 – Overview of the standard ARINC 653 System Architecture

Application partitions consist in general of one or more processes that exclusively use the services provided by a *logical* Application Executive (APEX) interface, meaning calls can only be made to the set of primitives defined in the ARINC 653 standard specification [A653-Part1]. However, a system partition may use also a set of specific functions provided by the core software layer and may therefore bypass the standard APEX interface, as illustrated in Figure 1.

In any case, the execution environment provided by the Operating System (OS) kernel module must furnish a relevant set of operating system services, such as process scheduling and management, time and clock management as well as inter-process synchronization and communication. The application software layer is obliged to strict robust space and time partitioning. Containment of possible faults inside the domain of each partition must be ensured in the core software layer [A653-Part1].

Thus, each partition makes use of a logical execution environment. Given most of the process-level services that need to be supplied by the OS kernel module are already provided by common off-the-shelf real-time operating system (RTOS) kernels, a natural approach to the definition and design of ARINC 653 systems may use the functionality provided by a given RTOS kernel (e.g. RTEMS [RTEMS_C 03, RTEMS_POSIX 03, RTEMS_Qualify 05]), completed with the specific functions needed for ARINC 653 system operation, namely partition management mechanisms [AIR-Paper 07, NASA_Partition 99].

The execution environment of each partition includes the corresponding application program (code and data), its configuration attributes and execution context (e.g. stack).

2.3. POSIX Standards

POSIX stands for Portable Operating System Interface [IEEE1003.1-POSIX], the term POSIX was suggested by Richard Stallman. Stallman is the main author of several copyleft licenses like the most used free software license, the GNU General Public License and is the founder of Free Software Foundation (FSF), a non-profit corporation that aims to promote the distribution and modification of computer software with no constraints.

In earlier editions of the IEEE standard, the term POSIX was used as a synonym for IEEE Std 1003.1-1988. Currently the name POSIX refers to a family of related standards: IEEE Std 1003.n (where n is a number) and the parts of ISO/IEC 9945.

Composed by a set of standards, POSIX was created by the Institute of Electrical and Electronics Engineers (IEEE) with the purpose of ensure source-code portability of application programs over different hardware and operating systems. This standard is recognized by the International Organization for Standardization (ISO) and American National Standards Institute (ANSI).

A set of standards is contained on the formal standard POSIX 1003.1 these are:

- POSIX 1003.1 defines the standard for basic system services on an operating system, and describes how system services can be used by POSIX applications. These services allow an application to perform operations such as process creation and execution, file system access, and I/O device management.
- POSIX 1003.1c defines a set of thread functions that can be used in the design and creation of multithreaded realtime applications.

- POSIX 1003.1b provides support for functions that support the needs of realtime applications, such as enhanced interprocess communication, scheduling and memory management control, asynchronous I/O operations, and file synchronization.

A combination of these standards is used on AMOBA implementation which makes it portable. It can be ported to another host system that supports also the same POSIX standards without any code modifications.

2.4. Real-Time Systems

Nowadays, computation has become an essential need like transportation, electricity or even sanitation. For some people, computers are only recognized as those things that are at home with a monitor, mouse and keyboard. In fact at the moment, computers are everywhere, such as: cellular phones, video recorders, cars, aircrafts …

Embedded systems usually are real-time systems used to control the computers devices in order to accomplish their tasks.

Real-Time System definition:

"Real Time Systems are those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced"

Types of RT Systems

In real-time systems, there frequently exist applications with very different temporal requirements: some must guarantee timely service under all circumstances while others have no such constraint, but are instead expected to work "as fast as possible" by making good use of all resources they can possibly get. The differences between such real-time and non-real-time programs are reflected in the functionalities they need underlying operating system to provide. So real-time systems are usually categorized in the following types:

- Hard Real Time - real-time programs for which even the slightest violation of timeliness specifications (e.g.: task deadlines) must be considered a fatal error.
- Soft Real Time - real-time programs for which it is generally desired to complete within a fixed amount of time, but exceeding the deadline occasionally is not harmful.

2.5. RTOS

Real-time operating systems (RTOS) are the base software for the real-time systems. They make the link between the software applications and the hardware resources. They must be robust enough to deal with unexpected events without causing any irregular behaviour.

In earlier days, some proprietary RTOS was developed by aerospace companies in order to achieve optimized performance for specific functions in their own real time systems. Like, Honeywell with the “digital engine operating system” (DEOS) and Rockwell Collins, with the “virtual machine operating system” (VMOS).

Today, there is a large diversity of COTS (Commercial-off-the-shelf) and FOSS (Free and Open Source Software) RTOS available on the market, which makes the selection of the appropriate RTOS a difficult and extremely important task to accomplish before the concretization phase. Besides, it is important to consider if the concerning real time system requirements matches with the RTOS features.

Real-time applications usually run on top of the operating systems designed to secure a timely behaviour known as real-time operating systems (RTOSes). The RTOS main purpose is to provide support services to real-time systems / real-time applications, so system developers don’t have to worry about further complex problems when developing a system like this.

The usual categories of services available on a normal RTOS are:

- Task Management
- Memory Management
- Time Management
- Input/Output Management
- Inter-Task communication

Task Management

Task Management can be seen as a service provided by the RTOS that allows the developers of real-time systems to organize their software code in different execution sections. Each section is called a “task”.

When the system is running is the task management service that chooses the executing tasks. The choosing decision is related with the scheduling algorithm, the task priority and the task state. Task Manager also provides a set of services to manipulate task execution, for example: change priority, start a task, or stop a task.

Inter-Task Communication

The Inter-Task Communication is a category that is present in most RTOS. It has a set of services available to pass information between tasks in a reliable manner.

The services contained on this category also intend to provide synchronization means so it could be possible for tasks to coordinate between each other and cooperate with other tasks.

Time Management

Real-time systems have real-time requirements, for instance, a task could need to wait a certain period of time. Besides it also could need to adjust the deadline times or invoke period events. Most of the RTOS kernels provide services to satisfy these temporal requirements such as task delays, time-outs and periodic events.

Memory Management

This category of services provides mechanisms for manipulate memory dynamically. These services are able to supply to tasks the opportunity to allocate blocks of RAM memory for temporary use. Neither all RTOS provide memory management services. This fact is related with the need to save memory within memory-limited environments and with the provision of timeliness guarantees.

Input/Output Management

The Input/Output Management category of services is responsible for provide to tasks a method to access and control hardware devices. This management is done by the RTOS kernel, generically through a device drivers or Interrupt Manager specific for the hardware device in concern.

RTEMS

RTEMS stands for Real-Time Executive for Multiprocessor Systems, it is a free open source real-time operating system (RTOS) distributed under GPL license and designed specially for embedded systems. RTEMS provides a POSIX API. It is available for several target processor architectures (e.g. SPARC, Intel x86, ARM, PowerPC …). Part of the AMOBA development used RTEMS as a verification platform to perform verification tests over different platforms and architectures.

2.5.1. Microkernel

Basically, the microkernel concept consists in a “small” kernel. A usual kernel is normally the core component of an operating system (OS) and manages all system's resources.

A microkernel is a reduced kernel (is not an OS) whose purpose is to provide only the essential mechanisms in order to implement the Operating System services. [Liedtke 95]

The earliest invented microkernels emerged around 1970, at the beginning this concept was not well accepted because in practice denoted several performance problems. The experiences on first-generation microkernels like Mach and Chorus demonstrated a lack of performance on systems based on them.[Bradley_and_Bershad 93]

However, in 1993, a German computer scientist, well-known for his work on microkernels named Jochen Liedtke, demonstrated that these performance problems weren't inherent in the microkernel concept [Liedtke 93]. The operating systems based on microkernels has some benefits, this concept is normally known to provide several advantages such as:

- Flexibility and Extensibility. Microkernel system has the ability to be easily adapted to new hardware or software. Only selected OS modules needed to be modified or added to the system. Modifications can be made and tested online.
- Reliability and Maintainability. Small and organized code is easier for developers to update the system and avoid errors. Reliability on Operating Systems can be improved by exploiting the inherent strengths of microkernel technology [uKernel 07].
- Fault containment. OS modules malfunctions are as isolated as normal application malfunctions
- Portability. Changes to port the system to new machine architecture are only needed in the microkernel –not in the other services.

Reliability and Safety are issues that shall be most of times present on RTOSes conception. Generally RTOSes need to accomplish their goals in a time contained manner an unexpected failure can lead into a dramatic failure into the system. The RTOS trustworthiness is not an option when the safety is taken as a requirement. [Lane 00].

“The second-generation microkernels may be a basis for all types of operating systems, including timesharing, multimedia, and soft and hard real time” [Liedtke 96]. Some studies are being developed for real-time programming on top of microkernels [Ruocco 06, uKernel_RT 05, uKernel_RT 01]. Other success evidence of the microkernel’s technology is the microkernel-based RTOSes that are currently used on several domains. Next follows some examples:

- Sysgo PikeOS. Uses a L4kernel microkernel approach and it’s traditional PikeOS usage domains are safety and mission critical systems [PikeOS].
- Windriver VxWorks is a well known microkernel-based RTOS are mostly used on industries like automotive, avionics, space and robotics.
- Greenhills INTEGRITY® is built on the velOSity™ microkernel, it is used mostly on avionics and automotive industries.
- Honeywell DEOS is a proprietary microkernel-based operating system for avionics real-time application [DEOS 01].
- Phoenix-RTOS is an open source microkernel-based available under GPL license. It intends to provide a platform for embedded systems to be targeted on board computers or system on chips.
- QNX has a microkernel RTOS design that aims as primary objective supply the embedded systems market.
- Phar Lap ETS. Based on ETS microkernel this RTOS is used for instance on LabVIEW real-time targets.

Some recent microkernels approaches are now evolving to support IMA concept as explained in [L4_uKernel 03, IMA_L4 02], this new trends could be an important step to improve RTOS’s features.

2.5.2. RTOS ARINC 653

With other features available than usual RTOS, the RTOSes ARINC653 follows a philosophy based on IMA and are mostly used for avionics systems. A RTOS to be considered compliant with ARIN653 has to complete a demanding certification process.

RTOSes ARINC653 have to be extremely reliable, therefore safety of airborne software an essential issue to consider. In order to guarantee more reliability around safety-critical on board systems, some RTOSes especially those that are ARINC653 compliant are certified according with DO-178B standard.

RTOS ARINC653 provides a genuine ARINC653 execution environment while AMOBA intends to provide a simulated ARINC653 environment.

2.5.2.1. COTS RTOS ARINC653

Commercial, off-the-shelf (COTS) is a term generally used on technology or computer domains which means a product that is ready-made and available for sale, lease, or license to the general public.

At the moment the available COTS RTOS in the market compliant with ARINC653 [ESE_News 06] are:

- "LynxOS 178", from LynuxWorks
- "CsLEOS", from BAE systems
- "Integrity 178B", from GreenHills
- "VxWorks 653", from WindRiver
- "PikeOS", from Sysgo

LynxOS 178

The LynxOS®-178 is a commercially available real-time operating system, it is produced and distributed by LynuxWorks. The system was developed and certified accordingly with the premier aviation industry's software safety standard RTCA/DO-178B, Level A. The LynxOS conforms to the POSIX 1003.1 system call interface standard and has been implemented according to the POSIX 1003.1b real-time extensions and the 1003.1c threads extensions. [LynxOS]

CsLEOS™

Developed by BAE SYSTEMS, the CsLEOS™ RTOS intends to provide an approach to reduce development costs in safety-critical applications. CsLEOS is certified by the most recognized safety standard in avionics domain, the RTCA/DO-178B Level A. This operating system is also compliant with the POSIX and ARINC653 standards. [CsLEOS]

INTEGRITY®-178B

Integrity-178B is an RTOS manufactured and marketed by Green Hills Software. Designed for real time the Integrity-178B is POSIX and ARINC653 compliant, it provides a portable and partitioned environment for safety critical applications containing multiple programs with different levels of safety criticality on a single processor. The Integrity-178B safety is guaranteed and engineered accordingly with the DO-178B Level A standard. [GHS-Integrity178]

VxWorks® 653

VxWorks 653 is Wind River's robust operating system for controlling complex ARINC 653 IMA systems. This RTOS offers complete ARINC 653-1 compliance and DO-178B certification evidence, a range of language options – C, C++ or Ada is available for ARINC 653-1 system development with this product. Applications can be written through VxWorks, ARINC or POSIX APIs. The partition-level operating system varies for different certification levels and adaptation of legacy operating systems, enabling the OS to run existing code with little change. [VxWorks653]

PikeOS

PikeOS is a product from SYSGO AG providing a platform for embedded systems where multiple applications can run simultaneously in a secure environment. PikeOS provides support for a variety of hosted operating systems, runtime environments and APIs. It also enables legacy applications (e.g. Ada or legacy RTOS) to run concurrently with new applications based on standards like POSIX and ARINC 653. All these OSs, runtime environments and APIs run on the same PikeOS kernel and can be combined as needed. [PikeOS]

3. Conception Requirements

3.1. General requirements

The AMOBA simulator must address several purposes that are intended to be necessarily achieved upon concretization. In particular, the simulator shall have the following capacities: be able to work on several platforms (Portability), ease on changing (Modularity) and be compatible with a given specification in the time domain or at least with a representative temporal ordering events. We catalogued these requirements into 3 different categories as explained next.

Portability

Portability is a paradigm that is used on software engineering. A software element is considered to be portable if can be installed into a new environment without redevelop it. In this context a software element denotes a system, a module, a component or other piece of software. [Mooney 97,. Mooney 90]

One of the major purposes of this simulator is to be portable to several platforms.

Why portability is a requirement for AMOBA?

There are two different points of view:

- The first is the developer viewpoint. Space developers work mostly with complex hardware components. These can be expensive and not immediately available. So it would be a great advantage for space programmers if they could have access to a platform immediately available that offers an environment similar to the final platform.
- The second is the AMOBA perspective. From AMOBA's perception there is a special reason for portability. This is related with the fact that AMOBA will be a simulator commercially available, thus, will be a benefit if it is available on several platforms and subsequently accessible to much more users.

Portability could bring new opportunities to the space application development process. As the simulator can be easily ported to several platforms the space developer can choose the better environment to adapt to its needs. Besides, this development process can be done on an evolutive manner, i.e., it can be separated into different stages. The underlying environment doesn't need to be, at first sight, the end-platform. For instance, the developer can choose for the first stage a functional (non real-time) environment to implement the application and on the next development iteration decide to port the application to an emulation of a real-time environment. Some environments supported by simulator are illustrated later on the section 4.4 Emulation environments.

Modularity

Modular design is a discipline of the software engineering that began on the 1970's. Around that time, David Parnas, one of the pioneers in this area, wrote a significant paper "On the criteria to be used in decomposing systems into modules" [Parnas 72] that still is a reference for many researchers today. Years later new ideas based on the modular pioneers theories came out [Modularity 02, Modularity 01].

Although exists many theories around this subject the concept remains the same. In computer science the "Modular design" consists in an abstract method for project systems that: divides complexity and establishes boundaries through concept blocks, named modules. Module is a piece of the whole system that represents a specific purpose within the system.

Despite the Modular Simulator is not a main need for AMOBA, it could bring many advantages to follow this approach. The advantages are mostly related with the capacity to reduce complexity and to promote simplicity for changes.

Temporal requirements

One of the constraints imposed by the ARINC653 standard is the temporal segregation. This constraint enforces strict time periods of execution for each partition. As a consequence, ensures that each partition has a given execution period, even if other partitions are faulty. Therefore it is important to guarantee that the simulator's underlying environment (OS and Hardware) supports the accuracy and granularity expected to fulfil the partition execution time division requirements.

3.2. Space Requirements

3.2.1. Applying IMA and ARINC 653 to Space

The IMA concept and ARINC 653 standards are still in a very early phase of introduction on the space market. However, they have achieved a high degree of maturity on the aeronautic market both with the publication of the ARINC 653 standard by the AEEC and also with the inclusion of ARINC653 systems and general IMA concepts in the current generation of aircrafts produced by the big air framers, namely Boeing 787 and Airbus A380. The practical usage of the IMA concept and ARINC 653 standard on so complex and relevant projects is a strong evidence of the validity and utility of the technology and provides great confidence that it shall be of great usefulness to the space market. On the other side the obvious similarities on the space and aeronautics markets imply that a relevant amount of work is already done and can be reused, being the adaptations to the space market specificities minimal in relation to the amount of work to be reused.

However, the differences between the two markets must be carefully studied. Avionics concepts will probably not be applied one-to-one to the space context; they will undergo certain modifications, improvements or pure alterations. Discussions and research activities in the aeronautical context are also addressing potential improvements in the IMA design. Some of the proposed improvements appear to point in a very similar direction as the discussion related to space. The modifications IMA has to undergo to fit into space needs are probably also necessary for future avionics requirements. The AMOBA project is certainly not the central platform for this discussion. But AMOBA may be seen as a model project within a broader framework: The application of IMA to space. The next sections will discuss some of the differences and the resulting issues. AMOBA is not able to solve all these issues, but it is a good occasion to study them.

3.2.2. Open Issues with ARINC in Space

Despite the similarities in aeronautics and space markets, there exists a range of differences, especially with unmanned spacecrafts and satellites which lead to different software requirements and must be addressed by AMOBA or by future work in this area.

Manned missions are put aside here for two reasons; first, because they have less relevance for market considerations; second, because the differences are more of a quantitative than of a qualitative nature: as in aircrafts there is personnel on board of manned spacecrafts to intervene in system processing. Higher safety levels that arise with manned space missions are related to the life hostile environment of space, so there is the need for a plus in safety. Since safety is already a core feature of avionics systems the mentioned plus is not a paradigmatic change, but a plus in the very meaning of the word.

The main differences dividing space and aeronautic contexts follow from three characteristics of space missions:

- Their duration - be it a spacecraft on a journey to the outer planets, be it a satellite orbiting earth – which makes it impossible to update or maintain the computer systems physically in a ground based hangar
- The lack of personnel on board to intervene in critical situations or to change mission programs
- The distance from ground which makes it difficult not impossible to intervene in time from ground control, not to mention the impossibility of an emergency landing

These characteristics imply **autonomy**, **flexibility** and the ability of **remote control** as core features of systems in space.

Autonomy means that the system is able to perform short time decisions on its own. It must be able to control itself and the proceeding of the mission, to take decisions on the base of this analysis and to perform the actions which follow. An important instance of autonomy is a health-monitor not only checking responses of devices for errors but also checking the entire spacecraft actively. The system must be enabled to draw conclusions of hardware or material failures, to substitute a running component with a redundant one [SPIDER 05]. The necessary complement of such a health-monitor is a redundancy concept enabling the system to take actions when it is required. Redundancy is demanded as a basic service provided by IMA space systems, but there is no universal redundancy solution for all problems, all hardware and all applications. So redundancy is not treated as a requirement for the platform but as a requirement for supplier of Harware and Software components.

Flexibility means that the system is reconfigurable during runtime – a more dynamic configuration concept is needed than that proposed by ARINC 653. Reconfiguration includes the fall-back to a reduced operational mode to guarantee the realisation of the main goals of the mission dropping additional tasks in case of emergency, switches to different hardware components including their controllers and their system-wide addressing in case of hardware failures and the update or substitution of running software.

The lack of pilots and system administrators in unmanned missions requires the possibility to control the system remotely. **Remote control** demands communication channels from ground control not only to the control system but also to the payload applications hosted on this system. For this a robust security concept is necessary that authenticates controllers and authorises tasks to happen on the space system. The channels must be encrypted to secure sensible information like passwords or system internals.

An additional difference, not as fundamental but with important implications to system design, is defined by typical payloads of space missions. Aircrafts normally transport people or cargo – scientific experiments on board of an aircraft are rather exceptions. This is not true for space missions. Their payload normally includes computer systems dedicated to scientific, commercial tasks. The payload systems are complex and normally produce a great amount of data traffic, controlling devices or exchanging data with ground stations. Including those systems in an IMA-like environment has significant impact on traffic on the data buses used by the entire system. Special care to the design of data exchange and I/O must be taken.

There are some concrete requirements that can already be outlined: The need for dynamic configuration, remote connection, security, advanced health monitoring and a carefully designed approach to data exchange including I/O-tasks.

There are more issues which may be addressed. Traditionally other programming languages and approaches were used in space context. It may be considered to enlarge the set of language bindings by FORTAN, FORTH or even HAL/S to meet these cultural differences strengthening a multi-supplier approach at the same time. A closer look reveals that most of those languages are disappearing or niched to very special environments like the NASA Space Shuttle project [NASA 88]. More interesting appears the adoption of modern object-oriented languages like Java widely used in industries outside the aerospace context.

Space phenomena like Single Event Upset (SEU), energy charged particles with impact on memory cells and processor state, must be considered not only for hardware decisions but also for the design of failure recognition and tolerance mechanisms. The space – and also the aeronautical - community have already produced a wide range of studies concerning radiation problems [NASA 88, NASA_Partition 99] and solutions are available. Therefore it seems not necessary to deal with these phenomena in this dissertation. This is also true because the simulator, the main goal of the project, won't be exposed to such conditions.

4. Simulator Conception

Within this section will be described issues regarding AMOBA simulator conception. This simulator conception approach will not concern the previous aspects mentioned regarding the spatial requirements. The modifications to meet the spatial requirements are not within the scope of this dissertation it will be integrated when a stable version of the simulator ARINC653 is completed.

This chapter includes four sub-sections:

- “Design concepts” – Explains the overall concepts that shall be achieved on the simulator conception
- “Architecture definition” – Intends to define the architecture for the simulator.
- “Design specification” – specify the role for the components identified on the architecture phase.
- “Emulation Environments” – Describes several possible platform scenarios for the simulator

4.1. Design Concepts

One of the main purposes of AMOBA simulator is to provide to users an execution environment with the capability to execute and verify ARINC 653 applications. This simulator aims to provide a low cost, yet effective, environment to develop space applications and verify their behaviour without having access to the final target platform and without the need for a real ARINC 653 RTOS.

Another fundamental design attribute of AMOBA concerns with the modularity and portability of the simulator. A modular simulator design shall promote code organization and an easy way to change modules. In this manner, it would help to find the space requirements for IMA and lead IMA-space concept into the right way. On the other hand a multi-platform approach shall provide availability of simulator on every POSIX-compliant operating system.

In addition to allowing verification of ARINC 653 applications over any POSIX-compliant operating systems, AMOBA shall also provide extra capabilities to the user like logging time measurements, and overall system monitoring, such as detecting and reporting failures. In this sense, AMOBA emulates and extends the scope of the functions assigned to the Health Monitor component foreseen on the ARINC 653 standard.

4.2. Architecture definition

“…architecture is not just a phase or an activity in the software development life cycle, but a discipline pervading all phases of development.”
Institute for Software Research University of California, Irvine

Complex systems like spacecrafts can have many requirements to accomplish and several services to provide. At the same time that the software system grows, the dimension also increases, thus it is important to organize and define the main service components. The architecture has an important role on the Simulator design and implementation.

The AMOBA Simulator intends to be modular and multi-platform, so it must avoid dependency between APEX and the OS Kernel (cf. Figure 1), which means that, it shall not access directly the given OS kernel interface unless it is strictly necessary. To avoid and control that possible dependency AMOBA foresees a middle abstraction layer, called “AMOBA-Core”, which provides support to a “virtual” ARINC 653 execution environment.

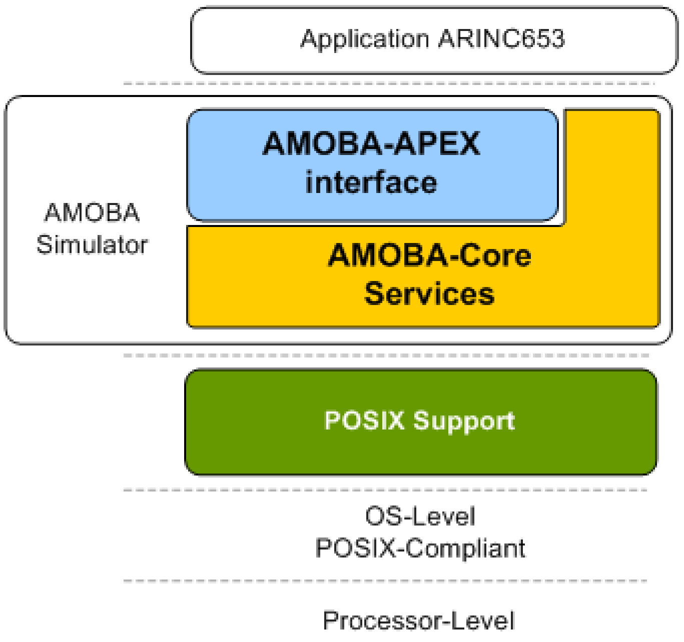


Figure 2 – AMOBA: ARINC653 Simulator Architecture

The architecture chosen for AMOBA is similar with ARINC653 architecture mentioned in section 2.2 ARINC653. AMOBA follows a layered design approach as shown in Figure 2, the main modules identified for the simulator are:

AMOBA-APEX Interface - is the APEX interface implementation within the AMOBA simulator and its main objective is to provide the applications with the set of primitives defined in the ARINC 653 standard specification and supply compatibility/portability to APEX-based applications.

AMOBA-Core Services - is the central part of the AMOBA simulator and its main objective is to provide all OS necessary services to upper layers making use of a POSIX interface (e.g. RTEMS POSIX API [RTEMS_POSIX 03]). The following components have been identified to be included in the AMOBA-Core Services module:

- configuration management;
- time-measurement and monitoring;

- ARINC 653 functional emulation.

The configuration and time-measurement/monitoring components are mandatory in the AMOBA architecture and must always be present. The ARINC 653 functional components are intended to provide the services needed for APEX implementation, such as partition management, process scheduling and management, time and clock management.

4.3. Design specification

4.3.1. AMOBA-APEX layer design

This section describes the APEX services as specified on the ARINC653 standard and as it will be subsequently implemented on the AMOBA simulator.

As described before on section 2.2 ARINC653, the ARINC653 foresees a design for an executive interface (APEX). This interface specification defines a set of services that shall be implemented in order to achieve an execution environment for ARINC 653 space applications.

The AMOBA-APEX layer implementation shall follow the services as defined on APEX these are divided into the following categories:

- Partition Management
- Process Management
- Time Management
- Intra-partition Communication
- Inter-partition Communication
- Health Monitoring

4.3.1.1. Partition Management Services

Partitioning is a fundamental concept of ARINC 653 standard. A partition is an element of the standard specification that holds several processes and shall satisfy space (memory partitioning) and time (temporal partitioning) constraints.

Partitions are scheduled on a fixed, cyclic basis. They are activated accordingly to a pre-defined scheduling plan. The order of partition activation is defined at configuration time using configuration tables.

Each partition has an operational mode, indicating the partition state or operating mode. Every partition can be in one of the four available operational modes:

- IDLE: In this mode, the partition is not executing. The partition is not initialized, no processes are executing.
- COLD_START: In this mode the partition is on the initialization phase, the partitions initialization code is executing.

- WARM_START: In this mode the partition is on the initialization phase. This mode is similar to the COLD_START mode, with minor differences concerning the circumstances of the initialization.
- NORMAL: In this mode, the partitions process scheduler is active. All processes within partition have been created and those that are in the ready state are able to run.

The partition management contains two services; they can change the operational mode of the current partition or get partition status of the running partition:

- GET_PARTITION_STATUS; and
- SET_PARTITION_MODE.

AMOBA-APEX accomplish the Partition Management Services using the Partition Manager of the AMOBA-Core layer. This last module essentially is responsible for the schedule of partitions and also to provide services to control partition execution, it will be described in the section 4.3.2.2 Partition Manager.

Within AMOBA, partitions have representations at both layers; APEX and Core. At the APEX, the partition structure holds the partition status structure and resources tables (processes, interpartition and intrapartition communication). At the Core layer, the partition structure holds information related to its Time Manager, Health Monitor, process scheduler, and further information related to the partition scheduling.

4.3.1.2. Process Management Services

Process services as specified by ARINC 653 standard provide functionalities to control and monitor processes execution. The standard also defines a process scheduling algorithm as being a classic preemptive priority-based scheduler: higher priority processes (that becomes eligible to run) can interrupt the execution of lower priority processes. When several processes have the same priority, the oldest process is selected to execute. The process keeps control of the processor until a rescheduling event takes place, either caused by a direct request of the running process or by any partition internal event.

Processes are scheduled according to its priority and state. A process can be in one of the following states:

- **Dormant** - process ineligible to receive resources. A process is in the dormant state before it is started and after it is terminated (or stopped).
- **Ready** - process eligible for scheduling. A process is in the ready state if it is able to be executed.
- **Running** - process currently executing on the processor. A process is in the running state if it is the current process in execution. Only one process can be executing at any time.
- **Waiting** - process is not allowed to receive the processor until a particular event occurs.

APEX Functions:

`GET_PROCESS_ID` – Allows a process to obtain a process identifier by specifying the process name.

`GET_PROCESS_STATUS` – Returns the current status of the specified process. The current operating status of each of the individual processes of a partition is available to all processes within that partition.

`CREATE_PROCESS` – Creates a process and returns an identifier that denotes the created process. Partitions can create as many processes as the pre-allocated memory space supports. Consistency among process parameters and partition parameters are verified. A process is created in DORMANT state.

`SET_PRIORITY` – Changes a process current priority. The process is placed at the end of the queue with that priority.

`SUSPEND_SELF` – Suspends the execution of the current process if aperiodic, until the RESUME service request is issued or the specified time-out value expires.

`SUSPEND` – Allows the current process to suspend the execution of any aperiodic process except itself, until the suspended process is resumed by another process. If the process is pending in a queue at the time it is suspended, it is not removed from that queue. When it is resumed, it will continue pending unless it has been removed from the queue (either by occurrence of a condition or expiration of a time-out or a reset of the queue) before the end of its suspension. A process may suspend any other process asynchronously.

`RESUME` – Allows the current process to resume another previously suspended process. The resumed process will become ready if it is not waiting on a resource (delay, semaphore, period, event, message). A periodic process cannot be suspended, so it can not be resumed.

`STOP_SELF` – Allows the current process to stop itself. This service should not be called when the partition is in the WARM_START or the COLD_START mode. In this case, the behaviour of this service is not defined.

No return code is returned to the requesting process procedure.

`STOP` – Makes a process ineligible for processor resources until another process issues the START service request. This procedure allows the current process to abort the execution of any process except itself. When a process aborts another process that is currently pending in a queue, the aborted process is removed from the queue.

`START` – Initializes all attributes of a process to their default values, resets the runtime stack of the process. If the partition is in the NORMAL mode, the process deadline expiration time and next release point are calculated.

Services that causes a process to change its state, very often depend on the process previous condition; a process cannot be resumed if it is blocked on a resource. The process state can indicate that this process is in WAITING, but it does not indicate the reason for that. AMOBA process structures carry all the information required for the APEX services to identify a process current condition: resource references indicate the resources in use by the process (the reason for the process to be WAITING; the process is blocked on the resource), a time variable stores events programmed by the process within the Time Manager, flags are used to indicate the cancelation or expiration of a time service.

4.3.1.2.1. AMOBA-APEX Process Scheduler

Fundamental differences between POSIX scheduling policy and ARINC 653 expected scheduling behaviour compromises the compatibility between the two standards. AMOBA-APEX process scheduler objective is to implement a process scheduler for the AMOBA-APEX that is safe and ARINC 653 compliant.

The AMOBA scheduler implementation works as a wrapper on top of the OS native scheduler to provide the desired ARINC653 compliant scheduling behaviour. It ensures ARINC653 defined scheduling policy and at the same time gives AMOBA the control over process pre-emption. The resulting consequence is that this scheduler gives AMOBA independence of POSIX definitions emulating an ARINC653 OS at the partition level.

In the context of the AMOBA-APEX process scheduler, each process is represented by a process structure within an array correspondent to processes states. A process is in one of the following lists:

- DORMANT
- WAITING
- READY
- RUNNING

All AMOBA processes running at user application space assumes one of the three priorities:

```
not_running: 1
idle:        2
running      3
```

At runtime, there is one process with priority 3 as there can only be one process running at a time, and one process with priority 2, the idle process. All the other processes have priority 1. These priorities are used by the AMOBA-APEX process scheduler to intercept the operation of the native OS scheduler. But ARINC 653 defined current priority is the priority actually used to determine the order in which processes that are eligible to run (READY) gain access to the processor.

The idle process is unique within AMOBA, it runs to consume the remaining time of a partition with no process eligible to execute. All the partitions share the same idle process.

The invocation of the process scheduler (`ask_for_schedule`) causes the selection of a new running process. The process scheduler verifies if there is any process in the READY queue eligible to pre-empt the current running process. Usually, the scheduler is invoked after modifications on some processes state or priority, as part of an APEX service implementation or as a consequence of time related event or health monitor action.

Any modification in process state or priority is accomplished through the structure `amoba_apx_sched_change_t` filled through any of the following operations:

```
Type amoba_apx_sched_change_t is record
  ID_TYPE process_id;
  PRIORITY_TYPE old_prio;
  PRIORITY_TYPE new_prio;
  PROCESS_STATE_TYPE old_state;
  PROCESS_STATE_TYPE new_state;
```

void `amoba_start_p(process_ptr);`

This operation is invoked by the `START`, `DELAYED_START`, `SET_PARTITION_MODE` operations to move the process referenced by `process_ptr` to the `WAITING` or `READY` state queue (according to the partition mode and process timing behaviour – periodic or aperiodic).

void `amoba_set_ready(/*in */ PROCESS_TYPE process_ptr,`

`/*in */ CONTEXT_TYPE context);`

This operation is invoked to set a process pointed by `process_ptr` to the READY state. `amoba_set_ready` uses the `context` variable to verify the cause of the process state change. Based on the `context` value, `amoba_set_ready` verifies if the process can be eligible to run; if the process is not suspended or blocked on any resource. The operation fills the `amoba_apx_sched_change_t` structure with the `process_id`, the `new_state` set to `READY`, `old_prio` and `new_prio` set to the process current priority and invokes `amoba_setchange_apxProcessScheduler` using the `amoba_apx_sched_change_t` structure as parameter. It also modifies the process state (`PROCESS_STATUS.PROCESS_STATE`) to READY and the process suspended mode to `AX_NOT_SUSPENDED`.

The `context` variable indicates the cause for a process to be set to READY state, it can assume any of the values:

- RESUME_CTX – used by APEX service RESUME
- TIMEOUT_CTX – used when the TIME_OUT the process was expecting has already expired
- UNBLOCKED_CTX – used for processes that leave the WAITING state because are no longer blocked by a resource condition (ex: buffer full, empty blackboard, etc.)
- START_CTX – used by operation `amoba_start_p`

`amoba_set_ready` operation uses the context value because there are different verifications to be made according to the condition that led the process to be set to READY state. When a process is blocked (put in WAITING because it requires an unavailable resource), for instance, it can be set back to the READY state by `TIME_OUT` expiration (`TIMEOUT_CTX`), but not because of a `RESUME` service, (context value `RESUME_CTX`). Most verifications, however, are related to the process suspended mode (suspended, not suspended, self_suspended); a process that was not suspended cannot suffer a RESUME service even if it is in WAITING state.

`void` `amoba_set_dormant(/*in */ PROCESS_TYPE *process_ptr);`

This operation is invoked to set a process pointed by `process_ptr` to the DORMANT state. `amoba_set_dormant` fills the `amoba_apx_sched_change_t` structure with the `process_id`, the `new_state` set to `DORMANT`, `old_prio` and `new_prio` set to the process current priority and invokes `setchange_apxProcessScheduler` using the `amoba_apx_sched_change_t` structure as parameter. It also modifies the process structure state field at the APEX (`PROCESS_STATUS.PROCESS_STATE`) to DORMANT.

`void` `amoba_set_waiting(/*in */ PROCESS_TYPE *process_ptr);`

This operation is invoked to set a process pointed by `process_ptr` to the WAITING state. `amoba_set_waiting` fills the `amoba_apx_sched_change_t` structure with the `process_id`, the `new_state` set to `WAITING`, `old_prio` and `new_prio` set to the process current priority and invokes `setchange_apxProcessScheduler` using the `amoba_apx_sched_change_t` structure as parameter.

It also modifies the process structure state field at the APEX (PROCESS_STATUS.PROCESS_STATE) to WAITING.

```
int amoba_set_priority(/*in */ PROCESS_TYPE *process_ptr,
                       /*in */ PRIORITY_TYPE priority);
```

This operation is invoked to set a process pointed by process_ptr to the priority indicated by the parameter priority. amoba_set_priority fills the amoba_apx_sched_change_t structure with the process_id, the new_state and old_state set to the process current state old_prio with process current priority and new_prio set to the process requested priority (priority) and invokes setchange_apxProcessScheduler using the amoba_apx_sched_change_t structure as parameter. It also modifies the process structure status field at the APEX (PROCESS_STATUS.CURRENT_PRIORITY) to the priority requested.

```
void amoba_setchange_apxProcessScheduler(/*in*/
amoba_apx_sched_change_t apx_sched_change);
```

This operation causes the update of the process referenced within the apx_sched_change structure on the process scheduler queues. When a process state change is requested, the operation searches the process in the queue indicated by the apx_sched_change.old_state field value and inserts the process at the end of the queue indicated by apx_sched_change.new_state.

If a priority change was requested, this operation will simply move the process to the last position of its own queue (no process state modification). The result of this action is that among all the processes in the same queue with the same priority, this is the last process to be selected to run. As a consequence, if this process is running, it can be pre-empted by any other process in the queue with the same priority (new_priority).

The AMOBA APEX process scheduler is invoked by ask_for_schedule operation. This operation causes a signal to be sent to the scheduler thread. The scheduler thread is created at partition initialization and waits for a signal in a loop. When the signal is received, it searches in the READY queue for the highest priority process and put the process in RUNNING (set its priority to 3). The immediate execution of the scheduler thread is ensured by its priority within the partition; the process scheduler is the highest priority thread within a partition.

ask_for_schedule also verifies the partition lock level, it will not pre-empt a running process if the lock level is higher than 0. Whenever a LOCK_PREEMPTION service is invoked, the partition lock level is incremented and the process scheduler apx_preemption_locked variable is set to TRUE, LOCK_LEVEL is decremented by APEX service UNLOCK_PREEMPTION. When the partition LOCK_LEVEL reaches 0, the apx_preemption_locked variable is set to FALSE, and processes may be preempted.

4.3.1.3. Time Management Services

The Time Management module comprises a set of services that provides mechanisms to control periodic and aperiodic processes as well as to get the present time and to pause the processing of a process.

Next follows some considerations defined on the ARINC653 standard regarding the Time Management Services accomplishment.

- Time is unique and independent of partition execution within a core module. All time values or capacities are related to this unique time and are not relative to any partition execution.
- Time capacity is associated with each process. It represents the response time given to the process for satisfying its processing requirements.
- As long as the process performs its entire processing without using its whole time capacity, the deadline is met. If its processing requires more than the time capacity, the deadline is missed.
- A time-out (delay or deadline) can expire outside the partition window. Response actions for that will run at the beginning of the next partition window.
-

The Time Management Services available through APEX interface are:

`TIMED_WAIT` – suspends execution of the requesting process for a given period of time.

`PERIODIC_WAIT` – suspends execution of the requesting periodic process until the next release point.

`GET_TIME` – get the value of the system time.

`REPLENISH` – updates the deadline of the requesting process with a specified value.

AMOBA-APEX accomplishes the Time Management Services using the Time Manager from AMOBA-Core layer. This last module will be described on section 4.3.2.1 Time Manager, essentially it is responsible for provide all time services needed.

4.3.1.4. Interpartition Communication - Sampling Ports Services

Sampling ports are non-blocking mechanisms for partition communication. This object allows processes from different partitions to exchange one message of variable size up to a given upper size limit. Processes can always send messages, older messages are overwritten by new ones. At the destination side, a sampling message uses a validity attribute; the message age (defines how long a message has been at the destination port) cannot be greater than the port refresh period specified at configuration. At message reception (`RECEIVE_SAMPLING_MESSAGE`), the message validity is verified; invalid messages cannot be copied to the process.

The actual message exchange from one partition to the other is core software operation. For the ports implementation in AMOBA each partition is assigned one transportation thread.

The transportation thread runs in a loop waiting for a signal. When the signal is received, it copies the message from the buffer of the source partition to the destination partition buffer.

The transportation thread runs with a priority that is higher than the user processes within its partition – but lower than the partition scheduler priority. It maintains the address of the message buffer structures of the source and destination partitions.

APEX Functions:

`CREATE_SAMPLING_PORT` – Creates and initializes a sampling port structure and adds it to the partition sampling port table.

`WRITE_SAMPLING_MESSAGE` – copies the message from the operation parameter to the sampling port buffer and signals the transportation thread.

`READ_SAMPLING_MESSAGE` – copies the message from the destination port buffer to the address within the operation parameter.

`GET_SAMPLING_PORT_ID` – finds the sampling port at the partition queuing port table given its name and returns the sampling port id.

`GET_SAMPLING_PORT_STATUS` – finds the sampling port at the partition queuing port table given its name and returns the sampling port status.

4.3.1.5. Interpartition Communication - Queuing Ports Services

Queuing ports allow partitions to exchange messages; they are able to store a certain number of ordered messages. The order in which messages are sent from a source port is the same they are received in the destination port (FIFO).

Queuing ports are blocking resources, processes attempting to send a message through a full channel of a queuing port are blocked and a reference for these processes are maintained the queuing port waiting processes list. Processes attempting to receive a message from an empty queuing port buffer are also blocked.

The buffer space used to store messages to be sent is managed by the source partition as well as the time required by the operation. The actual message transference from one partition to the other is core software operation. For the ports implementation in AMOBA each partition is assigned one transportation thread.

The transportation thread runs in a loop waiting for a signal. When the signal is received, it searches for a new message in the buffer of the source partition. For each new message found, it copies the message to the destination partition buffer (when there is enough space available). Messages may be split in different segments, all segments are transferred to the destination partition but it is application responsibility to assemble the message segments into one single message.

The transportation thread runs with a priority that is higher than the user processes within its partition – but lower than the partition scheduler priority. It maintains the address of the message buffer structures of the source and destination partitions.

The queuing port data structure stores its attributes and the starting address of the message buffer. Each message slot has a state indicator that can assume one of the following values:

- FREE – message slot is empty and is ready to receive a new message.

- BUSY – message slot is being read or written by others.
- FILLED – message slot has data content.

The transportation thread searches the source partition message buffer list for the first slot with FILLED state. While FILLED state slots are found at the source partition message buffer the transportation thread performs the following steps: (i) searches the destination partition message buffer for a FREE state slot; (ii) set this slot to BUSY; (iii) copy the message from the source partition message buffer; (iv) sets the source partition message buffer slot to FREE.

APEX Functions:

`CREATE_QUEUING_PORT` – Creates and initializes a Queuing Port structure and adds it to the partition Queuing port table.

`SEND_QUEUING_MESSAGE` – Searches the buffer for a message slot with state FREE. When it is found, the process sets the slot state to busy, copies the message to the slot, sets the slot state to FILLED and signals the transportation thread. If no FREE segment is found, the process may block (if it uses a TIME_OUT parameter higher than 0) and be put in the port waiting processes list.

`RECEIVE_QUEUING_MESSAGE` – Searches the port buffer for a slot with a FILLED state. When it is found, the process sets the state to BUSY, copies the message from the slot, sets the slot state to FREE and returns successfully. If no FILLED segment is found, the process may block (if it uses a TIME_OUT parameter higher than 0) and be put in the port waiting processes list.

`GET_QUEUING_PORT_ID` – finds the queuing port at the partition queuing port table given its name and returns the queuing port id.

`GET_QUEUING_PORT_STATUS` – finds the queuing port at the partition queuing port table given its name and returns the queuing port status.

4.3.1.6. Intrapartition Communication – Buffer Services

ARINC buffers are communication structures used by process that share the same partition. Different from Blackboards, buffers have the capability to store various messages of different sizes. Processes attempting to read from an empty buffer are blocked as well as processes that try to write to a buffer that has already reached its capacity. Messages are stored at the buffer when a process writes the message there and are removed once the message is read. No message is lost or overwritten.

AMOBA buffers are implemented through a shared list of structures. Because the buffer itself may have a variable size (defined at buffer creation) as well as the messages that it can store (defined at runtime), each buffer is a dynamic array of structures. The buffer message structure stores the message itself and a message size field.

In AMOBA the processes that are blocked on buffer operations are put in a `WAITING_PROCESSES` list. The list stores processes information according to different disciplines; FIFO or PRIORITY. The queuing discipline defines the order in which these processes must be set to the READY state once the buffer state changes (no longer full or no longer empty).

`TIME_OUT` values are used by `SEND_BUFFER` and `RECEIVE_BUFFER` operations. This time value indicates the amount of time the process is to be blocked on the operation in case the buffer is full (in the case of `SEND_BUFFER` operation) or empty (in the case of `RECEIVE_BUFFER` operation). Time outs are accomplished in AMOBA by the Time Manager. A process blocked on `READ_BUFFER` operation, for instance, could be set to the READY state again when this `TIME_OUT` expires or when another process put a message in the buffer.

Whenever a process is put into the READY state again by the expiration of a `TIME_OUT`, it returns with a return code (`TIMED_OUT`), which also indicates that the `READ_BUFFER` operation was not accomplished (no message is returned to the process). When the process is put into READY state because the buffer condition changed, the operation on which the process was blocked is accomplished.

APEX Functions:

`CREATE_BUFFER` – Initializes a buffer structure and include it to the buffer table within the current partition.

`SEND_BUFFER` – creates a buffer message type structure and include it to the buffer message list in FIFO order. It blocks processes that attempts to send messages to a full buffer. The process is moved to the WAITING state and a reference to it is added to the buffer waiting process list.

`RECEIVE_BUFFER` – removes a buffer message node from the buffer message list and returns the message recovered from it. It blocks processes that attempts to receive messages from an empty buffer. The process is moved to the WAITING state and a reference to it is added to the buffer waiting process list.

`GET_BUFFER_ID` – finds the buffer at the partition buffer table given its name and returns the buffer id.

`GET_BUFFER_STATUS` – finds the buffer at the partition buffer table using its id and return the buffer status (`BUFFER_STATUS_TYPE`).

4.3.1.7. Intrapartition Communication – Blackboard Services

Blackboard services as specified on ARINC 653 standard provide functionalities for intrapartition communication. In contrast to buffer, blackboard does not support a message queue, when a message is displayed it overwrites the previous one. This unique shared message can be cleared at any time by any process in the same partition.

If a message is available at the blackboard, any process can read it immediately, messages can be read any number of times, and are only removed by the specific operation `CLEAR_BLACKBOARD`. On the standard ARINC653 is foreseen that one process blocks only when attempts to read from an empty blackboard. In this case, a `TIME_OUT` parameter can be provided to indicate for how long this process could be blocked. The process leaves the `WAITING_PROCESSES` queue when the `TIME_OUT` parameter expires or when other process displays a message in the blackboard (whichever occurs first).

On the occurrence of a DISPLAY_BLACKBOARD operation, all processes blocked on an empty blackboard are automatically unblocked (set to READY) at once. Therefore, no queuing discipline is required for this mechanism.

In AMOBA the blackboard is implemented by a unique variable shared by the processes and protected by a mutex. This variable allows different sizes of messages to be written in the blackboard.

APEX Functions:

CREATE_BLACKBOARD – Initializes a blackboard structure and include it to the blackboard table within the current partition.

READ_BLACKBOARD – copy the message within the blackboard found by the provided blackboard id. It blocks processes that attempts to read messages from an empty blackboard. The process is moved to the WAITING state and a reference to it is added to the blackboard waiting process list.

DISPLAY_BLACKBOARD – copy a message to the shared variable in the structure of the blackboard and set its EMPTY indicator to occupied.

CLEAR_BLACKBOARD – remove the message at the shared variable in the structure of the blackboard and set its EMPTY indicator to empty.

GET_BLACKBOARD_ID – finds the blackboard at the partition blackboard table given its name and returns the blackboard id.

GET_BLACKBOARD_STATUS – finds the blackboard at the partition blackboard table given its name and returns the blackboard status.

4.3.1.8. Intrapartition Communication – Semaphore Services

ARINC semaphores as well as POSIX semaphores are used to synchronize processes that share the same resources. The semaphore value indicates the number of resources available.

Processes decrement one unit from the value of a semaphore when they gain access to the resource and increment the value again one unit when it releases the resource; the semaphore is a counter indicating the number of resources still available. The possible operations on a semaphore are WAIT_SEMAPHORE (decrements its value one unit) and SIGNAL_SEMAPHORE (increment its value one unit).

When a process attempts to WAIT on a semaphore that is already zero, it is blocked and put in a WAITING_PROCESSES queue. The processes in the WAITING_PROCESSES queue are ordered by FIFO or by priority order. The queuing discipline is defined at the semaphore creation.

Processes in the WAITING_PROCESSES queue are set back to READY state (according to the WAITING_PROCESSES queuing discipline) when a TIME_OUT expires or when the semaphore condition change (its counter is no longer 0), whichever occurs first.

The WAIT_SEMAPHORE operation as most of ARINC defined blocking operations are invoked with a TIME_OUT parameter. This parameter indicates the amount of time this process is to be blocked (if it is the case) on the semaphore. When the TIME_OUT expires while the process is still blocked on the resource, the process is set back to READY state (unless it was suspended).

The ARINC653 Semaphores, however, have different behaviour; they can be used by a process without actually blocking it. That means that after decrementing a semaphore, the process can still access other resources, including any number of other semaphores. If this process is then stopped, it must be possible for the `STOP` operation to cause the process to release all the resources it is holding.

Another characteristic of semaphores that must be accounted is that the process which executes a WAIT operation on a semaphore must be the process to SIGNAL the semaphore when it is done using it. Each process must keep track of WAIT and SIGNAL operations performed on semaphores. AMOBA processes maintain a list of pending semaphores; semaphores for which the process has requested a WAIT operation but not a SIGNAL operation yet. The pending semaphore list is an array in which each node stores a reference to a semaphore and a counter of wait operations. The counter is incremented whenever the process performs a WAIT on the semaphore and decremented when a SIGNAL is issued on the same resource. When the counter reaches 0, the semaphore node is removed from the pending semaphore list, the semaphore is no longer being used by the process.

The semaphore counter in AMOBA is APEX shared variable protected by a mutex.

APEX Functions:

`CREATE_SEMAPHORE –` Initializes a semaphore structure and include it to the semaphore table within the current partition.

`WAIT_SEMAPHORE –` finds the semaphore in the partition semaphore table using its id. Decrement the semaphore value when it is higher than 0 and put a reference to the semaphore in the process structure (pending semaphore field), incrementing its pending semaphore counter. It blocks processes that attempts to decrement a semaphore that is already 0. The process is moved to the WAITING state and a reference to this semaphore is added into the process structure.

`SIGNAL_SEMAPHORE –` finds the semaphore in the partition semaphore table using its id. Increment the semaphore value when it is lower than its maximum value and decrement the process pending semaphore counter (if the counter reaches value equal to 0, it removes the semaphore reference from the process structure). It will put the first waiting process to READY when this process exists (removing the semaphore reference from its structure).

`GET_SEMAPHORE_ID –` finds the semaphore at the partition semaphore table given its name and returns the semaphore id.

`GET_SEMAPHORE_STATUS –` finds the semaphore at the partition semaphore table given its name and returns the semaphore status.

4.3.1.9. Intrapartition Communication – Event Services

Events, as defined on ARINC 653, are meant to signal a condition. An event will limit the progress of a process (which has called `WAIT_EVENT`) until its value becomes true (event is set). When the condition is unset the process shall block on it, waiting for the condition to be set. The event usage is similar to POSIX mutex, an event can only assume one of two possible values: UP or DOWN. An event is a synchronization object for processes.

Similar to other intrapartition mechanisms, the blocking operation of an event, `WAIT_EVENT` can have a `TIME_OUT` associated to it. The process is unblocked when the event is set or when the `TIME_OUT` expires (whichever occurs first).

Events values are modified by `SET_EVENT` and `RESET_EVENT` operations. By invoking `WAIT_EVENT` operation, a process condition its execution to the event state, it blocks on the event if its state is DOWN (otherwise the process continue normal execution).

The events themselves are simple variables shared among all the threads that implement processes accessing these events. Just as all the other AMOBA resources, each event is protected by a mutex.

Basically each Event object defined within AMOBA has a status and a waiting queue associated.

Status indicates whenever a certain Event is UP or DOWN. If the status is DOWN and WAIT_EVENT is called by a given process, that process goes to waiting state and is placed in the event's waiting queue. On the other hand if status is UP and WAIT_EVENT is called nothing happens to the calling process.

Waiting queue holds all blocked processes for a given event object, in order to know which are the processes that shall be resumed when a SET_EVENT occurs.

APEX Functions:

`CREATE_EVENT` – Initializes an event structure and include it to the event table within the current partition.

`SET_EVENT` – finds the event by its id at the partition event table and updates its value to UP. Moves all waiting processes to READY state and removes its reference from these processes structure.

`RESET_EVENT` - finds the event by its id at the partition event table and updates its value to DOWN.

`WAIT_EVENT` – finds the event by its id at the partition event table. If the event value is set to DOWN, move the calling process to the waiting state and put the reference to the event in the process structure.

`GET_EVENT_ID`– finds the event at the partition event table given its name and returns the event id.

`GET_EVENT_STATUS` – finds the event at the partition event table given its name and returns the event status.

4.3.1.10. Health Monitoring

The AMOBA Health Monitor provides APEX services for user applications related to the detection and treatment of application errors. One of the services provided allows user applications to create a process error handler specific for the user application processes. The process error handler runs user code but is a special kind of process. It has the highest priority among user processes and is not accessible by other processes (it is not possible for other process to get the error handler id or to invoke other APEX services on it). The process error handler can stop a partition in case of failures and can pre-empt the other processes even when pre-emption is disabled.

This error handler process remains in WAITING state until an `RAISE_APPLICATION_ERROR` service is issued by a user process (or a `raise_error` is issued from within AMOBA code). The error handler is activated through the process scheduler and must release the processor voluntarily.

Besides activating the error handler process, `RAISE_APPLICATION_ERROR` service also collect the error information within an `ERROR_STATUS` structure format. This information is stored by the Health Monitor and is later recovered by the error handler through the `GET_ERROR_STATUS` service.

APEX Functions:

`REPORT_APPLICATION_MESSAGE` – Allows an application process to send an error message to the Health Monitor. The message can be logged to be treated later. However, as the parameter used by this service does not include an error code, if the messages transmitted are to be treated, the message structure is implementation dependent.

`CREATE_ERROR_HANDLER` – Allows the creation of a process error handler through an entry point provided as parameter for the service.

`GET_ERROR_STATUS` – It is used by the process error handler to recover information about the error that originated its execution. This service returns an error status structure containing the information required.

`RAISE_APPLICATION_ERROR` – Allows an application process to raise an error. The error code (ARINC653 defined) used as parameter for this service determines the error response by the error handler process. The immediate action of this service is to activate the process error handler by invoking the AMOBA APEX process scheduler. The message passed as parameter of the service is also stored by the Health Monitor to be recovered later by the process error handler. When invoked by an error handler process, this service activates the partition error handler.

4.3.2. AMOBA-Core layer design

The AMOBA-Core layer is similar to the OS kernel component because like defined on the ARINC653 architecture this layer also provides an abstraction layer that avoids the direct dependency between the AMOBA simulator and the host OS. With this approach it is possible to achieve more portability for AMOBA simulator.

To accomplish an OS independent environment, the relationship between AMOBA-Core sub-modules and the OS must be as much abstract as possible. Nowadays most of the RTOS are POSIX (Portable Operating System Interface) compliant. Thus, AMOBA-Core sub-modules that access OS services are implemented as POSIX-based in order to reach a larger number of compatible Operating Systems. This layer could be extended to support more than POSIX compliant OSes hypothetically changes only need to be done on the AMOBA-Core layer. Although possible this issue is not concerned on AMOBA scope.

The AMOBA-Core layer concept described here is supposed to provide an adaptation abstraction layer to promote services in an OS independent environment.

As well as the entire simulator, the AMOBA-Core Layer follows a modular approach. This Layer is composed by several Managers each one has a specific role within the AMOBA-Core services. These managers will be described throughout this section.

4.3.2.1. Time Manager

Time manager is an AMOBA-CORE component responsible to provide all the time services.

4.3.2.1.1. Services Overview

The time manager provides time services to the AMOBA-APEX layer. The AMOBA-APEX layer needs two main services:

- wait services: allow a process to request a limited amount of time in WAITING state. This service blocks a process for the specified amount of time, moving it to the WAITING queue of APEX scheduler.
- deadline control: monitor processes that have a deadline attached to it and raise an error when this deadline is missed. This service does not modify the process state, but can invoke an error handler process (in case it exists) after the deadline expiration.

Both services can be cancelled; a deadline is cancelled by "deadline control" whenever the release point of a periodic process terminates before the deadline is reached. The wait service is cancelled in response to an event – a resource, the process was waiting for, becomes available or the process is stopped by another one. Wait services are requested by user while deadlines monitoring are services automatically managed by AMOBA.

4.3.2.1.2. Time Concepts

Granularity – is the value that establishes the size of the clock grain.

Accuracy – is the capacity to hit right on target. Or: is the capacity of being exact. With high accuracy, the probability to strike exact on target is much higher than with lower accuracy.

Precision – is the capacity to repeatedly hit on the same place this doesn't means that has to be on target.

Since Time Manager provides time related services, it has to cope with some specific temporal requirements to supply to the consumers a good service, or in other words to assume a commitment with consumers. Because time never stops and is a vacant field, it is necessary to establish rules or assumptions in order to define one standard environment to achieve coherence of component services. These assumptions are commitments that could also be seen as an agreement between service producer (Time Manager) and its consumers to ensure that every service works as expected.

The main reason for the existence of a commitment is to provide a well known behaviour to all Time Manager consumers.

As such and in summary, commitments that Time Manager assumes are:

- a deterministic environment;
- guarantee a known level of precision and accuracy;
- a specific value for granularity.

System granularity is a commitment that establishes the allowed accuracy and precision level. Granularity is strongly related with accuracy and precision because changes on granularity can result on reduction or increase of the accuracy or precision level allowed within the system.

Accuracy is extremely important to the Time Manager because it allows a guarantee that there will be a limited time interval for the service response. This interval stipulates the maximum delay allowed for a request response.

Example:

Consider the following scenario: If the granularity stipulated is milliseconds, it implies that the best situation allowed for schedule an event within the system is limited to one millisecond. If an event is scheduled to expire in [3s:10ms:12us], the only guarantee is that it will expire in [3s:10ms:~us]. (Where ~ can be any value between 0 and 999)

4.3.2.1.3. Functionality Description

The wait service is not realized entirely by the Time Manager, the blocking behaviour demands use of the APEX process scheduler operations. The implementation of the APEX service TIMED_WAIT is accomplished by a request to the Time Manager to schedule the wait period of time and a request to the APEX process scheduler to actually put the process into WAITING. However, when the programmed wait period expires, it is the Time Manager responsible to invoke the APEX process scheduler to set the process back to READY state.

There is one Time Manager per partition. This Time Manager is suspended and resumed together with all the other partition resources. If events from one given partition expire when other partition is running, those events will be handled when that partition is resumed again.

Operations to create, suspend and resume the Time Manager (described later in this section) are embedded in the partition management operation. Time services provided by the Time Manager are always valid only in the current partition context.

4.3.2.1.4. Functionality Details

Like the process scheduler, the Time Manager maintains representations of processes, process descriptor structures defined by `as_tm_proc_desc_t` listed bellow. These descriptors are organized in two queues, one related to deadlines (PER) and the other related to wait periods (ALL) of time (wait service). This second list stores descriptors of all processes in the partition, including the periodic ones (process with a deadline). The duplication of periodic process representation in the Time Manager is explained later in this section.

```
Type as_tm_proc_desc_t is record
  PROCESS_TYPE * process_ptr;
  char  TYPE;
  struct as_tm_proc_desc_st *nxt;
```

Time services are programmed in the Time Manager into a linked list of events. Each node in the list corresponds to a time point for which the Time Manager must perform some action (wake-up a process or raise a deadline miss error). The event list node is specified by the structure as_tm_evt_t listed bellow, it contains a field defining the time at which the event is programmed to happen, and pointers to manage a list of processes related to this event. Because many processes can have different events associated with the same point in time, the event list is a list of lists. Each node represents a point in time (for which an event was programmed) and also points to the head of a list of processes with a programmed event for that time point.

```
Type struct as_tm_event_t is record
  SYSTEM_TIME_TYPE TIME_POINT;
  as_tm_proc_desc_t *PROCESS_PTR;
  as_tm_proc_desc_t *last;
  as_tm_event_st    *nxt;
```

When an event is programmed, the Time Manager creates an event node with the time point field set to according to the requested service.

```
int as_tm_register_wait(PROCES_TYPE * process_ptr, SYSTEM_TIME_TYPE *p_timeout);
```

This operation registers a wait period for the process referenced by `process_ptr`. It is invoked by TIMED_WAIT, for instance and causes the Time Manager to create node at the event list associated to the time point described by the parameter `p_timout` (in case it does not exist already).The process referenced by `process_ptr` is searched at the list ALL (related to wait service) and its corresponding descriptor is removed from this list and inserted in the event list.

If the wait is not unregistered the process `process_ptr` will be put into the ready state and the scheduler will be invoked when the interval `p_timeout` has expired.

```
int as_tm_unregister_wait(PROCESS_TYPE * process_ptr);
```

This operation unregisters a programmed wait event registered earlier for process `process_ptr`. It removes the process referenced by PROCESS_PTR from the event list.

```
int as_tm_register_deadline(PROCESS_TYPE *process_ptr, SYSTEM_TIME_TYPE *p_deadline);
```

This operation registers a deadline for the process referenced by process_ptr. It causes the Time Manager to create a node at the event list associated with the time point described by the parameter p_deadline (in case it does not exist already). If the deadline is not unregistered the error handler will be invoked when p_deadline has expired.

```
int as_tm_unregister_deadline(PROCESS_TYPE * process_ptr);
```

This operation unregisters a deadline registered earlier for process process_ptr. It removes the process referenced by process_ptr from the event list and put it back to its original queue (PER).

The Time Manager is represented by the structure as_tm_man_desc_t listed bellow; it holds the context required for the partition Time Manager.

```
Type struct as_tm_man_desc_t is record
     as_tm_manager_t handle;
     as_tm_proc_desc_t *all, *all_last;
     as_tm_proc_desc_t *per, *per_last;
     as_tm_evt_desc_t *rsc, *rsc_last;
     as_tm_evt_desc_t *evt;
     int main_prio;
     pthread_t tm_thread;
     pthread_mutex_t tm_mutex;
     int err;
     struct as_tm_man_desc_st *nxt;
```

The operations listed bellow are used to initialize and set the Time Manager context for the running partition.

```
int as_tm_create_manager(as_tm_manager_t *p_manager);
```

This operation creates a new Time Manager and returns an identifier for the Time Manager in `p_manager`. It is used during initialisation in `create_Partition` operation.

```
int as_tm_set_current(as_tm_manager_t p_manager);
```

This operation sets the Time Manager referenced by `p_manager` as the current Time Manager. All time related operations acts on the current Time Manager. It is invoked at the partition initialisation before the invocation of `as_tm_init` and after initialization it is invoked before each call to `as_tm_resume`.

```
int as_tm_init();
```

This operation is called during initialisation once per Time Manager. It initialises the Time Manager structure and resources (processes and event descriptors).

```
int as_tm_suspend();
```

This operation suspends the current Time Manager.

```
int as_tm_resume();
```

This operation resumes the current Time Manager.

```
int as_tm_register_process(PROCESS_TYPE * process_ptr);
```

This operation registers the process referenced by `process_ptr` as internal resource in the time manager. It is invoked by APEX service `CREATE_PROCESS` and causes the Time Manager to initialize an event descriptor structure and add its node in the current Time Manager resources list. This resource list contains event structures uninitialized (they do not have a time point value) correspondent to the processes created within the partition. It is created one event structure per process within the list ALL plus one event structure per process in the list PER. When an event is programmed (a wait service or deadline scheduled), the event descriptor structure is moved from this resource list to the Time Manager event list. And when the event is processed or cancelled, this structure is cleared and put back in the resource list.

The Time Manager runs within a loop, when started, it is programmed to sleep during a predefined time interval. When this time interval expires, the Time Manager searches the event list. If the list is empty, it will sleep again during the pre-defined time interval. Else, Time Manager reads the first process structure data pointed by the first event descriptor and take appropriate action; if the process descriptor type field indicates PER, the Health Monitor raise_error is invoked with the error code of deadline miss. If the process descriptor type field indicates ALL, the Time Manager set the process to the READY state and invokes process scheduling (ask_for_schedule).

Whenever a programmed event is processed by the Time Manager, the event descriptor itself is removed from the event list and returned to the Time Manager resources list. The Time Manager searches the next event in its list and set itself to sleep until this next event time point.

Worst case scenarios within this design are caused by many processes with events programmed to the same time point.

4.3.2.2. Partition Manager

Partition Manager is an AMOBA-CORE component responsible to manage partitions and provide all the related services.

4.3.2.2.1. Services overview

Partition Manager provides two kinds of AMOBA-Core services:

- **"APEX Services" –** is responsible for the association between AMOBA-Core and AMOBA-APEX primitives, so the AMOBA-APEX interface can make available its services in a portable way for the simulator users.
- **"Execution Environment" –** is responsible for providing the internal services needed so AMOBA-Core can schedule partitions.

4.3.2.2.2. Partitioning Concepts

In avionics partitioning means a functional separation of the applications, usually used to prevent any partitioned function from causing a failure in another partitioned function.

An element of a partitioned system is called a partition. A partition is essentially an application contiguous in a single environment, it comprises: data, configuration attributes, etc.

Partitions are scheduled on a fixed, cyclic basis. To assist this cyclic activation, the OS maintains a major time frame of fixed duration, which is periodically repeated throughout the module’s runtime operation. Partitions are activated by allocating one or more partition windows within this major time frame, each partition window being defined by its offset from the start of the major time frame and expected duration. The order of partition activation is defined at configuration time using configuration tables. This provides a deterministic scheduling methodology whereby the partitions are furnished with a predetermined amount of time to access processor resources. Temporal partitioning therefore ensures each partition uninterrupted access to common resources during their assigned time periods.

The major time frame is defined as a multiple of the least common multiple of all partition periods in the module. Each major time frame contains identical partition scheduling windows. The periodic requirement of each partition on the module must be satisfied by the appropriate size and frequency of partition windows within the major time frame.

Each partition has predetermined areas of memory allocated to it. These unique memory spaces are identified based upon the requirements of the individual partitions, and vary in size and access rights. At most, one partition only has write access to any particular area of memory. Memory partitioning is ensured by prohibiting memory accesses (at a minimum, write access) outside of a partition’s defined memory areas.

The ARINC 653 standard definition of partitioning relies on 3 different definitions:

- spatial segregation;
- static configuration at system start up.
- temporal segregation;

Spatial segregation

Spatial segregation is a subject that is not in AMOBA scope, the memory specification and division between partitions is not considered.

Configuration

Configuration of all partitions throughout the whole system is expected to be under the control of the system integrator and maintained with configuration tables. The configuration table for the partition schedule will define the major time frame and describe the order of activation of the partition windows within that major time frame.

The issues regarding configuration are guaranteed by the “Configuration Manager” services. More details about configuration issues and concerning this manager can be found on section “4.3.2.4 – Configuration Manager”.

Temporal partitioning

Temporal partitioning is a time division method to ensure that each partition within the system has uninterrupted access to common resources during their assigned time periods.

As defined on the standard definition, to accomplish temporal segregation some requirements must be assured. These requirements are identified next:

- A scheduling mechanism should be provided whereby the partitions are furnished with a predetermined amount of time defined previously, to access processor resources.
- Access to the “configuration table” information is necessary to determine the partition scheduling plan.
- The order of partition activation is defined at configuration time by the configuration tables.
- There must be a way to withdraw and assign partitions to access processor resources.
- Partition window is the time interval during which a partition gain processor control.
- Each partition is scheduled according to its respective partition window defined on the configuration tables.
- At least one partition window is allocated to each partition during each cycle.
- The scheduling algorithm is predetermined, repetitive with a fixed periodicity, and is configurable by the system integrator only.

4.3.2.2.3. Functionality Description

Regarding the requirements identified on the previous section next follows some answers in order to accomplish partition management.

Partition scheduler is a component of the partition manager, its objective is to accomplish the temporal segregation, i.e., guarantees that each partition has its execution time. To know the instant in which a partition shall switch context with other partition, the partition scheduler must have access to the configuration table. All configuration information is loaded and accessed through the Configuration Manager services which shall be responsible for guarantee all the configuration issues.

All the temporal information acquired by partition scheduler, like time division between partitions execution, are accomplished with services provided by the Time Manager.

On Partition Scheduler initialization, the configuration tables are loaded thru configuration manager facilities after that a schedule plan for partitions is created. Based on schedule plan the Partition Scheduler withdraw and assign partitions to access processor resources. Because each partition has a set of associated processes, the access by the associated processes to processor resources is controlled through process scheduler services.

Partition Manager has a set of services available to provide an interface to control with Partition Scheduler.

Concluding, the Partition Manager to accomplish temporal segregation combines functionalities of configuration, time and process scheduler.

4.3.2.2.4. Functionality Details

Partition Manager is composed by two components:

- Partition Services – its responsibility is to provide an interface to AMOBA-APEX and to manage partition scheduler.
- Partition Schedule – its main purpose is to provide an execution environment capable of scheduling partitions and processes according to the requirements defined in the ARINC653 standard.

-

Partition Services

In this section it is presented this component description including its data structures and operations. Its objective is to provide all partition related operations and maintain the structure definition compatible with the AMOBA-Core services.

Sub-components and Data-structures:

idlePartitionId – is a pre-defined constant value.

Partition data structure definition:

- Partition identifier - unique identifier, is how the partition is identified within the system;
- Process scheduler reference – a reference for the partition's process scheduler;
- Partition state – denotes the state of the partition.
- Time Manager identifier – stores the identifier of the partition Time Manger;
- Health Monitor identifier - stores the identifier of the partition Health Monitor;
- Process Scheduler identifier – stores the identifier of the partition process scheduler;
- Buffers table – stores references and a counter for all the partition buffers;
- Blackboards table - stores references and a counter for all the partition blackboards;
- Semaphores table - stores references and a counter for all the partition semaphores;
- Sampling ports table - stores references and a counter for all the partition sampling ports;
- Queuing port table- stores references and a counter for all the partition queuing ports;

Functions / Operations

Notice that, the following functions should only be called upon the initialization of Partition Scheduler, accomplished by the Partition Scheduler component described below.

create_Partition – given a partition entry point and partition identifier, this function shall create and register a new partition within the system.

```
// 1. process scheduler = initialize_ProcessScheduler();
// 2. run partition entry point
// 3. change partition state ( created )
```

// 4. add partition (process scheduler , partition identifier) – *add partition to partition scheduler*

create_IdlePartition – this function can be called only once and shall create an idle partition.

```
// 1. process scheduler = initialize_ProcessScheduler();
// 2. change partition state ( created )
// 3. add partition ( process scheduler , idlePartitionId ) – add partition to partition scheduler
```

start_Partition – given a partition id this function shall start a partition execution.

```
// 1. partition = get partition ( partition id ) – get partition from partition scheduler
// 2. if(partition.state != created) exit function
// 3. change process scheduler ( partition.process scheduler )
// 4. start process scheduler(); – start current process scheduler
// 5. change partition state ( running )
```

suspend_Partition – given a partition id this function shall suspend a partition execution.

```
// 1. partition = get partition ( partition id ) – get partition from partition scheduler
// 2. if(partition.state != running) exit function
// 3. change process scheduler ( partition.process scheduler )
// 4. suspend process scheduler(); – start current process scheduler
// 5. change partition state ( suspended )
```

resume_Partition – given a partition id this function shall resume a partition execution.

```
// 1. partition = get partition ( partition id ) – get partition from partition scheduler
// 2. if(partition.state != suspended) exit function
// 3. change process scheduler ( partition.process scheduler )
// 4. resume process scheduler(); – start current process scheduler
// 5. change partition state ( running )
```

Partition Scheduler

The Partition Scheduler objective is to accomplish the partition scheduling plan as described in the configuration file. It must run the partitions according to the specified plan making the contexts switch between partitions with no lost of information. The partition schedule is accomplished through the configuration file definition. ARINC 653 does not prevent the user to define partition windows that will not fit into the schedule major time frame. When gaps in the partition schedule are found, AMOBA creates idle partitions to fill the major time frame.

Sub-components and Data-structures:

All partitions – holds all the partitions data that belong to this partition scheduler.

Number of partitions – holds the number of partitions that are within this partition scheduler.

Functions / Operations

initialize_PartitionScheduler – this function initializes all data structures before running the partition scheduler.

```
// 1. number of partitions = 0;
// 2. create idle partition();
// 3. config temporal data = get_ConfigInfo_Temporal()
// 4. for (all partitions within config temporal data)
// 5.   create partition(entrypoint, config temporal data.Partition id);
```

run_PartitionScheduler – this operation performs the schedule defined by the configuration file in a loop.

```
// 1. while(1)
// 2.   config temporal data = get_ConfigInfo_Temporal()
// 3.   for(all partition window within config temporal data)
// 4.         partition window = get partition window( config temporal data ) – get from configuration information the partition window.
// 5.     partition = get partition(partition window.partition id);
// 6.     if(partition.state == created)
// 7.        start partition( partition window.partition id );
// 8.     else
// 9.        resume partition( partition window.partition id );
// 10.    Time Wait( partition window.partition duration );
// 11.    suspend partition( partition window.partition id );
```

4.3.2.3. Health Monitor

4.3.2.3.1. Services Overview

Health Monitor services encapsulate two tasks: error detection and response to failures. According to ARINC specification, the detection of errors is accomplished by several elements; hardware, core software and application. The detection of errors in AMOBA is embedded in both layers: AMOBA Core and AMOBA APEX. User applications can also raise errors through APEX specified interfaces.

4.3.2.3.2. Functionality Description

Health Monitor services are not used only by user processes. As well as the Time Manager, different components of AMOBA may raise errors on behalf of the running application. The Health Monitor must provide functionalities for the user processes through APEX services and for the AMOBA components as well maintaining consistency of stored error information.

The Core part of the Health Monitor provides the services for the APEX layer but it is also invoked whenever internal errors are detected. Error responses from the Health Monitor are triggered through the following operation:

```
int raise_error(PROCESS_ID_TYPE EH_ID);
```

This operation registers the process error (adding an error status structure to the Health Monitor error list) and activates the process error handler (when it was created). The process error handler activation is accomplished through APEX process scheduler operations. If the process error handler is not created, a signal to the partition error handler is sent. This operation is used by APEX service

`RAISE_APPLICATION_ERROR` implementation and is also invoked within the simulator. `raise_error` registers error information at the Health Monitor error list before triggering error responses.

4.3.2.3.3. Functionality Details

Errors information is stored into error status structures in the Health Monitor queue. The error nodes are included by `raise_error` operation and removed by `GET_ERROR_STATUS` service. Only process errors are stored in the list.

The partition error handler entry point is created based in definitions from the configuration file. If a partition error handler definition is provided, AMOBA Health Monitor creates a partition error handler thread. Once created, the AMOBA error handler thread blocks, waiting for a signal. When the signal is received, it executes the partition entry point (the definitions provided through the configuration files).

The operation bellow is used by the partition to initialize the Health Monitor.

```
int initialize_HealthMonitor();
```

This operation creates a new Health Monitor and returns an identifier. This identifier is copied to the partition data control structure.

Operations to manage the partition error handler are listed bellow:

```
int signal_PartitionEH();
```

This operation sends a signal to the partition error handler thread. It is invoked by `RAISE_APPLICATION_ERROR` when the calling process is the process error handler or by `raise_error` when the received error code is a partition error (error code value higher than process error codes).

```
int suspend_PartitionEH();
```

This operation suspends the partition error handler thread. It is invoked by partition scheduler operations and suspends the partition error handler for context switch.

```
int resume_PartitionEH();
```

This operation resumes the partition error handler thread. It is invoked by partition scheduler operations and sets the correct partition error handler to execute according to the running partition.

4.3.2.4. Configuration Manager

Configuration Manager is an AMOBA-CORE component responsible to provide all the configuration services.

4.3.2.4.1. Services Overview

The Configuration Managers main function is to accomplish the “application configuration” AMOBA-Core service. To accomplish that, the Configuration Manager performs two services:

- Load configuration from XML file;
- Provide configuration information.

4.3.2.4.2. ARINC653 Configuration Considerations

System Integrator

The system integrator is the authority responsible for the integration of the applications with the system configuration. On integration process it must define the configuration considering the application needs. When allocation of the partitions to core modules are accomplished by system integrator, the resulting configuration should allow each partition access to its required resources and should ensure that each application's avalability and integrity requirements are satisfied. Therefore, the system integrator must know exactly the timing, memory usage and external interface requirements of each partition to be integrated.

Configuration tables

Configuration tables are static data areas that contains configuration information defined by the system integrator. They cannot be accessed directly by ARINC653 applications, they are used by Core to verify the integrity between application and configuration. There are configuration tables to satisfy different purposes, configuration tables for Initialization, Inter-partition Communication and Health Monitor.

Configuration Tables for Initialization – contains partition identifiers which are resident on the module, the memory requirements of each partition and the number and size of ports required by the partition.

Configuration Tables for Inter-partition Communication – These tables contains inter-partition communication information, i.e., the mapping from the identified port to the next port in the channel. This information can only be known when the configuration of partitions on modules is determined.

Configuration Tables for Health Monitor – The Health Monitor (HM) uses configuration tables to handle each occurring error. These tables are:

- System HM table – defines the level of an error (Module, Partition, Process).
- Module HM table – defines the recovery action (e.g., shut down the module, reset the module) in case of a module error detection.
- Partition HM table – defines the recovery action (e.g., stop the partition, restart the partition in warm or cold mode) in case of a partition error detection.

The configuration data is described using an XML file. XML-Schema is used to define the format of the XML data and is defined in this document.

ARINC653 XML Schema

The ARINC 653 XML-Schema defines the structure of the data needed to specify any ARINC 653 configuration. It should not be expected that an ARINC 653 compliant application require any optional configuration information in order to function in a manner that is complaint with this standard. The configuration file data structure is ilustrated on figure below.

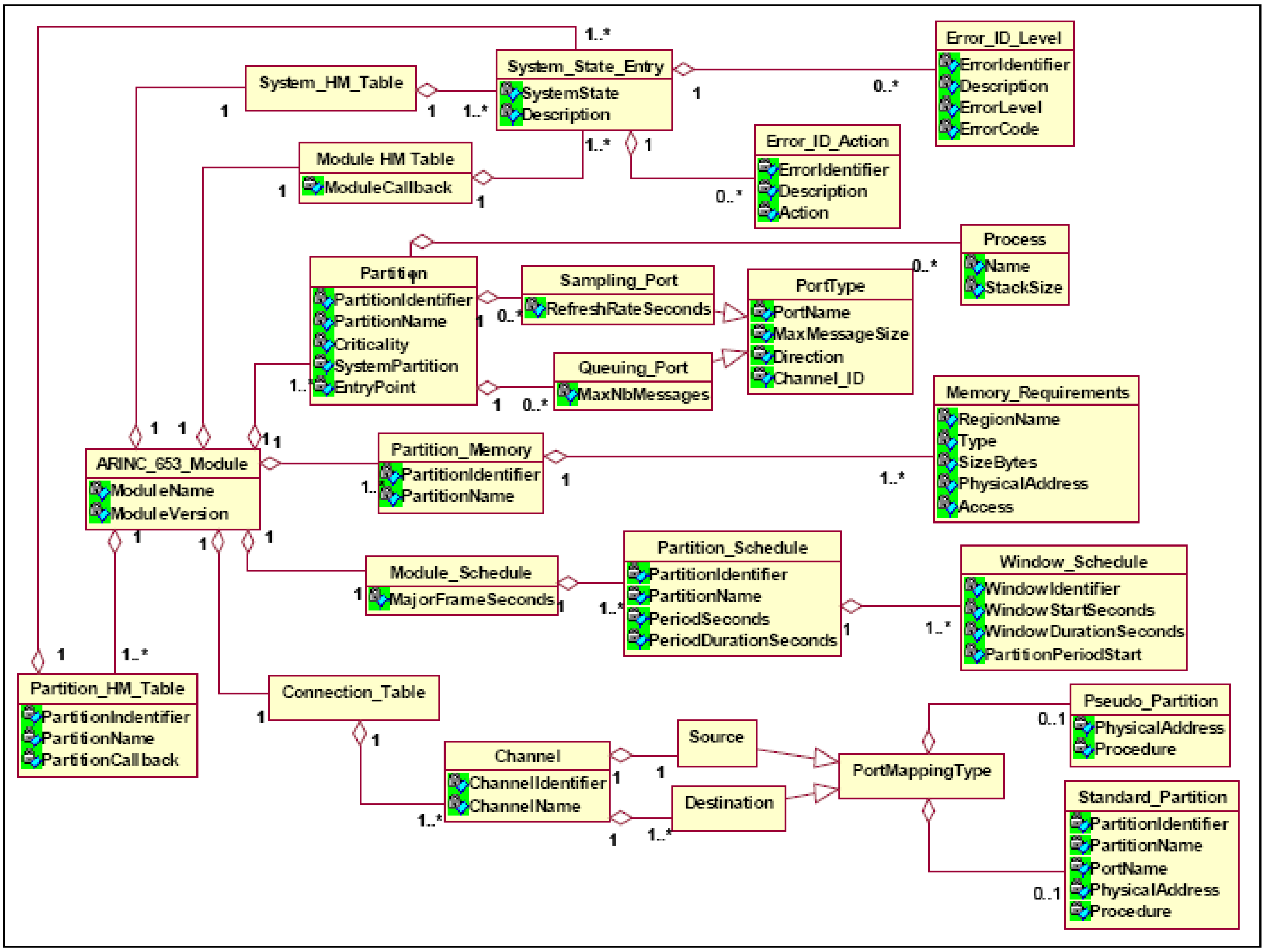


Figure 3 – ARINC653 XML-Schema elements relationships

4.3.2.4.3. Functionality Description

The functionality of the Configuration Manager is here described considering the following issues:

- Internal data structure,
- Loading internal data,
- Provide internal data;

Internal data structure

The internal data structure defined for the Configuration Manager is indentical to the data structure defined on the ARINC653 XML-Schema as ilustrated on the figure 3. Configuration Manager holds the configuration information in this internal data structure in order to provide its services.

Loading internal data

The configuration manager provides a private service to load the internal data structure with the configuration information stored within an XML file. The loading of internal data structure is performed by a XML parser that converts the XML data into understandable data for the configuration manager. After this process the configuration manager contains in its internal data structure all the data translated from the XML configuration file.

Provide internal data

The configuration manager provides a set of services for consulting the configuration information stored on the internal data structure. The data can be provided upon configuration manager's service calls only if the internal data were loaded before.

4.3.2.4.4. Functionality Details

This chapter describes the functionality details: the components and structures that are used on Configuration Manager implementation aiming to accomplish its services.

Sub-components and Data-structures:

- **Internal data structure** – holds the whole configuration information;
- **Temporal exchange data structure** – data structure used to provide configuration temporal information;
- **Communication exchange data structure** – data structure used to provide configuration communication information;
- **Health Monitoring data structure** - data structure used to provide health monitoring communication information;
- **XML Parser** – is the component responsible for getting the data read from xml file.

Functions / Operations

`initialize_ConfigManager` – initializes the configuration manager, prepares internal data structure so Configuration Manager could load the configuration information without any problems.
// 1. Initialize internal data structure

`parseXML_file` – returns the data structure with the information kept within a specified XML file.

`load_ConfigInfo_fromXML` – Parse and load the entire configuration information from XML file into the internal data structures.
// 1. internal data structure = parseXML_file(file name, xsd file)

`get_ConfigInfo_Temporal` – returns the configuration data related to time.
// 1. return getTemporal_Info(internal data structure)

`get_ConfigInfo_Communication` - returns the configuration data related to communication mechanisms.
// 1. return getCommunication_Info(internal data structure)

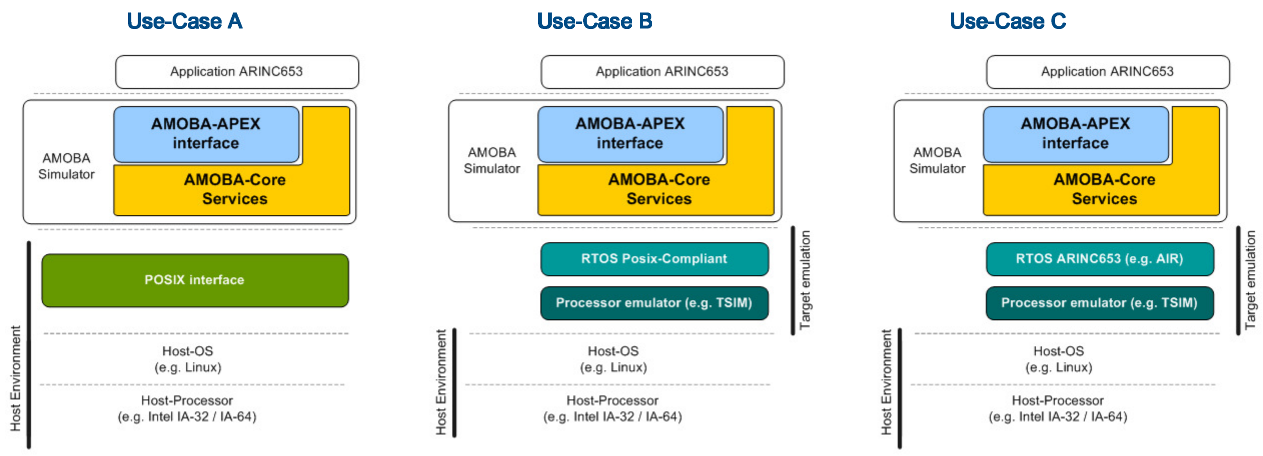
`get_ConfigInfo_HealthMonitoring` - returns the information related to health monitoring configuration.

// 1. return get HealthMonitoring_Info(internal data structure)

Before any operation is requested, the Configuration Manager must be initialized. Therefore it will call `initialize_ConfigManager`. After initialization, the component internal data structure is empty and able to be loaded with the configuration data from the XML file. Only after these two operations the component is able to provide configuration information to consumers.

4.4. Emulation environments

The relation between the AMOBA-Core components and the OS must be as much generic as possible. As such, AMOBA-Core components that access OS services shall be implemented as POSIX-based primitives in order to support a high number of compatible operating systems. Most of the currently available general-purpose operating systems (e.g. Linux) and RTOS kernels (e.g. RTEMS [RTEMS_C 03, RTEMS_POSIX 03], eCos [Massa 02], VxWorks [VxWorks 03], etc...) are POSIX compliant. The improved OS services, minimizing the direct dependency between the AMOBA simulator and the OS, allow more portability to AMOBA simulator.



ARINC 653 Functional Emulation	ARINC 653 Target Emulation	
AMOBA-Core	AMOBA-Core	
✓ full ARINC 653 functional emulation	some ARINC 653 functional emulation	✖ no ARINC 653 functional emulation
Development Environment	Development Environment	
host compiler toolchain POSIX support library	target processor cross-compiler toolchain target processor emulator (e.g. [RTEMS_C 03,VMs 05])	

Figure 4 – AMOBA Simulator and typical use-case scenarios

Different execution environments may be defined for the AMOBA Simulator, as illustrated in Figure 5. An ARINC 653 functional emulation uses the full AMOBA-Core to model the end target system, being executed directly on a POSIX-compliant host environment [Digital 96, POSIX 00], as shown in the Use-Case A of Figure 6.

Emulation environments (Use-Cases B and C of Figure 7) assume the utilization of target processor virtualization. For on-board space systems the utilization of SPARC LEON target platforms is assumed, emulated for instance through TSIM2 [TSIM 08] or SIMULUS/LeonVM [LeonVM 06].

At operating system level, a RTOS kernel can be used to partially provide ARINC 653 run-time environment functions (e.g. process scheduling and management) [A653-Part1, RTEMS_C 03, RTEMS_POSIX 03], as illustrated in the Use-Case B of Figure 8. In addition, the AMOBA Simulator may be used with COTS RTOS implementations of the ARINC 653 specification, such as the technology being developed within the scope of the AIR Project [AIR-Paper 07, Mooney 97]. This is shown in the Use-Case C of Figure 9. The added-value of this approach is the monitoring of the overall system in an emulated, yet very realistic environment.

In target emulated environments, time-related analysis (e.g. verification of timeliness properties and overall system validation in the time domain) may be assisted by additional methods and tools. For example, the method described in [Sched 08] allows to analyse the feasibility of scheduling a given process set, integrating both periodic and sporadic processes, by a priority-based preemptive algorithm and to determine the corresponding worst-case process response times.

The functionality provided by the AMOBA-Core, in the diagram of Figure 10, decreases from Use-Case A to C whereas the functionality of the emulation environment increases in this direction.

5. Concluding remarks and further Issues

As we have seen in the past years the Integrated Modular Avionics has taken a leading role within the aeronautic industry. Since the IMA's successful implementation on AIMS for Boeing 777, the space industry has demonstrate interest in migrate the IMA concept for space domain.

The ARINC 653 is a standard based on the IMA concept that specifies a programming interface for a RTOS (Real-Time Operating System), and, in addition, establishes a particular method for partitioning resources over time and memory.

Within this dissertation document was described the research findings and conclusions ideas reached during the design and implementation of a multi-platform and modular ARINC 653 simulator, called AMOBA, which shall emulate an execution environment for ARINC 653 space applications.

Achieve the IMA-Space concept is a challenge that AMOBA shall help to accomplish. Simulator's portability provides availability around several platforms which makes it easier for space programmers to develop experiences and space applications. Other issue that could lead the IMA-Space in the correct path it is Modularity on AMOBA, a modular approach provides a simpler way to change the simulator modules.

As further work, it is foreseen that the Simulator performance shall be improved. Reduction of the overhead on the modules and implementation monitoring facilities are the main issues to resolve in the future. Monitoring feature will allow user extra capabilities to supervise the applications execution beyond the features available on the ARINC653 standard.

References

[AIMS 97] – B. Aleksa, J. Carter. “Boeing 777 Airplane Information Management System operational experience”, Digital Avionics Systems Conference, 1997. 16th DASC, Oct 1997.

[News-IMA 07] – J. Ramsey. “Integrated Modular Avionics: Less is More. Approaches to IMA will save weight, improve reliability of A380 and B787 avionics”, Avionics Magazine, February 1, 2007.

[A380-IMA 07] – Jean-Bernard Itier. “A380 Integrated Modular Avionics”, ARTIST2 meeting on Integrated Modular Avionics, November 2007 Roma, Italy.

[Dassault-IMA 07] – Thierry Cornilleau. “Dassault Aviation feedbacks on its military and civil IMA applications”, ARTIST2 meeting on Integrated Modular Avionics, November 2007 Roma, Italy.

[A653-Space 05] – N. Diniz and J. Rufino. ARINC 653 in space. In *Proceedings of the DASIA 2005 .DAta Systems In Aerospace. Conference*, Edinburgh, Scotland, June 2005. EUROSPACE.

[IMA_Space 96] – M. Doss.; K. Liebel; S. Lee; K. Calcagni; R. Crum. “Migration of Integrated Modular Avionics to space”, Digital Avionics Systems Conference, 1996. 15th DASC, Oct 1996.

[A653-Part1] – Airlines electronic engineering committee (AEEC), avionics application software standard interface - ARINC specification 653 - part 1 (supplement 2 - required services). ARINC, Inc., 2006.

[A653-Part2] – Airlines electronic engineering committee (AEEC), avionics application software standard interface - ARINC specification 653 - part 2 (extended services). ARINC, Inc., June 2007.

[A651] – Airlines electronic engineering committee (AEEC), design guidance for integrated modular avionics - ARINC specification 651. ARINC, Inc., 1991.

[ESA_TechNote 03] – J-L. Terraillon and K. Hjortnaes. Technical note on on-board software. European Space Technology Harmonisation, Technical Dossier on Mapping, TOSE-2-DOS-1, ESA, February 2003.

[AIR-Paper 07] – 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, June 2007. EUROSPACE.

[AMOBA-Paper 08] – E. Pascoal, J. Rufino, T. Schoofs, 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. EUROSPACE.

[IEEE1003.1-POSIX] – IEEE Std 1003.1 - standard for information technology portable operating system interface (POSIX) system interfaces, 2004.

[ESE_News 06] – Martin Whitbread. “Safety-Critical Systems Lessons from avionics”, Embedded System Engineering (ESE) Magazine, September 2006.

[NASA 88] – James E. Tomayko. “Computers in Spaceflight: The NASA Experience”, Wichita State University NASA Contractor Report CR-182505, March 1988, 417 pages.

[RTEMS_C 03] – RTEMS C User’s Guide. On-Line Applications Research Corporation. September 2003.

[RTEMS_POSIX 03] – RTEMS POSIX API User's Guide. On-Line Applications Research Corporation, August 2003.

[RTEMS_Qualify 05] – J. Seronie-Vivien, C. Cantenot, RTEMS Operating System Qualification for the ERC32 Target, DASIA 2005,. Eurospace Edinburgh.

[NASA_Partition 99] – J. Rushby. “Partitioning in avionics architectures: Requirements, mechanisms and assurance. Technical Report” NASA CR-1999-209347, SRI International, California, USA, June 1999.

[AFDX 04] – ARINC, Aircraft Data Network, part 7 — Avionics Full Duplex Switched Ethernet (AFDX) Network, ARINC 664, draft 3, Sep. 13, 2004

[DO-297 05] – RTCA DO-297. “Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations”, Ago 2005.

[SPIDER 05] – E. J. Ruggiero. Modeling and Control of SPIDER Satellite Components. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA., July 2005.

[Mooney 97] – James D. Mooney. “Bringing Portability to the Software Process”, Tech. rep.,. West Virginia University, Department of Statistics and Computer Science, 1997.

[Mooney 90] – James D. Mooney. “Strategy for Supporting Application Portability”, Tech. rep.,. West Virginia University, Department of Statistics and Computer Science, 1990.

[Parnas 72] – D. L. Parnas. “On the criteria to be used in decomposing systems into modules”, Communications of the ACM, v.15 n.12, p.1053-1058, Dec. 1972

[Modularity 02] – C.Y. Baldwin, K.B. Clark, "The Option Value of Modularity in Design" Harvard Business School, 2002

[Modularity 01] – Sullivan, K., Y. Cai, B. Hallen, W. Griswold. “The Structure and Value of Modularity in Software Design,” Proceedings, ESEC/FSE 2001, ACM Press, pp. 99-108.

[Liedtke 93] – Jochen Liedtke. "Improving IPC by kernel design", 14th ACM Symposium on Operating System Principles, December 1993.

[uKernel 97] – H. Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian. "The performance of µ-kernel-based systems". Proceedings of the sixteenth ACM symposium on Operating systems principles, October 1997.

[WorkplaceOS 98] – B. D. Fleisch, M. A. A. Co, and C. Tan. “Workplace microkernel and OS: A case study. Software: Practice and Experience”, 1998.

[uKernel_RT 05] – C. Kirsch, M. Sanvido, and T. Henzinger. “A programmable microkernel for real-time systems” Proceedings of the First International Conference on Virtual Execution Environments (VEE), ACM Press, 2005.

[PikeOS] – R. Kaiser. “The PikeOS Concept History and Design”, Sysgo whitepaper.

[Liedtke 96] – Jochen Liedtke. "Towards Real Microkernels", Communications of the ACM, September 1996.

[Liedtke 95] – Jochen Liedtke. "On microkernel Construction", 15th ACM symposium on Operating Systems Principles, December 1995.

[Bradley_and_Bershad 93] – Bradley Chen; Brian Bershad. "The Impact of Operating System Structure on Memory System Performance", 14th ACM Symposium on Operating System Principles, December 1993

[DEOS 01] – P. Binns. “A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach”, The 20th Digital Avionics Systems Conference (DASC), 2001.

[Ruocco 06] – S. Ruocco. "Real-time programming and L4 microkernels", in Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications, Dresden, Germany, July 2006.

[Lane 00] – M. Lane. “Predicting the reliability and safety of commercial software inadvanced avionic systems”. The 19th Digital Avionics Systems Conferences DASC, 2000.

[uKernel 07] – G. Heiser; K. Elphinstone; I. Kuz; G. Klein, S. Petters. "Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level", Operating Systems Review, July 2007.

[uKernel_RT 01] – M. Bennett and N. Audsley. “Developing real-time micro kernel design process”, 22nd IEEE.Real-Time Systems Symposium, London, UK, December 2001. IEEE Computer Society Press.

[L4_uKernel 03] -- M. Bennett and N. Audsley. “Partitioning Support for the L4 Microkernel”, Technical Report YCS-2003-366, Dept. of. Computer Science, University of York, 2003.

[IMA_L4 02] – M. Bennett and N. Audsley. “Developing an IMA Kernel Based on L4 for Avionic Systems”. Technical Report, Dependable Computer System Center, Dept. of Computer Science, UK, 2002.

[Massa 02] – A. Massa. “Embedded Software Development with eCos”. Prentice-Hall, 2002. ISBN 0130354732.

[VxWorks 03] – Wind River, Alameda, CA, USA. “VxWorks Programmers Guide-5.5”, 2 edition, March 2003.

[VMs 05] – J. Smith and R. Nair. “Virtual Machines: versatile platforms for systems and processes”. Morgan Kaufmann, June 2005. ISBN 1558609105.

[Digital 96] – Digital. Digital Unix - Guide to Realtime Programming. Digital Equipment Corporation, March 1996.

[POSIX 00] – K. Obenland. The use of POSIX in real-time systems, assessing its effectiveness and performance. The MITRE Corporation, September 2000.

[TSIM 08] – Gaisler Research, Goteborg, Sweden. TSIM2 Simulator Users Manual (Version 2.0.9), April 2008.

[LeonVM 06] P. Marques, J. Feiteirinha, L. Pureza, N. Lindman, and M. Pecchioli. “LeonVM: Using dynamic translation for developing high-speed space processor emulators”, 9th Int. Workshop on Simulation for European Space Programmes (SESP'2006), Noordwijk, The Netherlands, November 2006. ESA/ESTEC.

[Sched 08] – M. Coutinho, J. Rufino, and C. Almeida. Response time analysis of asynchronous periodic and sporadic tasks scheduled by a _xed-priority preemptive algorithm. In Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS2008), Prague, Czech Republic, July 2008. IEEE.

[LynxOS] – http://www.lynuxworks.com/rtos/rtos-178-do-178b.php

[CsLEOS] – http://platformsolutions.na.baesystems.com:8080/CsLeos/CsLEOS.jsp

[GHS-Integrity178] – http://www.ghs.com/products/safety_critical/integrity-do-178b.html

[VxWorks653] – http://www.windriver.com/products/run-time_technologies/Real-Time_Operating_Systems/VxWorks_653/

[PikeOS] – http://www.sysgo.com/products/pikeos-rtos-technology/

Acronyms

Abreviation	Meaning
AEEC	Airlines Electronic Engineering Committee
AIMS	Airplane Information Management Systems
AMOBA	ARINC653 Simulator for Modular Space Based Applications
ANSI	American National Standards Institute
APEX	APlication EXecutive
API	Application Programming Interface
CCS	Common Core System
COTS	Commercial Off-The-Shelf
CSRG	Common Standards Revision Group
DEOS	Digital Engine Operating System
DIMA	Distributed Integrated Modular Avionics
EASy	Enhanced Avionics System
ESA	European Space Agency
FAA	Federal Aviation Administration
FOSS	Free and Open Source Software
GPL	GNU General Public License
IEEE	Institute of Electrical and Electronics Engineers
IMA	Integrated Modular Avionics
I/O	Input/Output
ISO	International Organization for Standardization
IT	Information Technologies
MDPU	Modular Data Processing Unit
MTF	Major Time Frame
NASA	National Aeronautics and Space Administration
OS	Operating System
POSIX	Portable Operating System Interface
PASC	Portable Application Standards Committee
RAM	Random Access Memory
RTCA	Radio Technical Commission for Aeronautics
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real Time Operating System
XML	eXtensible Markup Language