xky hypervisor

Code:	xky-wp-1
Internal code:	GMV 21331/20 V1/20
Version:	1.0
Date:	15/04/2020
Author:	Tobias Schoofs

GMV
Av. D. João II, 43
Torre Fernão de Magalhães, 7º
1998-025 Lisboa
Tel. +351 21 382 93 66, Fax +351 21 386 64 93
www.gmv.com.pt

DOCUMENT STATUS SHEET

Version	Date	Pages	Changes
1.0	15/04/2020	20	Initial version of the document.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. PURPOSE

This whitepaper presents the **XKY** Partitioning Real-Time Operating System and discusses a use case in Distributed Integrated Modular Avionics (DIMA).

1.2. SCOPE

This document is released in the context of the **XKY** product development.

1.3. REFERENCES

Table 1-1 Reference Documents

Ref.	Title	Code	Version	Date
[RD.1]	Miguel Tavares de Barros: "A Distributed Integrated Modular Avionics Platform for Multi-Mission Vehicles", Lisbon, 2019.	Barros	-	July, 2019
[RD.2]	ARINC 653 Part 1	A653P1	S4	August, 2015
[RD.3]	ARINC 653 Part 2	A653P2	S1	December, 2008

1.5. ACRONYMS

Acronyms used in this document and needing a definition are included in the following table:

Table 1-2 Acronyms

Acronym	Definition
AFDX	Avionics Full-Duplex Switched Ethernet
AHRS	Attitude and Heading Reference System
BSP	Board Support Package
CDS	Cockpit Display System
CPM	Core Processing Module
CPU	Central Processing Unit
DIMA	Distributed Integrated Modular Avionics
DMA	Direct Memory Access
GPS	Global Positioning System
GPU	Graphics Processing Unit
HMI	Human-Machine Interface
IMA	Integrated Modular Avionics
IDE	Integrated Development Environment
IO	Input/Output
IOP	IO Partition
IRQ	Interrupt Request
OS	Operating System
PCA	Performance Calculation Application
PCIe	Peripheral Component Interconnect express
PMK	Partition Management Kernel
POS	Partition Operating System
PSS	Payload Supervisor System
RDC	Remote Data Concentrator
RTOS	Real-Time Operating System
SDR	Software-Defined Radio
UAV	Unmanned Aerial Vehicle

2. THE XKY HYPERVISOR

2.1. OVERVIEW

XKY is a real-time operating system built from the ground up to support systems following the paradigm of Robust Partitioning used most prominently today in Integrated Modular Avionics (IMA). The design is based on a hypervisor approach permitting several applications, and even operating systems (OS), to safely coexist on the same hardware. The **XKY** Partition Management Kernel (PMK) runs directly on bare hardware to control and manage the guest systems.

The architecture consists of two levels: the lower level is composed of a software hypervisor that robustly segregates computing resources including access to processors, memory, Direct Memory Access controllers (DMA), timers and other IO devices; the second level, the application level, is composed by applications with their own virtualised execution environments running in isolated containers called partitions.

Through its hypervisor architecture, **XKY** enables different operating systems to execute within different partitions guaranteeing uninterrupted access to the resources configured for them at design time. Any OS can be used as guest system, including general-purpose OS like Linux. However, **XKY** is optimised for supporting real-time systems and the typical guest is therefore an RTOS.

Guest operating systems need to be adapted to interface with **XKY**’s virtualization layer resulting in a software module equivalent to a new board support package for the virtualized OS that targets **XKY**.

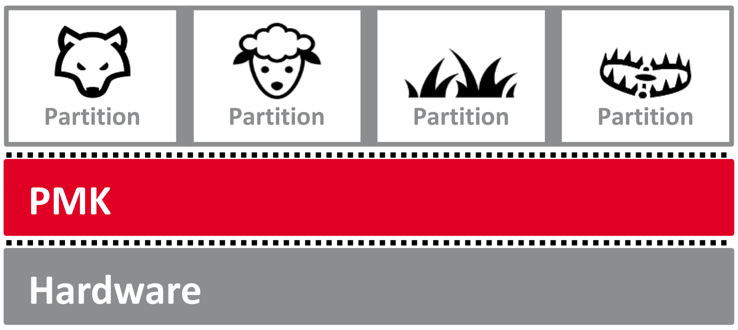


Figure 2-1: XKY **Architecture**

The PMK runs in privileged mode, while partitioned applications and guest operating systems run in user mode making it impossible for them to directly interact with critical features of the hardware. Any violation like calling a privileged instruction or accessing a resource that was not explicitly granted to this guest will cause an exception that is then handled by the kernel.

An important resource granted to partitions is processor time. Processor time is assigned by means of a static cyclic scheduling that assigns execution windows to partitions.

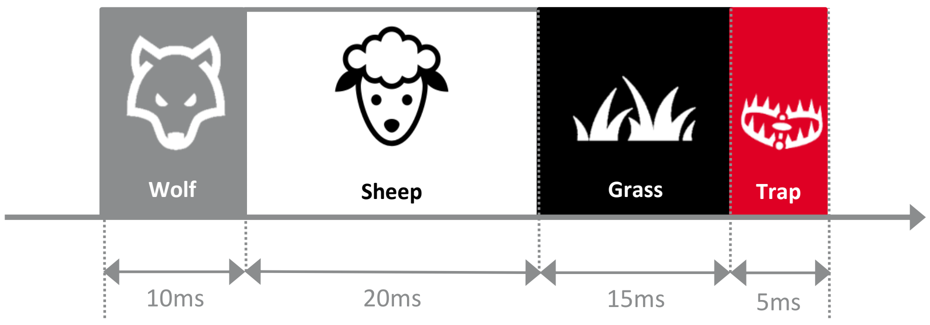


Figure 2-2: Time Partitioning

When the current execution window expires, the PMK stops the execution of the current partition and context-switches (“jumps”) to the partition assigned to the next execution window. It is, this way, impossible for a partition to “steal” time from another partition.

For real-time applications with periodic threads, the partition schedule must be aligned to the periods and the time requirements of the applications. To support an application that runs at, say, 10Hz the partition schedule must return to this partition every 100ms; if the application has a worst case execution time of 10ms, at least 10ms execution time must be assigned to this partition. The time may be allocated by one or several execution windows, e.g. one window with a duration of 10ms or two windows with 5ms each.

For safety-critical applications, typically seen in aeronautics, robust partitioning is strict; even side channels through which a partition may influence the execution time of another partition are closed. An example is caches. Every execution window starts with a clean cache (all entries are invalidated), so that the operating environment does not add to jitter. For less critical applications, those features can be relaxed optimising average performance but increasing jitter.

XKY supports multi-core. The execution windows of the scheduler refer explicitly to the processor core on which the partition shall run. It is, this way, possible to run partitions in parallel. A use case may be external IO; a dedicated partition handling IO could run in parallel to application partitions making data sent by or to partitions available almost instantly.

Multi-core, however, may lead to competition for internal resources like the memory bus. For critical systems, a careful trade-off is necessary to decide on the design of the system at hand. Often system integrators decide to use only one CPU even though more processing units would be available on the target board. **XKY** provides the mechanism to use multiple cores, but it does not prescribe the policy of how using these cores. The policy remains to be decided by the system integrator.

Compared to other partitioning OS, **XKY** has relevant advantages that make it unique even in the already small group of this kind of systems:

- Certifiability
  We are currently working on the documentation set for DO-178C certification up to DAL A. The documentation is expected to be available still in 2020.
- Hypervisor Approach
  The consequent application of the hypervisor approach makes **XKY** easy to be ported to new boards and to port other execution environments and OS to run on **XKY**. Our internal porting projects for new boards have schedules of two to four weeks (depending on the complexity of the target). Porting a classical RTOS (like RTEMS) to **XKY** usually takes about one to three months. Of course there are systems that are easily ported and others that are much harder (e.g. Linux).
- Small Footprint and Simplicity
  The **XKY** PMK has a very small code base of 5-6K lines of code. This leads to a relatively small document set for certification and, in consequence, helps to keep certification time and cost low. It also opens the way for a complete formal verification for future security certification at highest assurance levels (e.g. EAL7). Beyond certification considerations, simplicity is also a good means to reduce the risk of bugs in the code and makes **XKY** attractive even for non- or less critical projects.
- Performance
  XKY is fast. This is true for partition context switching, partition-internal process context switching and system calls. For critical systems, performance is of course not the main concern and often speed is sacrificed for stronger safety or security guarantees. However, having good average and worst case execution time is definitely an advantage in non-critical projects; but even critical systems usually benefit from good performance.
  The main statement in this paragraph is intentionally left vague. We will not discuss performance in detail here. Another publication will discuss this topic with the required scientific rigour.

2.2. FEATURES

2.2.1.AVAILABLE FEATURES

The core feature of the **XKY** RTOS is Robust Partitioning. Partitioning can be strict (for safety- or security-critical systems) or relaxed. In strict partitioning, caches are flushed and invalidated on each context switch. This makes covert channel attacks through the cache impossible (for security-critical system) and it reduces jitter (for hard real-time and safety-critical systems). For less critical systems, partitioning may be relaxed. In that case, caches are only flushed and invalidated if explicitly requested in the configuration.

XKY supports multi-core. Partitions can be scheduled on any number of processing cores by indicating the CPU on which the current partition(s) shall be executed.

XKY provides fast communication between partitioned applications by means of queueing and sampling ports and shared memory. The transport between ports is completely managed within the kernel and may use either memory copy, DMA or PCIe.

XKY also supports external communication through PCIe (implemented in the kernel) and through Ethernet or AFDX (implemented in a partition that is scheduled together with other partitions). These interfaces work out-of-the-box. Other IO drivers can be easily integrated at partition level.

XKY provides an ARINC 653 APEX for Part 1 (supplement 3 and partly supplement 4, [RD.2]) and selected services of Part 2 [RD.3]. The Multiple Module Schedule, Sampling Port Extensions and Shared Memory services (all Part 2) are integrated features of the **XKY** APEX. Add-ons are the Logbook System, Service Access Points and the ARINC 653 Filesystem (upcoming).

XKY currently targets PowerPC (32 and 64bit) and ARM (Cortex-A8, Cortex-A53, Cortex-M4 among others). There is also a hardware-independent "BSP" for Unix/Linux that can be used as a simulation and prototyping tool.

XKY is certifiable. We are currently working on the documentation set for DO-178C certification up to DAL A. The documentation set is expected to be available in the last quarter of 2020.

2.2.2.UPCOMING FEATURES

We are currently working on new sets of features like:

- GPU Support for fast numerical computing and graphics;
- Symmetric Multi-Processing (ARINC 653 P1 Supplement 4);
- A POSIX Personality including subsets 1, 1b and 1c;
- Support for Linux as guest OS;
- An Integrated Development Environment (IDE) with extensions for IMA Application Developers and System Integrators.

We are constantly working on new target boards, hardware architectures and device drivers.

2.3. ARCHITECTURE

Figure 2-3 below depicts the overall design of the PMK, showing its main components:

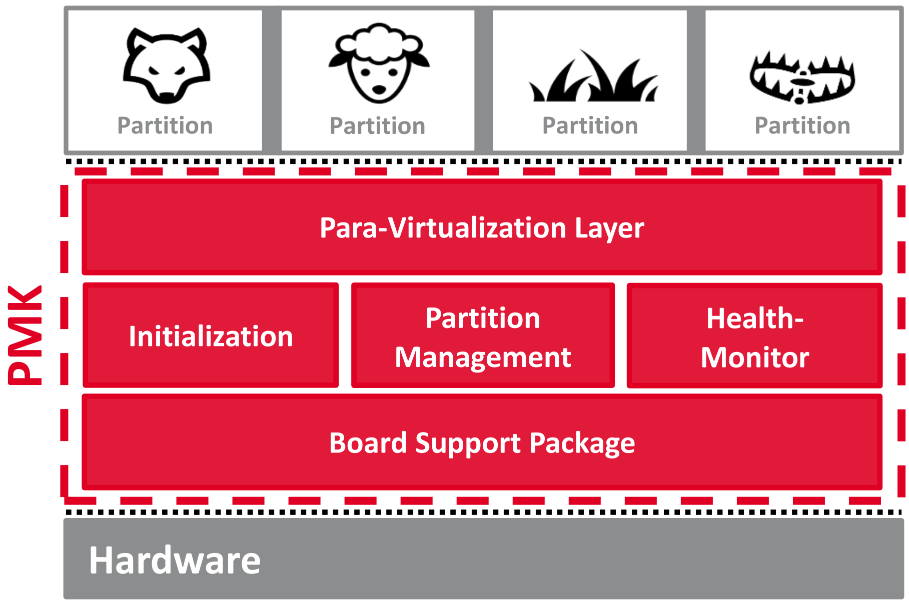


Figure 2-3: XKY PMK Design

The main components in the PMK are (bottom-up, left-to-right):

- The Board Support Package (BSP)
- A set of initialisation routines and hooks
- The Partition Management Component
- The Health-Monitor
- The Para-Virtualisation Layer.

The PMK prescribes a clear interface towards the hardware. This interface is called the Board Support Package (BSP). The BSP contains low-level initialisation routines, access to specific processor features (enabling/disabling cache, enabling/disabling interrupts, etc.) and services to interact with devices. The hardware-independent part of the PMK, the so called "Core", remains the same on all boards and processor architectures; what varies is the implementation of Core features on the specific hardware in the BSP. But also the structure of the BSP is very similar for different boards. It is basically defined by the features requested by the Core. This way, porting the PMK to a new board is a routine job for an experienced programmer.

The basic set-up of the hardware after system start is one of the tasks of the BSP. When the low-level initialisation routine concludes, the BSP calls the PMK initialisation which validates the configuration, initialises the partition schedule and the health-monitor and finally, starts the scheduler. From now on, the PMK will only be active in interrupt handlers. There are no kernel threads like in most general purpose OS and most RTOS.

Partition Management is mainly concerned with partition scheduling. It is worth mentioning that the PMK does not use a regular time tick. Instead, it just sets a timer for the next event. This approach, sometimes called "tick-less system", reduces interrupts drastically improving the overall performance of partitioned applications and, potentially, reducing energy consumption.

When the timer fires, the system enters the interrupt context for the timer interrupt. The handler performs some housekeeping (like updating system time and setting the current partition) and performs a "return-from-interrupt", i.e. it returns to partition execution.

An important component is the Health-Monitor (HM). The HM shields the system against all kinds of faults. Faults can be caused by software errors in application code, by hardware faults or by radiation and other unexpected events from outside the computer (e.g. fault of another computer in a distributed system).

How to react to errors depending on where (PMK, partition) and when (initialisation, normal execution, error handling) they occur is defined in the HM tables. The system integrator specifies whether errors are handled by application code or by the PMK and how they should be addressed (by invoking a user-defined error handler, by restarting or stopping a single partition or by restarting the computer).

It is essential that calling a privileged instruction in user mode (i.e. in a partition) will trigger an error and finally enter the HM. An application trying to break out of partitioning is, hence, considered a software fault that will be handled, for instance, by stopping the faulty partition.

The overall design of the **XKY** PMK follows, as mentioned, a hypervisor approach. The PMK provides a number of services to the guest OS, which at some point boil down to specific hardware features. To improve performance the **XKY** hypervisor uses *para-virtualisation* techniques. That means it provides a software interface to the guest system that is composed of system calls (syscalls for short) and virtual interrupt requests (IRQs), as depicted in Figure 2-4Figure 2-4

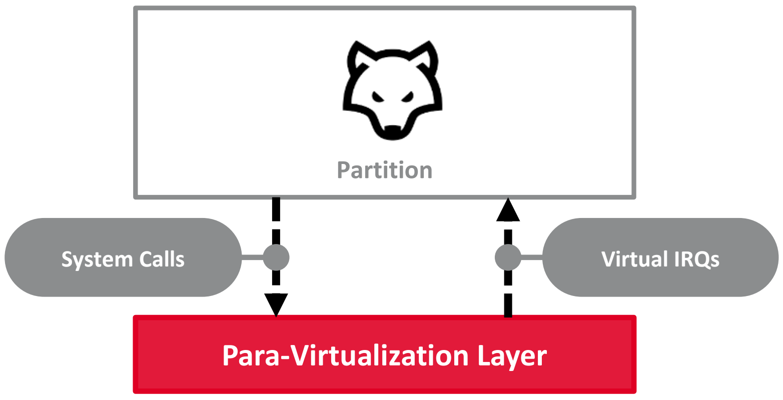


Figure 2-4: Virtualisation Means

Syscalls are software interrupts that, in essence, invoke services implemented in the PMK which, in their turn, interact directly with the hardware. This kind of virtualisation is inspired by the design of traditional operating systems. The alternative approach, to trap all privileged instructions issued by guest systems, analysing them and deducing the intended behaviour, may be more appealing in terms of virtualisation design, but is slower by orders of magnitude. It works fine on modern desktops and servers, but is often too slow for embedded real-time systems. It is also significantly more complex leading to increased certification cost.

The guest OS, on the other hand, must be prepared (manually or automatically) to run on the **XKY** PMK. That is: it must use the software interface provided by the PMK. This, however, is easier than it might sound at first. The PMK software interface mimics assembly language instructions providing calls that correspond closely to instructions typically found in common instruction sets, such as installing an interrupt handler, setting a timer or writing to/reading from an IO device. The virtualisation layer even provides architecture-specific syscalls for instructions that do not correspond directly to common PMK calls (e.g. accessing privileged special purpose registers). This class of syscalls is implemented directly in the BSP-part of the PMK and is only available for the specific target board.

In the other direction, the PMK provides virtualised IRQs. From the perspective of the guest, they look like ordinary IRQs. The guest system can install handlers for specific IRQs, disable and enable them and basically do anything one does with interrupts. Most of the virtual IRQs are very common. There are for instance timer interrupts, interrupts from devices and error exceptions. Additionally, there are very special interrupts not related to hardware events, namely interrupts indicating events in the partition scheduling, e.g. "partition period start" or "new major time frame".

The **XKY** ecosystem includes building blocks to define guest *personalities*. A personality is a specific design of a guest system including a partition operating system (POS) and a set of libraries defining language bindings, a threading model, inter-partition communication and so on. The simplest personality distributed with **XKY** is the *Bare* POS. The Bare is single-threaded execution environment that provides a C API wrapping syscalls and a set of services for interrupt handling and time management (e.g. "sleep").

The standard personality for **XKY** is, of course, the ARINC 653 APEX which provides a rich set of partition management services, process management services, error handling as well as inter-process and inter-partition communication services.

But there are also other personalities; in fact new personalities can be easily built, for instance a POSIX POS, an RTEMS POS, specific language bindings (e.g. C++, Ada, Rust or even scripting languages like Python or Lua) and finally a POS with a virtualised general purpose OS like Linux. Figure 2-5 shows schematically a possible "zoo" of personalities:

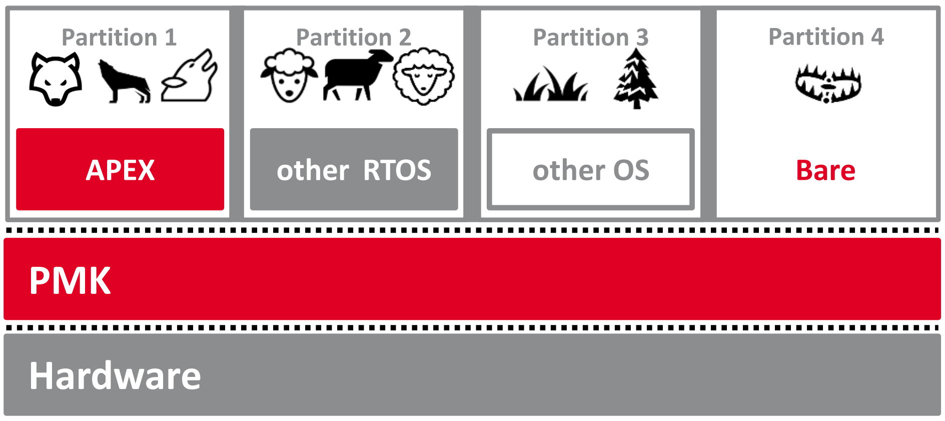


Figure 2-5: Guests

Of particular importance for inter-operability of complex systems is inter-partition and inter-module communication. **XKY** implements several communication means. The most important is channels whose endpoints appear in the ARINC 653 personality as queuing and sampling ports.

For local channels, i.e. for channels that connect partitions on the same computer, the PMK implements the transport by means of memory copy, directly or, for larger amounts of memory (and if available on the target board), DMA.

Data transport is, of course, subject to time partitioning. If the data cannot be copied within the execution window, the procedure is interrupted and resumed when the partition regains the processor. For application developers this means that the duration of the transport must be considered as part of the timing requirements.

Channels can also connect partitioned applications with the outside world. One way to do that is PCIe. The PCIe driver is implemented in the PMK. Data transfer through PCIe is therefore direct, i.e. data is transferred as soon as the device is ready. Note, however, that for transferring data to the device, the same qualification as for any memory copy applies: if the transfer does not fit into the execution window, it is interrupted on partition context switch and resumed when the partition regains the processor.

It is also possible to connect channels through common networks like Ethernet or AFDX (ARINC 664, Part 7) or even through buses like ARINC 429, MIL-STD-1553, etc. Those alternatives are implemented in a special kind of partition called IO Partition (IOP). Within the **XKY** configuration, the corresponding channels are specified as ordinary channels from a port in a sending application partition to a port in the IO partition or from a port in the IOP to a port in a receiving application partition. This approach is not direct like the PCIe binding. Data is sent through the bus only when the IO partition is scheduled, that means there may be a delay between the point in time when the message is sent by the application and the point in time when it is actually put on the wire by the IO Partition. Figure 2-6 depicts the principle of the IOP:

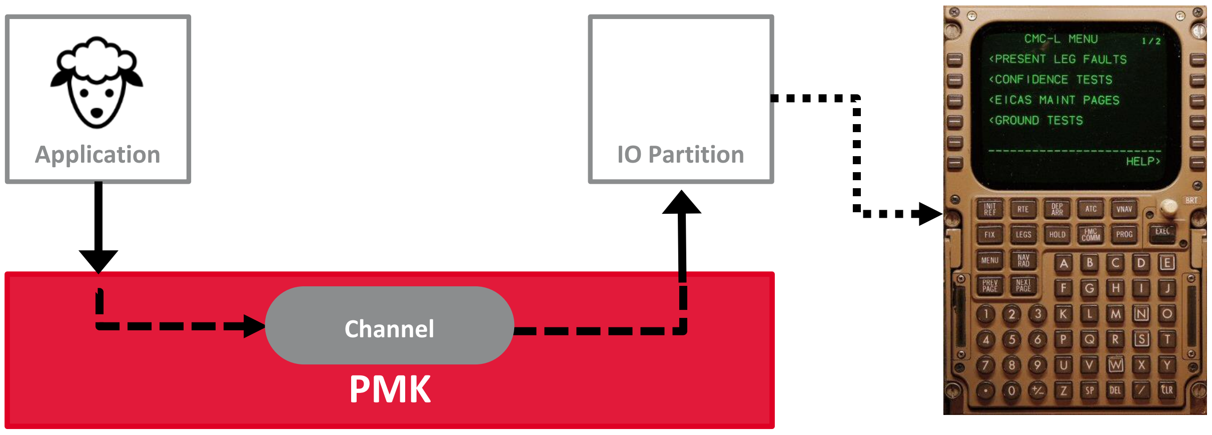


Figure 2-6: IO Partition

XKY is distributed with an IOP that provides UDP over Ethernet out-of-the-box. It is sufficient to configure the addressing for the ports in the IOP configuration; the actual sending and receiving is done internally by the IOP. There is also an add-on for AFDX that works similar but uses AFDX instead of Ethernet. It is further possible to provide a custom IO Partition that would use other networks or buses. At GMV we have a considerable set of drivers (for aeronautics, space, automotive and other use cases) that can easily be integrated with **XKY**.

3. USE CASE: DIMA

3.1. THE DIMA PROJECTS

The acronym "DIMA" stands for *Distributed Integrated Modular Avionics*. Distributed IMA, sometimes called 2nd Generation IMA, is an emerging architectural pattern for avionics. It refers, here, to a prototype for an advanced IMA platform developed by EMBRAER in cooperation with GMV during a number of projects from 2014 until today, and ongoing. The platform provides:

- Flexible use of computing and IO resources;
- System-level interoperability;
- Reuse of application components;
- Reconfiguration ("plug & play").

DIMA separates computing resources into two categories: Core Processing Modules (CPM) and Remote Data Concentrators (RDC). CPMs are computers that consist of a CPU, memory and an IO interface to connect the module to a network (typically AFDX). CPMs provide mainly computing power.

RDCs are IO-oriented computers that have a network interface (typically AFDX) and additional interfaces for field buses. RDCs are usually installed outside the avionics bay close to the sensors and actuators to which they connect.

CPMs and RDCs together establish a hardware platform to run complex, distributed software applications. They can be seen as a computing "cloud" on which applications may be hosted. Partitions in the CPMs run application components that perform the computing part of the job, often requiring significant processing power and memory. Application components on the RDCs mainly control sensors and actuators and perform data pre-processing.

An Attitude and Heading Reference System (AHRS), for instance, needs different sensors, e.g. gyroscopes, accelerometers and magnetometers. The sensors are connected to an RDC (or, in fact, for redundancy reasons to several RDCs). The RDC would pre-process the data and send the result to the AHRS core system hosted on a partition in a CPM. That component would use the incoming data to compute attitude, pitch, yaw, roll and so on in the given frame of reference applying Kalman filters or other techniques. It would further process and transfer the data to present them on an HMI in the cockpit display system (CDS).

The concrete mapping of software to hardware is controlled by the configuration. The configuration defines where application components are hosted, what kind of sensors are available and to which RDCs they are connected; but the configuration has also to ensure that the system interoperates correctly as a real-time system. In other words, the partition scheduling of all CPMs is part of the system configuration; likewise the HM tables of all computers and the channels that connect partitioned application components and the network routes that provide external links between modules and RDCs. For a realistic avionics system the configuration consists of thousands of items that must all fit together.

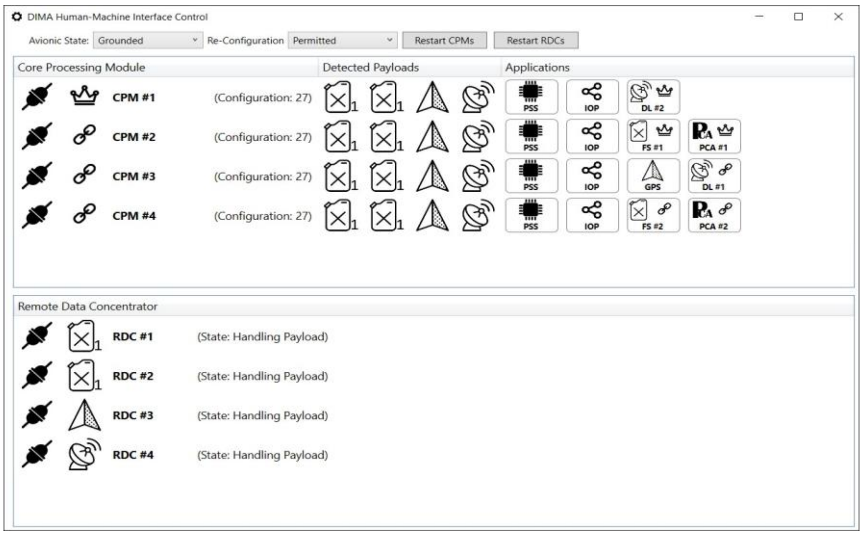
For safety reasons the configuration is static and defined at design time. This, however, leads to a system that cannot be reconfigured. But there are scenarios where reconfiguration is highly desirable, for instance:

- Different missions of the same system may require different payloads;
- Different phases of the same mission may require different workloads from different subsystems;
- Varying spare parts may require different drivers and imply changes in timing requirements;
- In case of a device failure, the system may need to downgrade to still be operable (e.g. for improving aircraft dispatchability).

The DIMA platform solves this dilemma by means of multi-static reconfiguration. That means that there is a set of static configurations, from which the most appropriate one is selected at system start. (For safety reasons, inflight reconfiguration is not allowed.)

In the selection process, the components of the system (CPMs and RDCs) share their view of the system state and try to reach a common view using a consensus algorithm. When consensus is reached and if there is a valid configuration for that system state, all components will select the same setup and apply the relevant configuration part to themselves. Applying the configuration potentially implies rebooting the whole computer or stopping some partitions and starting others and then switching to another partition schedule.

Finally, the network configuration must be changed. To make sure that this happens only once, a leader is selected that will apply the network change. The leader selection process is also based on a consensus algorithm and is repeated, whenever the current leader fails for some reason. Figure 3-1 shows a screenshot from the Configuration View of the DIMA demonstrator (see Section 3.2):



Avionic State: Grounded
Re-Configuration: Permitted

CPM #1 (Configuration: 27)
CPM #2 (Configuration: 27)
CPM #3 (Configuration: 27)
CPM #4 (Configuration: 27)

RDC #1 (State: Handling Payload)
RDC #2 (State: Handling Payload)
RDC #3 (State: Handling Payload)
RDC #4 (State: Handling Payload)

Figure 3-1: Configuration View in the DIMA Demonstrator

In the upper frame four CPMs are shown (#1 - #4). In this concrete situation, all CPMs are available and consensus has been reached. The leader is CPM #1. The selected configuration is the one with the identifier 27.

Each of the four CPMs hosts three to four application components: the PSS (which we will discuss in a minute), an IO Partition and at least one payload-related component; CPM #1, for instance, hosts a part of a software-defined radio application (SDR); CPM #2 hosts a fuel tank control application and a performance calculation program (PCA) which computes the flight envelope. CPM #3 hosts the GPS and another component of the SDR. Finally, CPM #4 hosts the other fuel tank controller and a redundant PCA replica. (Notice that for applications that consists of several components or redundant replicas there also are leaders and followers. For brevity, we will not discuss such details here.)

In the lower frame four RDCs are shown, each connected to a payload. There are two RDCs (#1 and #2) that are connected to fuel tank sensors; #3 is connected to a GPS receiver and #4 is connected to the SDR.

The Configuration View shows that all components are in a consistent state. For each payload handled by an RDC there is at least one application component hosted on one of the CPMs that handles this payload. Furthermore, all CPMs have detected the same system state, i.e. the same set of payloads, as the column “Detected Payloads” in the upper frame indicates.

The reconfiguration framework is also a good example for system-level interoperability and component reuse. The reconfiguration logic including the consensus algorithm is implemented in a component called Payload Supervisor (PSS). This component is hosted on all CPMs; the supervisors communicate with each other to reach a common system view, select the leader and perform the reconfiguration activities. The PSS, this way, is a component that makes the set of computers work together as a system. We can observe that the *system* performs reconfiguration as if it was one single application.

At the same time the PSS is a reusable component. It fulfils a clearly defined task in a system and can be used in any application that needs this capability. There are many more candidates for such reusable components; in fact, all components shown in Figure 3-1 may, to some extent, be reusable.

3.2. THE DIMA DEMONSTRATOR

In the course of the DIMA projects, GMV and EMBRAER implemented a number of demonstrators to provide a proof-of-concept for the DIMA platform. The main demonstrators are a lab demonstrator and a flight demonstrator. The flight variant was used in a model aircraft (see Figure 3-2).

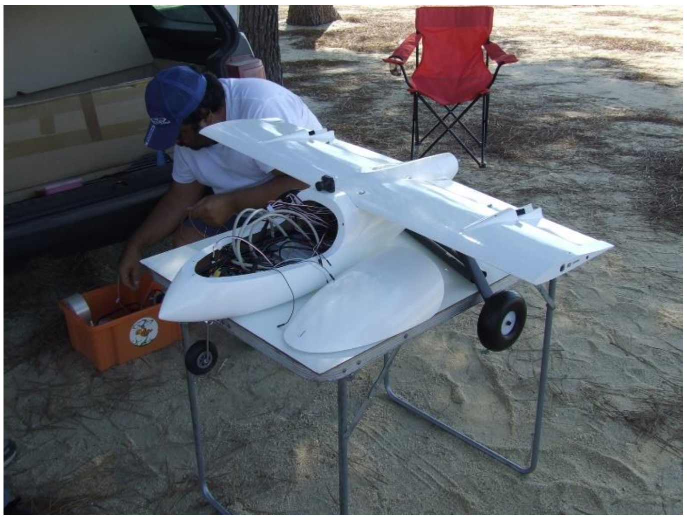


Figure 3-2: Avionics Bay of the Flight Demonstrator

We will focus here on the lab demonstrator, because the flight demonstrator consists of only two CPMs and two RDCs. The lab demonstrator, on the other hand, consists of

- Four CPMs running on Freescale PowerPC P1010 boards;
- Up to seven RDCs running on BeagleBone Black boards (ARM Cortex-A8);
- Up to seven payload components, some with real sensors and actuators, some with simulations running on Arduino boards, namely
  - Fuel Tank sensors (simulated)
  - A weapon system, “Bombs” (simulated)
  - Gyroscope (real sensor)
  - Global Positioning System (real receiver)
  - Camera System (real, with remotely actuated gimbal)
  - Software Defined Radio (real, including FPGA board).

The CPMs run **XKY** with three to four partitions each: the PSS (A653 APEX), one or two application components (A653 APEX) and an IOP. Note that there were sufficient memory and processing resources left to host more applications on the CPMs. But for the purpose of the demonstrator that was not necessary.

We also used alternative set-ups where the CPMs were hosted not on PowerPC boards, but on Beagle Bones. The reason for this was just pragmatic: for demonstration outside our facilities we didn’t want to move the relatively expensive Freescale boards around. Instead we used the much cheaper BeagleBones for such occasions. (In the course of the DIMA projects, we managed to destroy, in one way or the other, several boards. Therefore, having **XKY** BSPs for cheap toy boards available proved to be valuable).

The RDCs run RTEMS. The reason, again, was purely pragmatic. Many drivers are already available for RTEMS and so we did not need to develop new drivers for those cases.

The pieces are connected by a conventional Ethernet network.

Figure 3-3 shows one possible configuration of the Lab demonstrator:

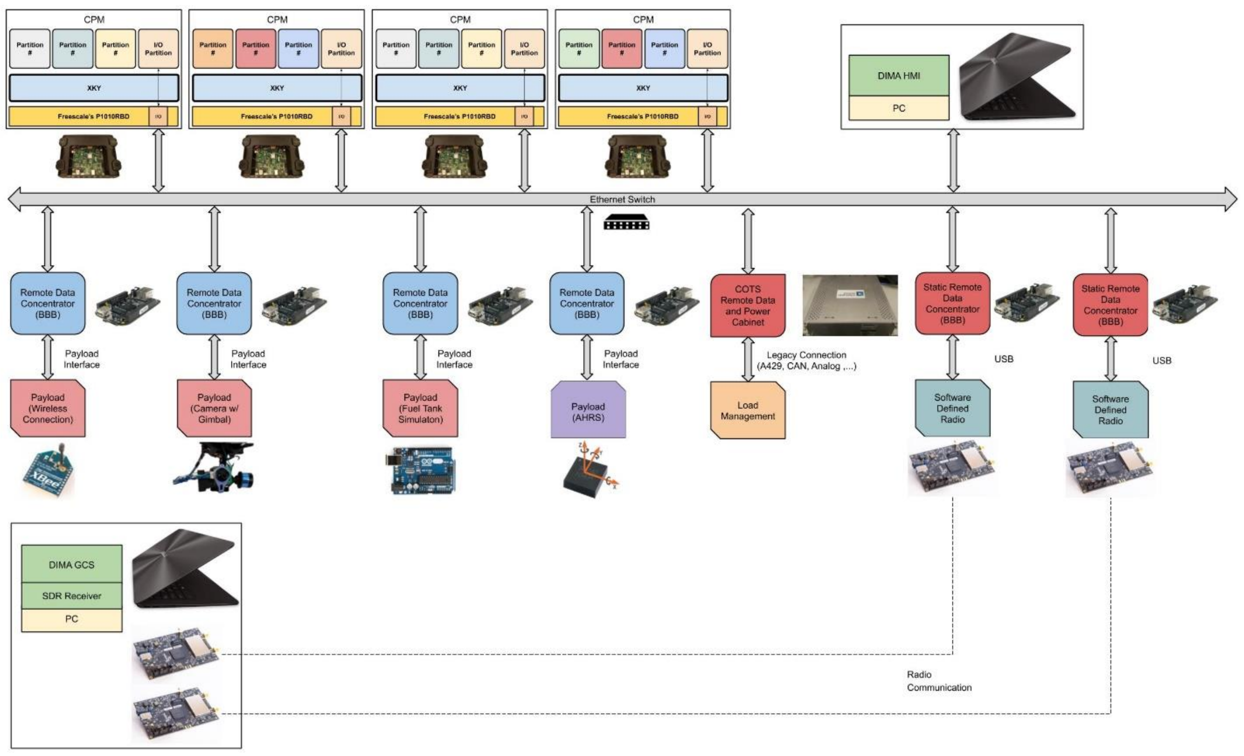

CPM
Partition #
I/O Partition
XKY
Freescale's P1010RBD
I/O
DIMA HMI
PC
Ethernet Switch
Remote Data Concentrator (BBB)
Payload Interface
Payload (Wireless Connection)
Payload (Camera w/ Gimbal)
Payload (Fuel Tank Simulator)
Payload (AHRS)
COTS Remote Data and Power Cabinet
Legacy Connection (A429, CAN, Analog ,...)
Load Management
Static Remote Data Concentrator (BBB)
USB
Software Defined Radio
DIMA GCS
SDR Receiver
PC
Radio Communication

Figure 3-3: Possible Configuration of the Lab Demonstrator

There are some constraints that limit reconfiguration, for instance:

- The fuel tank payloads must be connected to an RDC close to the tanks (which in the lab demonstrator were, of course, just assumed to exist);
- The gyroscope must be connected to an RDC close to the centre of gravity of the aircraft;
- Some payloads, for instance the camera (which is not a realistic payload for a civil aircraft, but more likely for a UAV), make sense only at the nose of the aircraft.

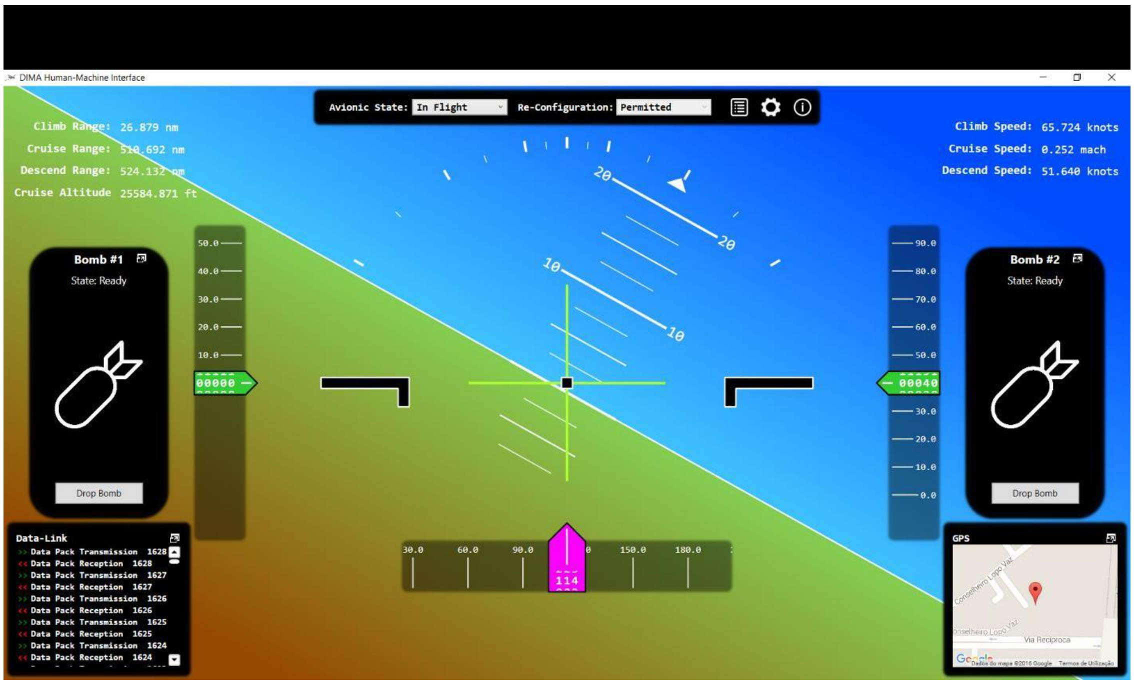
Despite such constraints, there are 27 possible configurations for the lab demonstrator. The configurations take different combinations of payloads into account, but can also deal with missing RDCs or failure of single CPMs. If an RDC is missing the payloads supposed to be connected to that RDC would be removed from the CPMs. On the other hand, if a CPM is missing or fails during the configuration phase, essential applications that were configured to run on that CPM would be re-hosted on other CPMs. ("Re-hosting" here means, of course, to activate another configuration that foresees this new mapping of applications to CPMs.)

When a payload is changed, the CPMs (i.e. the PSS running in the CPMs) would stop the partition that hosted the controller for the removed payload, start the partition containing the controller for the new payload and switch to another schedule. For this to happen the PSS has supervisor privileges; it is able to stop or reboot the module, to start or stop partitions and to change the module schedule (see Part 2 of ARINC 653, "Multiple Module Schedules").

The **XKY** OS allows defining advanced privileges at a fine grain. This is part of the normal partition configuration. Usually a partition is not allowed to change or even see the state and settings of other partitions. A partition must be explicitly configured to grant it the power to control its peers.

A concrete example of changing a payload is the gyroscope. There is indeed a valid configuration without a gyroscope; needless to say that this is not a realistic scenario for a real airplane, but for the lab demonstrator we decided to add this possibility. If no gyroscope is connected, the system is unable to compute attitude and heading. It may still perform other tasks, though, like controlling the camera payload.

When the gyroscope is plugged in to an RDC, the system reconfigures and starts the AHRS. The system, now, is aware of attitude and heading and is able to simulate a cruising airplane. This is reflected in the DIMA main HMI:


DIMA Human-Machine Interface

Avionic State: In Flight
Re-Configuration: Permitted

Climb Range: 26.879 nm
Cruise Range: 510.692 nm
Descend Range: 524.132 nm
Cruise Altitude 25584.871 ft

Climb Speed: 65.724 knots
Cruise Speed: 0.252 mach
Descend Speed: 51.640 knots

Bomb #1
State: Ready

Bomb #2
State: Ready

```
Data-Link
>> Data Pack Transmission  1628
<< Data Pack Reception  1628
>> Data Pack Transmission  1627
<< Data Pack Reception  1627
>> Data Pack Transmission  1626
<< Data Pack Reception  1626
>> Data Pack Transmission  1625
<< Data Pack Reception  1625
>> Data Pack Transmission  1624
<< Data Pack Reception  1624
```

Figure 3-4: Main HMI Window – DIMA in Flight

The simulated airplane in Figure 3-4 is leaning to the left. In the simulator this is not achieved by steering, but by manipulating the gyroscope. Without this sensor connected, the simulator would always show the horizon parallel to the top- and bottom border of the window. In the configuration that is shown here, the fuel tank sensors (which are part of the default configuration) were removed in favour of the "bombs", which are shown to the left and the right of the centre. As can be seen in the bottom-right corner, the GPS receiver is connected to the system and correctly reports the position of the simulated aircraft.

The lab demonstrator was used for demonstrations in-house and at EMBRAER facilities. It will be the basis for more complete and more advanced simulations in the future adding essential parts of a real flight system. It is also planned to add GPU support for fast numerical computation and, with this, implement on-board components for autonomous systems like obstacle recognition and collision avoidance.

The flight demonstrator flew twice in August and September 2018. The flight was controlled remotely by a trained UAV pilot. During a short timeframe while cruising, the remote control was interrupted and the model flew autonomously for several minutes. At the end of the second flight an accident happened; the model crashed from about two metres during landing. This, however, was not related to the DIMA-driven avionics, but due to strong wind gust at that day.

During the DIMA-2 project a master thesis in Aeronautical Engineering was written [RD.1]. The thesis puts the DIMA projects into context with aeronautical research and development, describes the demonstrators in detail and draws some conclusions. The future work proposed by the thesis had strong influence on the planning of the next DIMA activities.

EMBRAER and GMV consider the DIMA projects successful and will continue the cooperation on Distributed IMA in the near future.

For GMV the DIMA projects also provided an optimal opportunity to use **XKY** in a non-trivial environment. We are proud to conclude that **XKY** mastered the challenges very well. In the course of the projects, **XKY** was deployed on different hardware and hosted very divergent applications, both intense in computation and in communication through ARINC 653 channels and through an Ethernet network. No particular difficulties were imposed by the choice of the **XKY** RTOS. On the contrary, thanks to strict compliance with standards, software development and porting of reusable components already developed internally at EMBRAER was easy and painless. An important milestone for us was the **XKY** maiden flight in August 2018. This is evidence that we are able to increase the Technology Readiness of the **XKY** RTOS very soon. The **XKY** development team is now looking forward to the next challenge.

END OF DOCUMENT