D4.1

Alpha OA & RTE prototypes

Project number:	644080
Project acronym:	SAFURE
Project title:	SAFURE: SAFety and secURity by dEsign for interconnected mixed-critical cyber-physical systems
Start date of the project:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1

Deliverable type:	Report
Deliverable reference number:	ICT-644080 / D4.1 / FINAL | 1.0
Work package	WP4
Due date:	July 2016 – M18
Actual submission date:	29 July 2016

Responsible organisation:	SYSG
Editor:	Mikalai Krasikau
Dissemination level:	PU
Revision:	1.0

Abstract:	The document is a report complementing D4.1 Alpha demonstrator. It describes the amount of work done, the demonstrated achievements, and also future plans of WP4.
Keywords:	run-time engines, multicore, scheduler, mixed-criticality, monitor, crypto, security

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080.

This work was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Editor

Mikalai Krasikau (SYSG)

Contributors

Sylvain Girbal (TRT)

Stefania Botta, Luigi Santamato (MAG)

Jaume Abella (BSC)

Rehan Ahmed (ETHZ)

André Osterhues, Cheng Lu (ESCR)

Marco Di Natale (SSSA)

Mikalai Krasikau (SYSG)

Reviewer

Dominique Ragot (TCS)

Executive Summary

The first delivery of work package four provides an Alpha version of the demonstrator and this document is its complementing. The demonstrator reflects the current state of the work package. The work done is represented in a hardware prototype and described in the current report. Also the current report provides results of evaluation of the future work within the work package.

This report will be complemented by the results of the final demonstrator at the end of the work package.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The current report provides preliminary results and achievements of the work package four. This work package uses modeling and algorithms from work packages two and three in demonstrator implementation. Results of this work package will be used in Telecom and Automotive use cases of work package six.

The report is divided by chapters and sections where every chapter describes a separate part of the work package and sections highlight different topics within that common part.

Chapter two focuses OS support and security aspects of WP4. It contains a section describing the results of porting PikeOS on the ARMv8 architecture and the status of the port on the real hardware. The following section is dedicated to port the cryptographic library CycurLIB by Escrypt on top pf PikeOS. Further, there also a section with analysis of implementation of the EDF scheduler in PikeOS. The last two sections focus on the architecture of Secure Boot and Secure Update techniques.

Chapter three considers scheduling in terms of mixed criticality in different ways. The hardware support for measurement of mixed-criticalities and benchmark characterization methodology are described in the first section. The following section gives an introduction of run-time engine prototype. And the last section is focused on extension of thermal protection in mixed criticality scheduling algorithms.

Chapter four evaluates the further development (taking into account the results of chapter three) of the run-time engine in the demonstrator prototype. More details on different types of critical levels of applications are given there.

Chapter five is dedicated to safety of AUTOSAR OS-applications. First section provides some extracts from ISO 26262 standard in terms of safety. The following section focused on types of AUTOSAR OS-Applications and software safety protection mechanisms. And the last section describes the developed firmware drivers for timing and memory protection.

Chapter six focuses on support of mixed criticality in AUTOSAR OS and also on code generation in AUTOSAR run time environment for mixed critical applications based on their models.

Chapter 2 PikeOS Support

2.1 PikeOS on the target

2.1.1 Platform Selection

During the first six months of the project we have assessed several candidate platforms for the project. In the short list were boards based on DragonBoard 810 and ARM Juno board. Both boards are development platforms based on the similar ARM 64 bit CPU architectures. However, these boards target different development purposes.

The DragonBoard 810 is designed as a development board for earlier application prototyping.

ARM Juno board is designed for development of low-level system software such as hypervisors and assessing CPU architectures for end designing devices.

The following table present a decision matrix we made for PikeOS:

	DragonBoard 810	**ARM Juno**
Fully open documentation	No	Yes
Fully open hardware for low-level development	No	Yes
Suitable for application development	Yes	Yes
Suitable for system SW development	No	Yes
Enable close to market demonstrator	Yes	No
JTAG interface	No	Yes

Table 1: Comparison of DragonBoard 810 and ARM Juno

Having this information and to avoid delay in the project the consortium agreed on the following steps:

- Port PikeOS on Juno board and develop needed hypervisor support
- Setup contact to Qualcomm to get access to low-level documentation on SnapDragon 810
- Port PikeOS on DragonBoard 810

In this deliverable we describe the status of porting PikeOS on ARM Juno board.

2.1.2 Juno Board

Within work package four SYSGO provides support for PikeOS on the ARMv8 architecture. We have structured the porting work in two phases. The first phase, early development, has been done on Fixed Virtual Platforms simulator provided by ARM. The second phase has been done on ARM Juno board. ARM Juno board has been chosen as a target for the porting considering the following arguments:

- It is the only fully open development ARMv8 board with industrial support available on the market (at the time of making the decision on the board)
- It provides required debug facilities (jtag and trace, serial port) for developing an operating system and hypervisor
- The support is provided directly by ARM Company.

The characteristics of the board are:

- Compute Subsystem
    - Dual Cluster, big.LITTLE configuration
    - Cortex-A57 MP2 cluster (r0p0)
    - Overdrive 1.1GHz operating speed
    - Caches: L1 48KB I, 32KB D, L2 2MB
    - Cortex-A53 MP4 cluster (r0p0)
    - Overdrive 850MHz operating speed
    - Caches: L1 32KB, L2 1MB
    - Quad Core MALI T624 r1p0
    - Nominal 600MHz operating speed
    - Caches: L2 128KB
    - CoreSight ETM/CTI per core
    - DVFS and power gating via SCP
    - 4 energy meters
    - DMC-400 dual channel DDR3L interface, 8GB 1600MHz DDR
    - Internal CCI-400, 128-bit, 533MHz

- Rest of SoC
    - Internal NIC-400, 64-bit, 400MHz
    - External AXI ports: using Thin-Links
    - DMAC : PL330, 128-bit
    - Static Memory Bus Interface : PL354
    - 32bit 50MHz to slow speed peripheral
    - HDCLCD dual video controllers: 1080p

- Debug
  - ARM JTAG : 20-way DIL box header
  - ARM 32/16 bit parallel trace

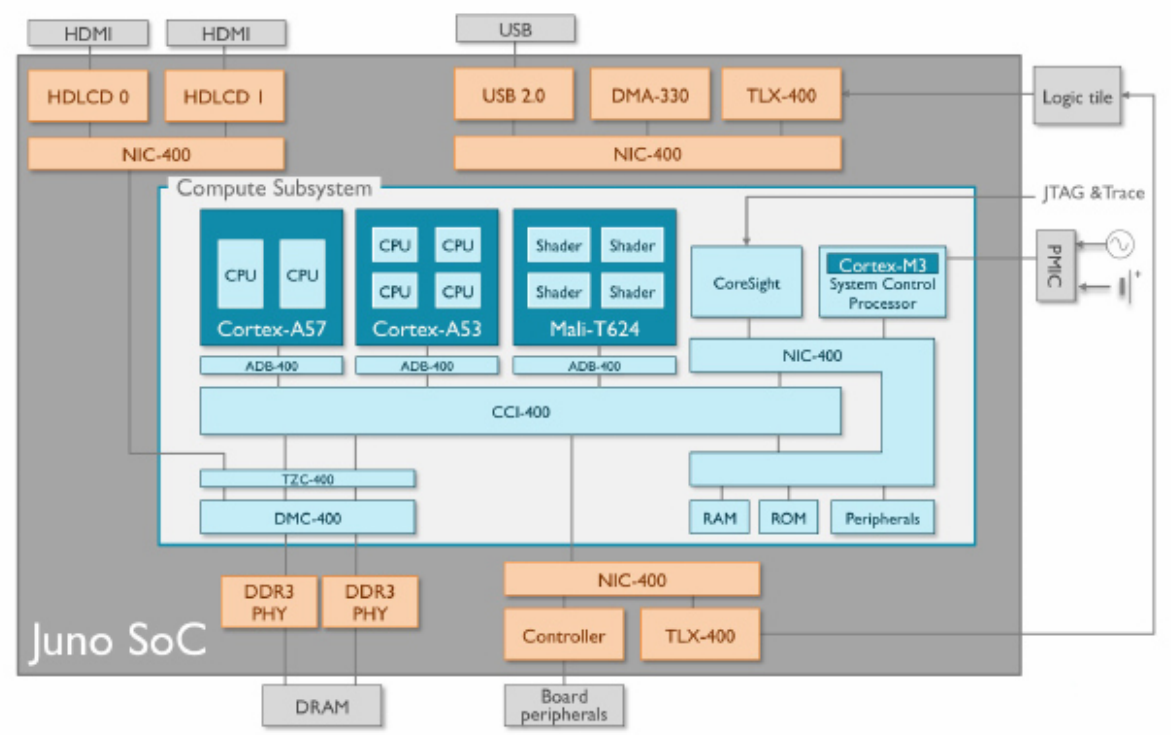

Figure 1: Juno SoC architecture

2.1.3 Alpha: Porting Results Fixed Virtual Platforms simulator

As it has been said before the first porting has been done with using ARM Fixed Virtual Platform simulator for A53-A57 cores.

The software models provide Programmer's view models of processors and devices. The functional behaviour of a model is equivalent to real hardware. PV models sacrifice absolute timing accuracy to achieve fast simulated execution speed. This means that you can use the PV models for confirming software functionality, but you must not rely on the accuracy of cycle counts, low-level component interactions, or other hardware-specific behaviour [1].

The following features was developed on this simulator:

- **Hardware Virtualization**

  Hardware virtualization or *platform virtualization* refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources [2]. It allows PikeOS to run different operation systems on the same hardware simultaneously with just a little (or even without) modification of the operation system. It also provides almost no overhead in compare to software virtualization, so performance of the operation system running under hypervisor supporting hardware virtualization is comparative to the performance of the same native operation system.

- **Trust Zone**

  TrustZone technology is programmed into the hardware, enabling the protection of memory and peripherals. Since security is designed into the hardware, TrustZone

avoids security vulnerabilities caused by proprietary, non-portable solutions outside the core. Security can be maintained as an inherent feature of the device, without degrading system performance, enabling device manufacturers to build security applications, such as DRM or mobile payment as protected applications that run on the secure kernel [3]. PikeOS supports this technology for ARMv7 architecture and now it has been ported to ARMv8.

2.1.4 Alpha: Porting Results Juno

Currently Juno board is supported officially in PikeOS 4.1 with the following features and interfaces:

- Hardware Virtualization
- Trust Zone
- Serial driver

  Serial driver provides console support and allows establish basic communication between target and host. Usually it is used for debugging and administration purposes.
- Ethernet driver

  Ethernet driver provides network support.
- ElinOS BSP

  Board Support Packages (BSPs) contain the necessary adaptations to be able to run a Linux kernel on a specific target platform. ElinOS kernels contain some modifications to ensure a smooth operation with the ElinOS tools

2.1.5 Alpha: Testing on Juno

Juno board support has been testing with the PikeOS Generic BSP Test Suite.

This test suite is used for a regression testing as well as for a product testing.

The results are listed in Table 2: Benchmarks.

Test	Operation	Start	End	Result
arm-config-test	Run	17:52:00	17:52:38	OK
basic-linux-test	Run	17:52:38	17:54:17	OK
coherency-test	Run	17:54:17	17:54:54	OK
context-switch-bench	Run	17:54:54	17:56:13	OK
cpu-bench	Run	17:56:13	19:03:48	OK
decode-test	Run	19:03:48	19:04:31	OK
demo-linux-guest-test	Run	19:04:31	19:05:53	OK
demo-pikeos-guest-test	Run	19:05:53	19:06:45	OK
directio-test	Run	19:06:45	19:07:47	OK
fpu-test	Run	19:07:47	19:09:04	OK
hello-world	Run	19:09:04	19:10:06	OK
high-address-test	Run	19:10:06	19:21:12	Error
interrupt-forward-test	Run	19:21:12	19:21:53	OK
memory-test	Run	19:21:53	19:22:44	OK

Test	Operation	Start	End	Result
network-bench	Run	19:22:44	19:24:49	OK
p4bus-bench	Run	19:24:49	19:28:25	OK
p4bus-console-test	Run	19:28:25	19:29:19	OK
p4bus-ethDriver-network-test	Run	19:29:19	19:31:26	OK
p4bus-iomem-test	Run	19:31:27	19:32:38	OK
p4bus-mmaplseek-specvmem-test	Run	19:32:38	19:34:47	OK
p4bus-mmaplseek-test	Run	19:34:47	19:36:56	OK
p4bus-multiApp-vmchar-test	Run	19:36:56	19:37:49	OK
p4bus-multiFD-vmchar-test	Run	19:37:49	19:39:33	OK
p4bus-muxa-test	Run	19:39:33	19:41:53	OK
p4bus-network-test	Run	19:41:53	19:45:46	OK
p4bus-nonblock-vmchar-test	Run	19:45:46	19:46:59	OK
p4bus-severalMem-test	Run	19:46:59	19:48:46	OK
p4bus-sigkill-vmchar-test	Run	19:48:46	19:53:50	OK
p4bus-test	Run	19:53:50	19:56:45	OK
p4bus-vmapi-sysfs-test	Run	19:56:45	19:57:50	OK
p4bus-vmapi-test	Run	19:57:50	19:58:55	OK
p4bus-vmchar-fp-test	Run	19:58:55	20:00:09	OK
p4guest-console-test	Run	20:00:09	20:01:01	OK
p4guest-multiApp-test	Run	20:01:01	20:01:50	OK
p4guest-vmfp-test	Run	20:01:50	20:02:34	OK
stress-test	Run	20:02:34	20:03:44	OK
stress-test-smp	Run	20:03:44	20:05:32	OK
virtio-test	Run	20:05:32	20:06:15	OK

Table 2: Benchmarks

2.2 Fixed-priority and EDF for mixed-critical VM/task scheduler

2.2.1 PikeOS Scheduling

PikeOS is a virtualising embedded real-time operating system. It's basic mechanism for ensuring isolation of different applications is partitioning, which enforces isolation both spatially and temporally. This way, mixed-criticality systems can be constructed based on the guarantees the PikeOS partitioning provides.

Partitioning in PikeOS is two-fold: to partition CPU time, an ARINC653 based time partitioning is available, while other resources like memory, I/O access, and communication rights management are handled by resource partitioning. In this technical paper, we will concentrate on the scheduling functionality provided by PikeOS. The top-level system scheduling in PikeOS is time-partitioning, which uses a static round-robin fixed sequence schedule to determine which partitions are scheduled at which point in time. For some kind of dynamics to react to different system states or situations, PikeOS supports different time partition schedules that can be switched by a system partition, but there is no way to construct new time partition schemata at run-time. This static configuration is done intentionally to ease argumentation about guarantees needed for certification of the temporal

isolation and timing properties of the system. I.e., for simplicity, this paper will restrict the view to a single static time partition schedule.

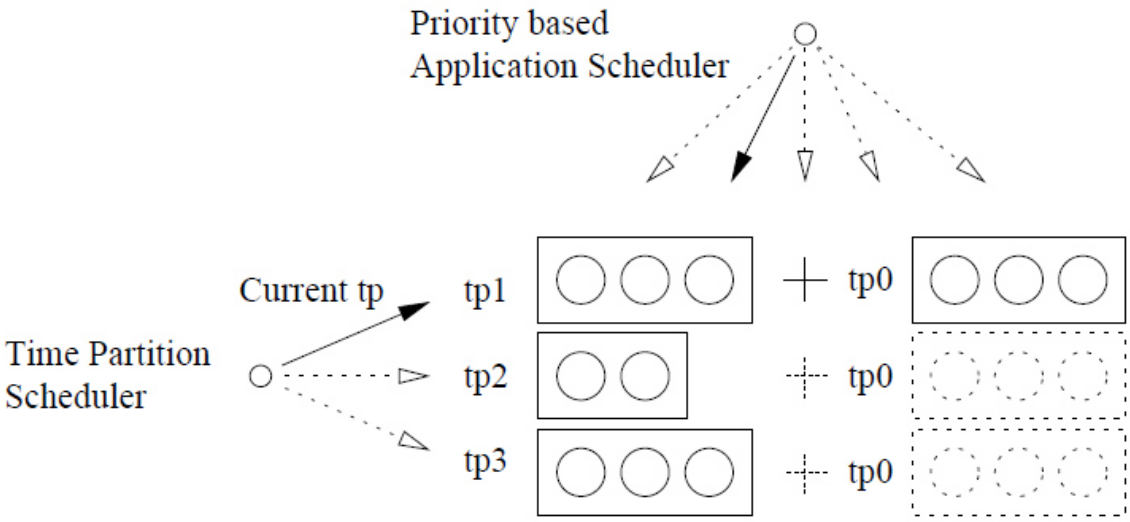

Figure 2: PikeOS time partitioning

The time partition scheduling in PikeOS has an extension over the ARINC653 partitioning: a time partition that is always active, which is called tp0. At any time, threads from the current time partition plus those from tp0 are eligible to thread scheduling. Figure 2 shows how PikeOS time partitioning works: the time partition scheduler selects, based on the static round robin schedule, one of the time partitions, and additional to that, tp0 is also active. From the set of threads selected this way, the second level scheduler, which is priority based, selects the thread with the highest priority.

TP0 is an important concept in PikeOS, because it can be used for two major tasks:

1. To run background tasks at low priority, i.e., a Linux partition that should be active only whenever the system has nothing else to do. This way, free CPU resources can be utilised without allocating explicit CPU time. This possibility is best-suited for non-time critical tasks. Figure 3 shows this possibility.
2. To run high-priority error handlers that are mostly inactive except in critical situations. E.g., a power failure handler could be allocated to tp0 with high priority, so that it can react quickly if necessary. Again, no CPU time would have to be pre-allocated to such tasks, because in the normal case, the error handler is expected not to run, and in the exceptional case, it can still react with minimal delay. Because such error handler have highest priority, they are automatically in the highest criticality class of the system, because it tp0, they are always active and could disrupt the whole system in case they went out of control. Figure 4 shows this use case of tp0 in PikeOS.

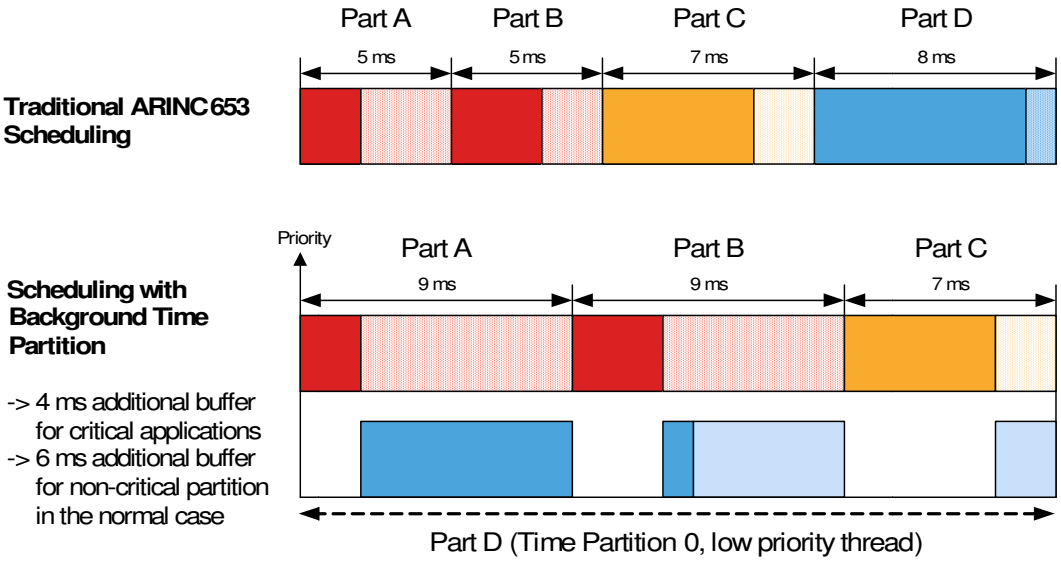

Figure 3: Background task in tp0, as an extension to traditional ARINC653 time partitioning

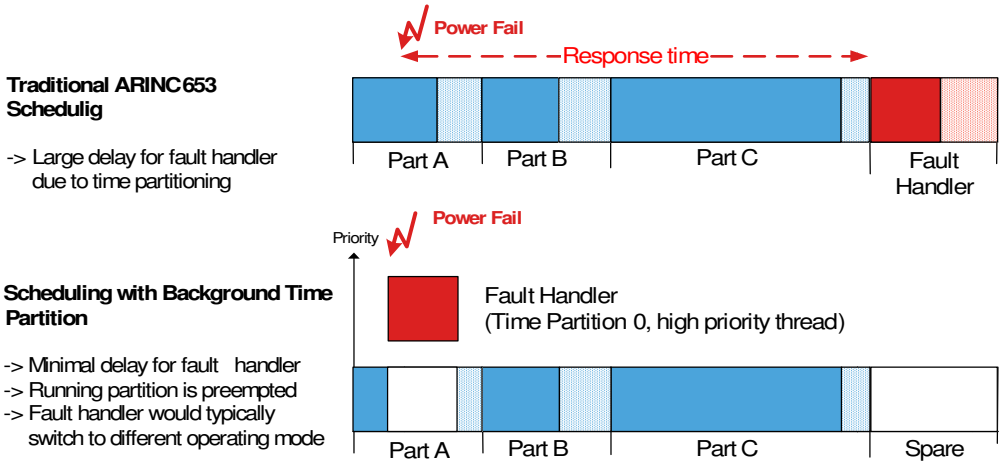

Figure 4: Fault handler in tp0, another extension to ARINC653 time partitioning

2.2.2 Earliest Deadline First (EDF)

Earliest deadline first (EDF) is a dynamic scheduling algorithm used in real-time operating systems to place processes in a priority queue. Whenever a scheduling event occurs (task finishes, new task released, etc.) the queue will be searched for the process closest to its deadline. This process is the next to be scheduled for execution [4].

In the course of the project, we want to evaluate more possibilities of scheduling in a real-time system. Together with our research partners, we want to pair up to implement and evaluate how EDF scheduling could improve the real-time properties and guarantees in a real-time operating system that uses time partitioning.

2.2.2.1 Event Handling in PikeOS

EDF is most attractive for sporadic, event-driven tasks, and for background activity that is not particularly time-critical, but for which starvation must be ruled out. It makes less sense for purely periodic activities, as such activities can be budgeted for in the time partition table. Therefore, EDF is most attractive for latency-sensitive threads in TP0 that should be able to preempt threads in other time partitions.

Application threads are routinely assigned to TP0 to serve as low-latency event handlers (e.g., interrupt threads). Such threads are typically lightweight, but there is no enforcement mechanism. In the case of an "interrupt storm," high-priority event handlers assigned to TP0 could starve other time partitions (in the current system).

By coupling EDF with mandatory budget enforcement, it is possible to assign threads to TP0 that previously would have been too dangerous to allow the ability to run at any time: starvation of threads in other time partitions can be prevented. In particular, this makes sense if the replenishment period of an event-driven thread in TP0 is shorter than the partition cycle. If the budget of an event-driven thread is larger than a time partition's slice, starvation is still possible. By making the replenishment period (substantially) shorter than the length of a time partition, it is ensure that no partition is starved in its entirety.

2.2.2.2 Design Considerations

Any EDF scheme needs to be opt-in, as compliance with standards such as ARINC 653 and POSIX must not be affected. For practical legacy/adoption reasons, it is highly desirable that older "EDF-oblivious" components continue to work even if some newer developments make use of the new EDF interfaces.

For efficiency reasons, it is desirable to assign multiple event-driven threads to the same reservation. Requiring an individual reservation[1] for each thread introduces additional pessimism. The classic fixed-priority view of the system is: "nothing of lower priority can delay me" (modulo controlled priority inversions).

EDF somehow needs to fit into this view of the system. Always prioritizing EDF above fixed-priority tasks would be too intrusive. Always prioritizing fixed-priority threads over EDF threads defeats the purpose of introducing EDF. Thus, EDF threads and FP threads need to co-exist within a shared priority space.

In the classic FP design, threads of equal priority are typically queued in FIFO order (at each priority level). There are some exceptions to this basic rule, e.g., in the case of priority inheritance as defined by POSIX. Round-robin with fixed time slices is also not uncommon, e.g, again POSIX. When integrating EDF into the design, three basic options exist: at configuration or integration time, declare one or more of the existing fixed priority levels to be

1. a "pure EDF priority level," with the obvious interpretation that threads (or reservations) of that (fixed) priority are queued in EDF order, and not FIFO order (with possible "deadline inheritance" exceptions),
2. a "shared EDF/FP priority level," where both EDF threads (or reservations) and fixed-priority threads exist, and where fixed-priority threads are treated as if they had an "infinite deadline" (i.e., EDF threads/reservations always have precedence at that priority level), or

[1] For example, on a single-CPU system, a single task attached to a reservation of 10ms every 100ms is guaranteed to be scheduled on the CPU for 10ms every 100ms [12].

3. a “shared FP/EDF priority level,” where both EDF threads/reservations and fixed-priority threads exist, and where fixed-priority threads are treated as if they had a deadline of zero (i.e., fixed-priority threads always have precedence at that priority level).

Of the three possible designs, only the second option makes sense. The third option is equivalent to having a pure FP level π and a pure EDF level π – 1 (i.e, at the next lower priority level), and the second options subsumes the first option if the user does not mix threads at the same priority level (which the first option entails anyway).

2.2.3 Evaluation Aspects

The implementation of the new PikeOS EDF scheduling will be subject to several evaluation metrics. Obviously, the most question to be answered is whether EDF is a gain for SYSGO customers when designing a system. By careful consideration, we believe this to be true, and it just remains to be shown that EDF actually works well together with time partitioning.

Secondly of all, we will evaluate whether the scheduler is as efficient as the current PikeOS one. There may be a scheduling overhead due to the more complex selection criterion. We want to evaluate how much difference this means. The numbers will help system designers to decide whether the overhead is worth the gain.

Thirdly, we will evaluate how well an EDF implementation is embeddable into a certified real-time operation system like PikeOS. To the best of our knowledge, there is no data indicating whether EDF can be easily certified according to the high standards that are used for PikeOS. To evaluate this, SYSGO will use some of its certification experts to examine the source code of the implementation and to see whether it would be usable in a certification environment.

2.2.4 Implementation

It is conceptually possible to implement an EDF scheduler in user space, but this may introduce overheads that may be too great to be useful. Therefore, an in-kernel implementation will be targeted.

In PikeOS 4.0, SYSGO introduced pluggable kernel drivers. The design is such that at configuration time, the final PikeOS kernel binary is linked together from the core kernel binary, plus the platform support package (PSP), containing lowest level timer and boot support, plus user defined kernel driver modules. This mechanism is available to customers, so no special support needs to be given when developing a kernel driver.

A kernel driver for scheduling would be special, because it needs to plug into the PikeOS kernel’s scheduler algorithm. For this, no direct support is currently available in PikeOS, so SYSGO will have to extend the kernel driver concept in such a way that the PikeOS scheduler can invoke callbacks into a pluggable scheduler driver to extend the scheduling.

Once such pluggable scheduler functionality is available, the EDF driver can be implemented as a kernel driver. The PikeOS kernel driver framework provides mechanisms for configuration of drivers using an XSD/XML based approach. This can be used by the EDF driver to establish the global reservation table and to pass in any other configuration data from the system integrator. Configuration information can be global and per-partition, the PikeOS configuration concept supports both.

2.2.5 Summary

In the project, we want to implement and evaluate EDF scheduling as part of PikeOS, in combination with time partitioning. We have identified that it is best an addition to the tp0 concept of PikeOS, for sporadic, not particularly time critical tasks for which starvation needs to be ruled out. Latency-sensitive threads in TP0 are most suited for running with EDF.

Within the PikeOS 4.0 kernel driver framework, it will be possible to implement the EDF scheduler extensions that the PikeOS kernel will need for EDF. The framework also provides the necessary means to configure the scheduling.

Chapter 3 Security for Mixed-Criticality

3.1 CycurLIB on PikeOS

In order to meet the security requirements defined in the D1.2, (i.e. system integrity that both the operating system and the run-time environment are not manipulated.), standard cryptographic algorithms (like AES, SHA-2, RSA, and ECC) are required. These cryptographic algorithms provide application programmers an easy and standard way to add security to applications, and SAFURE will integrate the most relevant algorithms into the run-time environment of PikeOS.

3.1.1 CycurLIB

CycurLIB is a cryptographic library developed by ESCRYPT, which is a collection of common cryptographic algorithms used in embedded systems where resources are particularly limited.

The library meets the following design criteria:

- Minimized ROM size
- Minimized RAM usage
- Compliance with MISRA-C:2012
- No dependencies on external libraries (including the C standard library)
- No dependencies on a particular microcontroller or operating system (e.g. independence of endianness, no assembly code)
- No dynamic memory allocation
- Fully re-entrant
- Easy to integrate in existing systems

3.1.2 Cryptographic Algorithms

Currently the cryptographic algorithms being ported into PikeOS are selected from CycurLIB according to discussion result of data integrity in D3.1. There are basically three aspects:

- Symmetric key encryption/decryption algorithm (i.e. AES, ChaCha20, Salsa20)
- Asymmetric key encryption/decryption algorithm (i.e. RSA, ECDSA)
- Message Authentication Codes (MAC) (i.e. HMAC, CMAC, Poly1305)

Please note that the cryptographic algorithms listed here are only the suggestion. The real ported cryptographic algorithms could be extended or reduced according to the real design and the use of PikeOS.

3.1.3 PikeOS File Provider

The PikeOS System Software allows applications to register themselves as file providers to the rest of the system. They can then be accessed using standard file system semantics to

implement all kinds of file systems. File providers are also used as the interface to I/O drivers such as for Ethernet, serial or CAN. PikeOS comes with a number of ready-to-use built-in and configurable file providers. [5]

- Internal file providers are part of the PikeOS System Software and are available as soon as the system starts up.
  - ROM File System (rfs prefix)
  - Shared Memory File System (shm prefix)
  - Property File System (prop prefix)
- External file providers are applications running in a partition. They allow the functionality of PikeOS to be extended
- System extension file provider are running in the PikeOS System Software. They also allow to extend the file system. [6]

3.1.4 Porting CycurLIB using PikeOS File Provider

In order to provide an easy and standard way for application programmers to add security to applications, the cryptographic algorithms from CycurLIB are integrated into PikeOS by using the structure of File Provider.

The PikeOS applications can use the functionality of the File Provider through the PikeOS File System API. Different types of file are managed by File Providers.

The cryptographic algorithms from CycurLIB will be integrated into PikeOS in the form of external File Providers which are applications running in a partition. Each function of the algorithm is implemented as a file. The file can be accessed by the other PikeOS application through the PikeOS File System API. For example, AES_CBC encryption function is implemented as a file called *file_aes_cbc_enc.* If other PikeOS applications want to using AES_CBC encryption function, they can use *vm_open(), vm_ioctl(), vm_write(), vm_read(), vm_close(), etc.* to access the file *file_aes_cbc_enc,* write inputs to the file and read output from the file.

By using File Provider, the applications do not need to know how the cryptographic functions are implemented. Instead, they just need to know the file name of the cryptographic functions.

In order to make it easy to use, a CycurLIB_Api interface will also be implemented. The interface will look similar as the normal cryptographic function with input and output parameters. This interface will take care of the sequence of calling the PikeOS File System API for each cryptographic function to set the corresponding parameters through the file provider to the CycurLIB function. In this way, PikeOS applications can directly use CyurLIB_Api without the knowledge about File Provider.

3.2 Architecture for Secure Boot

Secure boot is of interest for devices, where a malfunction has disastrous consequences for the user. Medical devices for example mostly use segregation (separation) to provide the right level of safety and security. By instrumenting the firmware or the operating system, the hacker is able to bypass every software security mechanism and to modify the functionality of the device, so that an infusion pump may provide too much medication to a patient, causing possible catastrophic consequences.

Secure boot is not only a means for securing the boot process. It can also be used to protect intellectual property and system secrets by encrypting the boot image. With this additional security the attacker will neither be able to start the system with instrumented code (in order to analyse the image content) nor will they be able to reverse engineer the application image, which they may read from the Flash device.

This section describes the secure boot principles and illustrates how the boot process for a PikeOS system is secured and how application loading can be secured in a chain of trust.

Some companies (e.g. Texas Instruments, Qualcomm etc.) don't provide any information about Secure Boot implementation in their SoCs without a NDA agreement. Therefore for the first development to achieve quick and portable results, we have chosen Freescale QorIQ platform as one with well documented implementation of Secure Boot. The designed Secure Boot concept is platform independent and will be portable with very limited resources to other platforms.

3.2.1 Secure Boot Components

While looking at the Trust Architecture, we will focus only on the components required for the secure boot process. A more detailed description of the Trust Architecture can be found in [7]. The Software and Hardware components, which are essential for the secure boot process, are:

Security Engine (SEC): The primary function of the SEC is to accelerate cryptographic operations. The SEC also supports a security violation detection function known as the Run Time Integrity Checker (RTIC). The RTIC uses the SEC's cryptographic hashing capability to periodically check the integrity of designated sections of system memory.

Security Fuse Processor (SFP): The SFP is used to program fuses passing keys and other secret values to other hardware blocks of the QorIQ processor. The values in a locked SFP cannot be read, modified or scanned.

Security Monitor: The Sec_Mon senses and controls the security state of the QorIQ. If a security violation is detected, configurable actions are executed. The possible actions range from SoC reset to more severe lock-out options, which can make the SoC unusable.

Pre-Boot Loader (PBL): Before the local cores are permitted to boot, the pre-boot loader (PBL) loads a reset configuration word (RCW) and Pre-Boot Initialization (PBI) commands from a non-volatile memory interface and does some basic chip configuration.

Internal Secure Boot Code (ISBC): The ISBC is an unmodifiable internal boot ROM. Its function is to validate a signature over the next code to execute.

External Secure Boot Code (ESBC): The ESBC is whatever the OEM programs it to be and is usually a boot loader, which implements functionality to validate the next software to be loaded. Freescale provides a modified U-Boot, which can be adapted to the hardware by the OEM.

Code Signing Tool (CST): The CST enables manufacturers to sign or encrypt the software for their products to ensure that only authentic software is allowed to run on the end product.

Secure boot is usually initiated by putting the processor into a specific state by setting the Intend To Secure (ITS) bit in the security Fuse Processor (SFP). If the ITS bit is sensed during power-on, the system jumps to an unmodifiable Internal Boot ROM, which contains the Internal Secure Boot Code (ISBC). The function of the ISBC is to validate the authenticity of the next code to be executed. The described sequence of operation is designed to be unmodifiable and is called the root of trust.

3.2.2 Code Signing

In order to guarantee authenticity of the code to be executed, secure boot relies on validating a signature, which has been generated from the system image (which shall be validated). The Freescale Code Signing Tool (CST) provides all required means to digitally sign code and apply encryption. The RSA-algorithm is the most common sign and verify tool, which is also used in the Freescale CST. The CST can establish a Public Key Infrastructure (PKI) tree of keys and certificates needed for code signing in addition to generating digital signatures across data provided by the OEM. The signatures generated by the CST can then be included as part of the end product software image [8]. Figure 5 shows the code signing process for a "System Image" using the Code Signing Tool. The generic signature process can be summarized as:

1. Calculate a hash over the system image
2. Sign the previously generated hash with the private key
3. Calculate hash over public key and burn this into Super Root Key (SRK) fuse

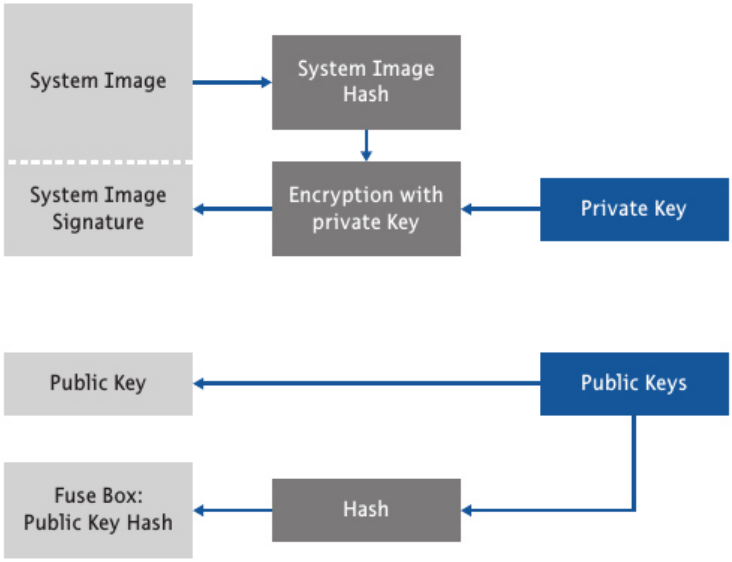

Figure 5: Code signing process

Rather than signing the whole image, a hash algorithm (e.g. SHA256) is used to generate a hash over the System Image. Then the hash is signed with the RSA private key and appended to a header, which is known as the CSF- header. During the boot process the public key will be used to validate the hash. The header and the system image are then written to the flash. Therefore a hash is generated from the public key, and programmed into a fuse block. Although Figure 5 shows the hash being calculated over system image, it is possible (even advantageous) to encrypt portions of the image with the One-Time-Programmable- Master-Key (OTPMK), which is a persistent secret value held in the SFP. This prevents attackers from stealing code from flash for reverse engineering.

3.2.3 The Secure Boot Process

Starting from the root of trust, secure boot shall make sure that the firmware to be loaded is authentic. Either it is the original image or it is an updated image, which must have gone through the same code signature process as described in the previous chapter. The secure boot process can be divided into two steps. The pre-boot phase, which initializes the SoC for

the secure boot process and the ISBC phase, which performs the signature validation over the firmware, which was named system image for code signing.

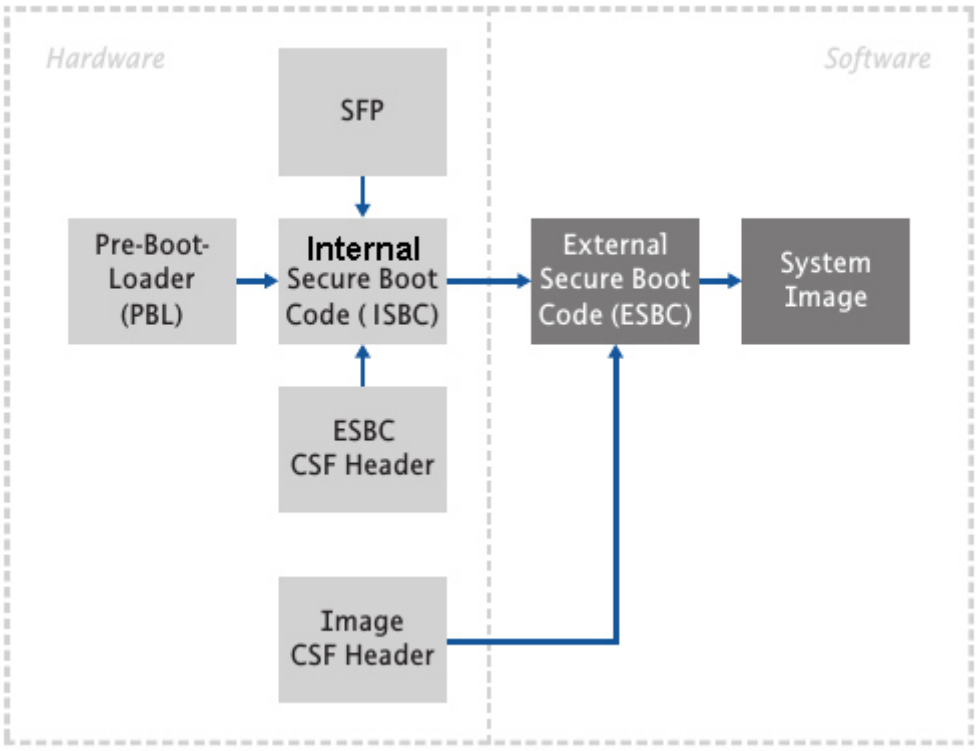

Figure 6: The secure boot process

Freescale provides reference firmware code, which can be adapted by the OEM to support the customer's hardware. This firmware code is logically divided in two parts. The first part, which is executed by the ISBC, performs further device configuration and code authentication using similar mechanisms as the ISBC. This part is called External Secure Boot Code (ESBC). As Freescale uses U-Boot code, it is also named "Trusted U-Boot". The second logical part is the "Trusted U-Boot Client". This Trusted U-Boot client is validated by the ESBC and loaded into main memory for execution.

Pre-Boot phase

Before the local cores are permitted to boot, the reset control logic blocks all activities until fuse values are sensed. The fuse value, which indicates the intention to use secure boot, is the Intend To Secure (ITS) bit. If the ITS bit is set, interfaces, memory permissions and MMU configurations are locked down.

The Pre-Boot-Loader (PBL) loads the configuration values from the Reset Configuration Word (RCW) and the Pre- Boot-Initialization (PBI) commands, thereby performing minimum chip configuration, making sure that the ISBC knows the location of the CSF header. After this setup, the PBL enables Core 0, which begins executing code from the hard wired ISBC.

ISBC phase

As described earlier, the ISBC is considered inherently trusted, because it cannot be modified.

Apart from some platform self-test and policy checks, the main task of the ISBC is the signature validation of the firmware. As shown in Figure 7, the information needed for the validation of the firmware is taken from the CSF header. The validation process uses CPU 0 to execute the ISBC code, while the security monitor monitors the security state during this

process and the Security Engine is used to decrypt the image or the hash values. The detailed execution process is shown in Figure 7 and has the following steps:

1. The ISBC executes self-tests
2. A hash is calculated over the public key, taken from the CSF header
3. The calculated hash is compared with the hash in the Super Root Key (SRK) (see Figure 5)
4. If the hashes are identical, the hash for the CSF header and the system image is calculated
5. Using the public key, the system image signature (see Figure 5) from CSF header is decrypted
6. The hashes of the signatures from step 4 and 5 are compared
7. If the hashes are equal, the execution is passed to the ESBC

ESBC validation process

The ESBC is mainly a Freescale modified U-Boot, which performs minimal additional chip configuration such as mapping physical memory, initializing the network interfaces, data path infrastructure and loading next-stage software (Trusted U-Boot Client) into main memory.

In Freescale's reference code, this client validation process uses the same mechanisms as the ISBC (security monitor to monitor the security state and the Security Engine to decrypt the image or the hash values). The ESBC has the same CSF header format prepended to it. The public key used for this validation can be the same as used by the ISBC, or it can be a new public key from the trusted U-Boot client's CSF header.

If the signature passes, the Trusted U-Boot jumps to the entry point within the Trusted U-Boot Client and begins execution. At this point, the developer's authentic device configurations, OS and applications can be started [9]

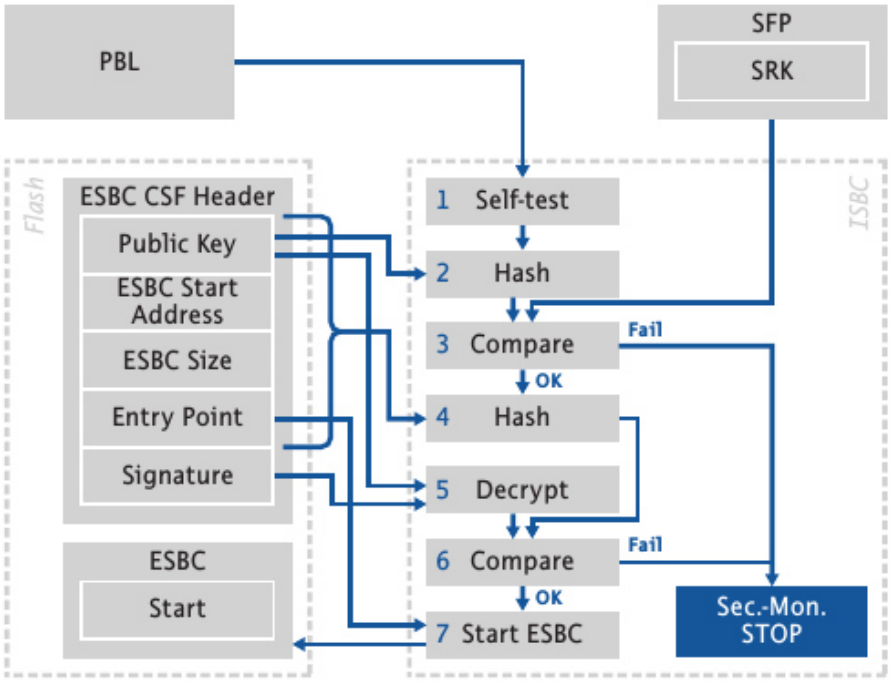

Figure 7: The validation process

Chain of Trust

The ISBC is SoC internal unchangeable code and thus called the root of trust of the secure boot process. The ESBC is the first external piece of code, which runs through a validation process and thus is also secure. If each software module is validated by the previously loaded software, we build a chain of trust, which guarantees that each software module is authentic and trusted (Figure 8).

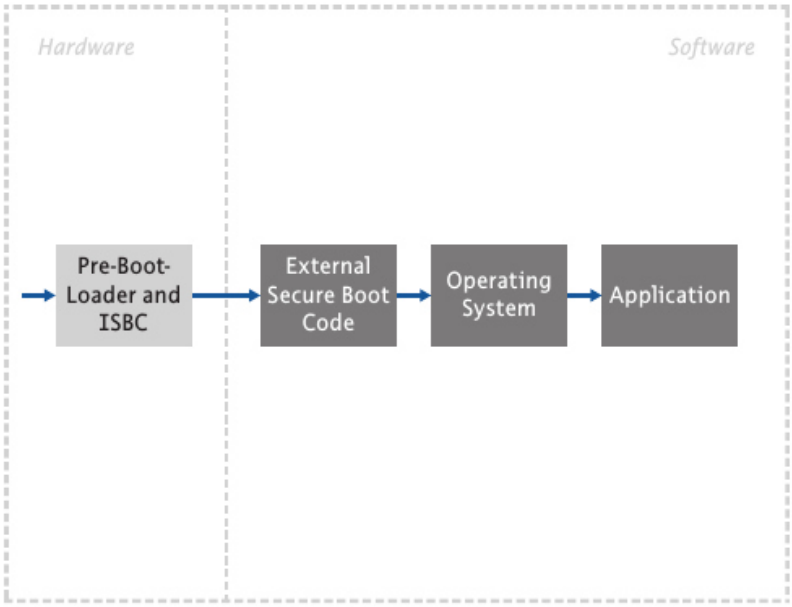

Figure 8: Chain of Trust

The ESBC is the first link in the chain of trust verified by the ISBC. By implementing appropriate validation algorithms in the ESBC, this can validate the operating system, and by implementing appropriate validation algorithms in the Operating system image, this can be used to validate the application.

3.2.4 PikeOS Security

PikeOS is a real-time hypervisor, which is based on separation kernel architecture. Besides offering real-time capabilities, PikeOS can separate and isolate a defined set of hardware resources (e.g. memory, CPU-cores, processing time, interrupts, etc.) into partitions (Figure 9). This separation of resources ensures that partitions do not interfere with each other. The communication between the partitions follow a white list policy, which ensures that communication is only possible, if the communication means are generated and strictly assigned for the communicating partitions. A partition can host user applications in several flavours. Hosting legacy code in a bare metal environment is also possible as hosting several APIs for PikeOS (ARINC653, AUTOSAR, POSIX, JAVA, ADA, etc). Even complete operating systems like Linux can operate in a partition. Of course, multiple instances of the aforementioned runtime systems can reside in its own PikeOS partition with each owning its dedicated resources.

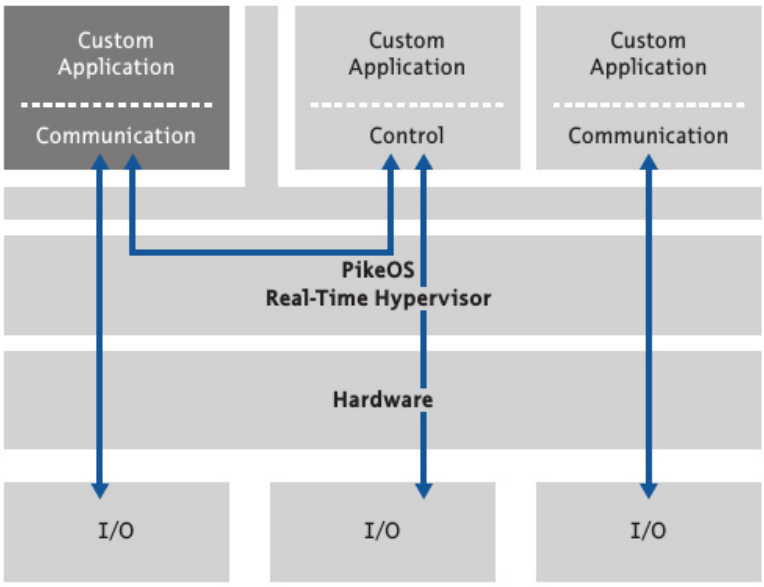

Figure 9: Security by separation and controlled information flow

Partition separation and isolation restricts hacker attacks using malicious application software on one partition, isolating them from all other partitions on PikeOS.

The strict control of PikeOS over hardware resources, processing time and communication channels adds “security by separation” to the “security by design” concept for IoT devices. For a state of the art microkernel based operating system, security by design includes aspects like malware protection, encryption, monitoring, etc. In addition, PikeOS offers security by separating sensitive data from insensitive data. A hacker who is able to penetrate a PikeOS Linux Partition through a known security leak, will neither be able to access hardware resources, which are not assigned to the partition, nor will he be able to access PikeOS resources and also he will not be able to access other partitions, if this has not been explicitly configured for the infected partition.

3.2.5 PikeOS Secure-Boot

As PikeOS increases the cost and effort for hacking a system disproportionately, the modification of firmware, operating system or application image might be a viable alternative for hackers to break into a PikeOS system. Application of secure boot to a PikeOS system will ensure that nobody can change the system software without help and information from the OEM.

The PikeOS partitioning concepts allows the integration of third party applications into a partition. When involving a third party into application development, we need to add the third party code into the chain of trust. Thus, we need modularity of the software components, which are loaded one after the other. Another important aspect is that we do not want to share the private keys, which we have used for signing own code (boot loader and PikeOS image). Either the third party has his own pair of keys or we need to provide a pair of keys to the third party. The third party will use these keys for generating the CSF header for his application. In order to keep the system modular and flexible, each component (Bootloader, PikeOS image and application) will be a standalone executable, which is stored in non-volatile memory. The validation of the boot loader and the PikeOS image follow the earlier described secure boot process. The validation process for the third party application is implemented as a user application running in a PikeOS partition (Figure 10).

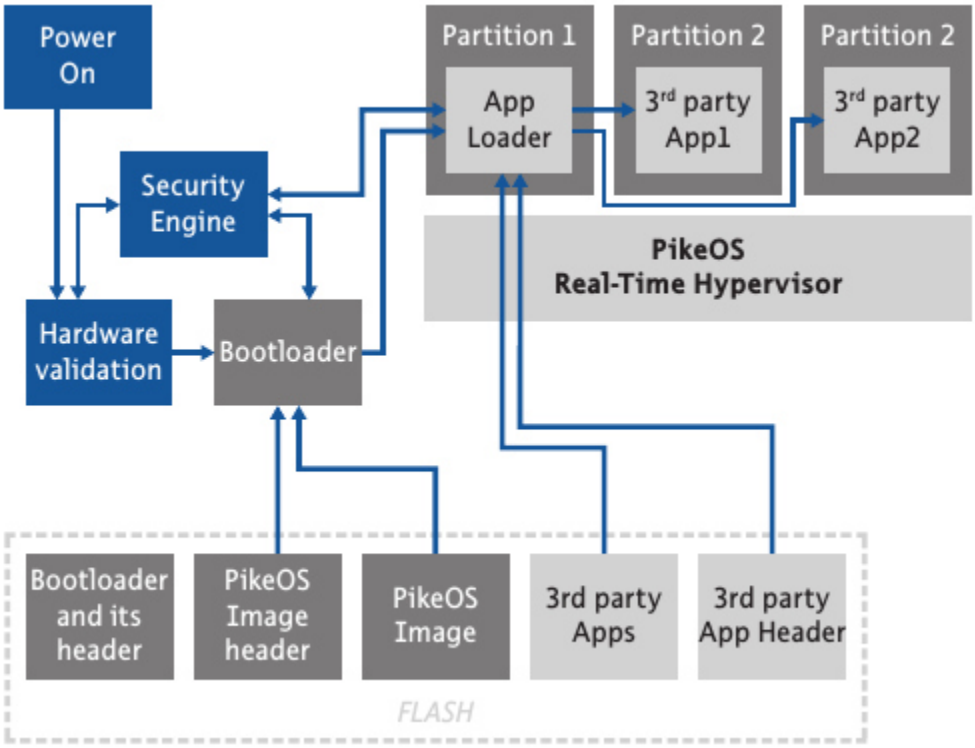

Figure 10: PikeOS Secure boot with application validation

This user application loader will need the ability to control the execution in other partitions. To validate the user application image, the application loader needs to implement functionality to use the cryptographic accelerator of the security engine. It will also be required to access the SFP, which stores the key hashes burned into the fuses. If the validation fails, the security monitor may be used to indicate the error and take appropriate action. The first step of the loader is to read the following information from an internal PikeOS file system:

- The location of a CSF header, which will be used for the validation of the corresponding user application.
- The hash of the Public Key.
- The partition number which will host the validated user application.

After validating the user application, the loader copies it into the corresponding partitions memory and starts the partition.

3.2.6 Conclusion

Using secure boot is an effective way to protect an IoT device against firmware level modifications, which are neither visible to the operating system nor to the user applications. A SoC internal hardware makes sure, that the secure boot process starts from a guaranteed secure root of trust and validates the software, which will be loaded and operated in the next step. By enabling the validated software to validate the next software package, the very last software package can rely on a chain of trust, which is authentic and secure.

Adding the user application to the secure boot chain of trust, is achieved easily by isolating these application into a PikeOS partition. Loading and execution the user application is performed by an application loader, which ensures that only validated applications are executed. Compared to a standard microkernel based OS, the PikeOS real-time hypervisor

offers additional security by its partitioning capabilities. The partitioning concept of PikeOS is well recognized by the security certification authorities to be certifiable up to Common Criteria Evaluation Assurance Level 6.

Here is described the configuration and execution of secure boot on QorIQ architecture. Comparable features of course exist on other SoC like the Intel Core-i or the TI OMAP architecture. PikeOS of course runs on these SoC's and provides the full bandwidth of security described in this section.

3.3 Architecture for Secure Flash Update

Currently, software updates for automotive ECUs are predominantly performed during service intervals within the service station. However, in order to react timely against emerging vulnerabilities and new threats, it is desirable to perform updates more frequently, ideally using a wireless connection to a remote server. To prevent attackers from modifying the firmware, security becomes a crucial part of the firmware update process.

The firmware update system consists of a Backend Server (operated by the OEM or a service provider) and the firmware update components within the embedded system.

First, the firmware has to be transferred from the Backend Server to the embedded ECU. This can be done using a regular Internet connection (e.g., via GSM) to the Backend Server. In order to preserve the integrity and authenticity of the firmware, digital signatures or MACs can be used. Optionally, the firmware can also be encrypted in order to perverse the confidentiality and thus prevent reverse engineering of the software functions. Hence, the transferred data include the firmware itself (possibly encrypted) and a digital signature or MAC value which is attached to the end of the firmware.

Second, the firmware update is forwarded to a Secure Update application that (optionally) decrypts the firmware update. Then, the verification of the digital signature or MAC takes place. To accomplish this, the transferred digital signature is verified using standard cryptographic algorithms (like RSA, ECDSA, EdDSA) or – in the case of MACs – the MAC value of the message is calculated and checked against the transferred value. If both values are identical, the firmware update is considered authentic and the flashing of the firmware can be performed.

The architecture allows both asymmetric and symmetric algorithms. Depending on the platform (RAM size, ROM size, CPU speed), either the first or the latter is better suited. In the case of asymmetric cryptography, only the public key has to be stored in the embedded system. It is no problem if a potential attacker can read this public key. However, it has to be ensured that this key cannot be modified, because otherwise an attacker could replace it with his own public key. Similarly, the key used to generate the MAC value has to be securely stored and even protected from read-out by an unauthorized party. The rationale here is that an attacker could use the key to calculate a legitimate MAC value for a manipulated firmware update.

Chapter 4 Mixed-Critical Run Time Engine

4.1 Hardware Support for Mixed-Criticality Multicore Systems

4.1.1 Hardware Counters

The number of hardware performance monitoring counters (PMCs) available in the SnapDragon processor is very limited. For instance, cache and memory accesses can be counted, but not whether L1 and L2 cache accesses turn out to be hits or misses. This complicates the development of our methodology to measure the impact of contention in the access to shared resources.

In order to interface PMCs we have developed a library with the following interface to read/write PMCs (only main functions listed here). Names are self-explanatory. Further details are provided in the library source code and will be conveniently documented in the final prototype:

void pmu_start();
void pmu_stop();
void pmu_reset_counters();
void pmu_set_counter(unsigned int event, unsigned int counter);
void pmu_activate_counter(unsigned int counter);
void pmu_stop_counter(unsigned int counter);
uint64_t pmu_read_counter(int counter);
void pmu_write_counter(int counter, uint64_t value);

Such library can be found in **BSCmicrobenchmarks.zip** under **src/pmu.h**, **src/pmu.c**
In order to quantify the impact of contention in the access to the different shared resources we have developed several microbenchmarks devised to stress each specific resource separately. This allows estimating the maximum delay a request to a particular shared resource can suffer, and so these data is used to feed the templates/signatures needed for upper-bounding contention impact.

So far stressing benchmarks have been devised to account for contention in the access to the shared L2 cache and to the shared memory controller. Those benchmarks have the following structure:

```
R1 = 0;
for (i=0; i<N; i++) {
    reset PMCs
    for (j=0; j<M; j++) {
        R2 = Load [@A+R1]
        R1 = R1+STRIDE
        R2 = Load [@A+R1]
        R1 = R1+STRIDE
        …
        R2 = Load [@A+R1]
        R1 = R1+STRIDE
    }
    read PMCs
}
```

Those benchmarks use a sufficiently large data vector starting in address @A. Measurements are collected N times (e.g., 10) since a measurement can be polluted, from time to time, by the Linux OS running below. M and the number of LOAD operations in the loop are set to values sufficiently high so that the overheads of the loop and to fill the instruction L1 cache become negligible (e.g., M=1000 and 16 LOAD operations). The particular PMCs read and reset depend on the contention that is to be measured in a particular experiment. Finally, STRIDE relates to the distance between memory objects accessed so as to make sure that they either hit in L1, miss in L1 and hit in L2, or miss in L1 and L2.

Some details of those microbenchmarks are omitted in this description for the sake of simplifying the explanation. The actual microbenchmarks developed so far can be found in **BSCmicrobenchmarks.zip** under **src**/ folder. Preliminary documentation describing each one of them can be found under **src**/**doc**/ folder.

4.1.2 Signatures and Templates

As described in D3.1 (delivered by M15), timing integrity due to multicore contention is accounted for developing resource usage signatures and templates. We refer the reader to D3.1 for details on signatures and templates.

We can build our signatures and templates either by running microbenchmarks with the specific number of requests that we want to upper-bound or by counting the number of requests to be upper-bounded and multiplying it by the maximum per-request delay. Our experience on a different architecture shows that the former approach may not be fully precise under some circumstances, so we build upon the latter.

Signatures and templates will be built upon the delay per request obtained from those microbenchmarks presented in previous subsection.

4.2 Alpha RTE Prototype

Time Critical Systems are characterized by stringent timing requirements expressed as deadlines for the applications running in the system. Classical time-critical systems require full time isolation that should guarantee that no task or application can delay another task or application.

With mixed-critical time-critical systems, the applications / tasks composing the system run with a different level of criticality. As a consequence, the global timing requirement of the system would now translates into no task should be allowed to delay any task with equal or higher degree of criticality.

We further relax this requirement by allowing low critical tasks to alter (slow) the timing behaviour of high critical tasks, as long as it does not endanger the high-critical task deadlines.

The **Run Time Engine (RTE)** is in charge of guaranteeing the high-critical task deadlines by proactively acting during each time slot on the lower critical task scheduling. We consider three different kinds of task appearing in Figure 11:

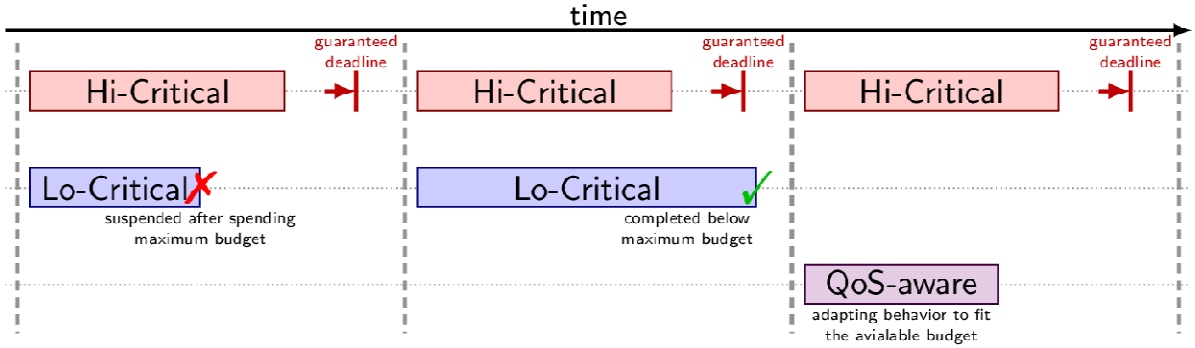

Figure 11: RTE Behaviour

- **High critical task**, which deadlines have to be guaranteed. Their scheduling shall not be altered by the RTE.
- **Low critical task** is allowed to run up to an available budget in term of resource access. If the low-critical task can complete without spending the entire allocated budget, it does so, else it is suspended by the RTE at least until the next time slot (when a new budget will be available).
- **QoS-aware** Firm real time tasks are low critical tasks that adapt their behaviour to the remaining resource budget. Rather than reactively suspending the task when it reaches its budget, the task can shift to some kind of degraded mode that is less time consuming but usually provides a less accurate result.

The available budget for low-critical and QoS-aware task is defined during a first profiling phase of the critical application and the target architecture relying on the signatures presented in this chapter. It heavily relies on Performance Monitor Counters to determine the number of access to each shared resource.

The first release of the RTE will only encompass High-Critical and Low-Critical tasks, and QoS-aware tasks will be added in a later time. The whole time-critical prototype is presented in Chapter 5, including the hardware and software environment.

We will also mainly focus on resource access and timing budget first, but the RTE will be design such as the requirements can easily be shifted to power and temperature requirements. In such a context, thermal models as defined in WP2 will be used to compute the available budget, and hardware thermal probes will be used in lieu of classical hardware counters.

4.3 Adding thermal protection to mixed criticality scheduling

Temperature adds a new dimension to the Mixed-Criticality scheduling problem. Sharing a hardware platform will cause thermal interferences between criticality levels. To understand what we mean by thermal interferences we first explain thermal constraints and thermal protection mechanisms which are in place in modern processing platforms.

Modern processing platforms tend to have high power densities. Therefore, in periods of high processor activity, switching too many transistors at a time generates more heat than can be dissipated, possibly damaging the chip due to exceeding the maximum safe temperature. To alleviate this, hardware driven Dynamic Thermal Management (DTM) is used. DTM resorts to techniques (e.g., sharp speed throttling) that severely impair performance.

In a mixed-critical setting, DTM significantly complicates the scheduling problem. We can imagine a scenario where a low criticality task over exercises the CPU and causes DTM to be triggered. Now, while DTM is active, a high criticality task may be scheduled for execution. Consequently, the high criticality task will experience degraded performance; which in the worst-case, may lead to a deadline miss. In this manner, the low criticality task can interfere with the execution of hi criticality task.

4.3.1 Thermal protection mechanism

To ensure thermal protection, we adopt a thermal analysis based approach. Based on the application and task model, we characterize the worst-case temperature for executing Hi-Critical tasks. If the worst-case temperature is higher than the temperature threshold where DTM is enabled, the mixed criticality taskset is deemed "unsafe". If the worst-case temperature is lower than the temperature threshold, the additional temperature margin is used to assign a *thermal budget* to the remaining Lo-Critical tasks.

4.3.1.1 **Thermal model**

We assume that the processor has active and idle states; with each state having a different power consumption. We approximate the heat flow in a system by the following differential equation:

$$\Omega \frac{dT}{dt} = -G(T - T_{amb}) + P$$

Equation 1

Where Ω, P, G and T_{amb} are thermal capacity, power consumption, thermal conductance and ambient temperature respectively. If power consumption is assumed constant within a given interval, then Equation 1 has the following closed form solution:

$$T(t) = T^{\infty} + (T(t_0) - T^{\infty}).e^{-a.(t-t_0)}$$

Equation 2

4.3.1.2 **Adding thermal protection to RTE Prototype**

We can use [10] to determine the worst-case maximum system temperature, when only Hi-Critical tasks are executing on the system. [10] presents an approach to determine the task execution trace which would lead to maximum system temperature; given task scheduling constraints and system thermal model. This *thermal worst-case* execution trace is then simulated to determine the maximum temperature. The maximum temperature is then used to determine/assign *thermal budgets* to Lo-Critical and QoS-aware tasks. A thermal budget is defined as the maximum allowed temperature increase caused by the execution of a Lo-Critical/QoS-aware task. Thermal budget is chosen such that DTM is never triggered. At runtime, a Lo-critical task will be suspended if it exceeds its thermal budget. A QoS-aware task may degrade its service to meet thermal budget constraints.

Chapter 5 WP4 Time-Critical Prototype

As presented in previous chapter, the time-critical prototype encompasses high-critical, low-critical, qos-aware tasks as well as the RTE engine. Figure 12 details the target environment for this WP4 prototype.

Mixed-critical tasks as well as the RTE will run as single or multiple partitions applications or threads on top of the PikeOS operating system. We will develop a driver for PikeOS to provide privileged access to the PMC hardware counters. The prototype will target the Dragonboard 810 architecture thought the ARM64 PSP present in the PikeOS operating system.

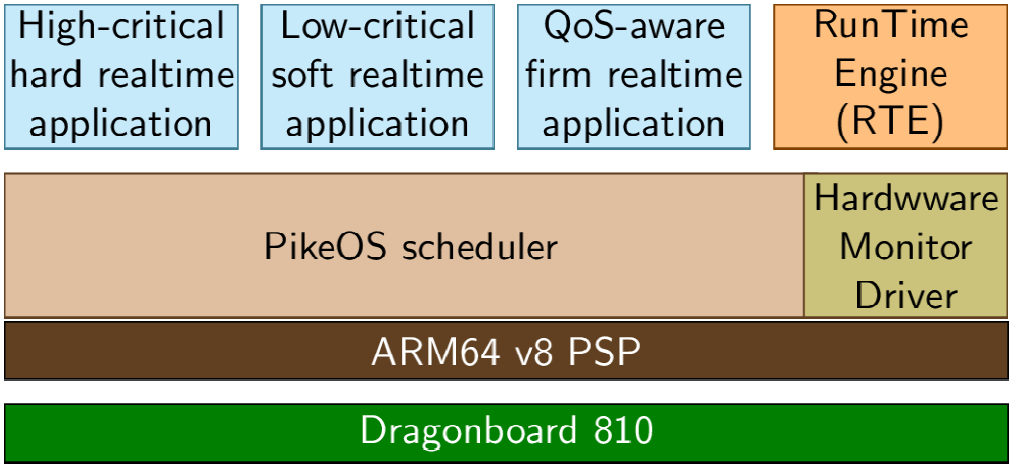

Figure 12: Target environment for the Time-Critical prototype

As the ARM PSP will first be developed for the Juno board, we will start with a baremetal prototype where the RTE will also be in charge of scheduling the baremetal tasks. As a consequence, several versions of the application will be provided. The selected applications are presented in the next subsections.

5.1 Hard Real Time High-critical Application: Flight Management System

The selected hard real-time **high-critical** application for the WP4 time-critical prototype is a mark-up Flight Management System (FMS) application from the avionics domain.

The purpose of the Flight Management System (FMS) in modern avionics is to provide the crew with centralized control for the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management, and geographical situation information. Taking charge of a wide variety of in-flight tasks, the FMS allows us to reduce the workload of the flight crew allowing us to reduce crew size.

The FMS is especially responsible for services that allow in-flight guidance of the plane. From pre-set flightplans (take-off airport to landing airport), the FMS is responsible for plane localization, trajectory computation allowing the plane to follow the flightplan, and reaction to pilot directives.

The FMS application is constituted by 25 time-critical tasks that are regrouped into different task groups as presented in Figure 13.

The **Sensors** task group is in charge of generating all the localization data from various sensors (Anemo-barometric sensors, IRS (Pure Inertia Reference System), GPS (Global Positioning System), HYB (Hybrid Inertia Reference System), Doppler sensor)

The **Localization** task group is in charge of analysing outputs of sensors to generate the most probable position of the aircraft (BCP). This localization data is composed of: Position (latitude, longitude, and altitude), Attitude (Pitch, Roll and Yaw angular rates), Velocity (Ground speed and Vertical Speed), Acceleration (lateral and longitudinal), and Wind related data (speed and angle).

Note that a single sensor may not provide the full Localization information. The Doppler sensor for instance does not provide any position related information such as longitude and latitude. It however provides very accurate velocity (speed related) information. The role of the Localization task group is therefore to merge information from the sensors with different trustworthiness levels.

The purpose of the **Nearest** Airports task group is to continually build a list of the nearest airports, during the flight. This information is useful in case the pilot decides to have an impromptu landing for some reason. The tasks from this task group do not participate directly in flight management, and the computed output only has to be sent to the display.

The **Flightplan** task group is in charge of managing and processing modification requests on the flightplans that are pre-set routes used to guide the airplane. Three different flightplans coexist concurrently on the system:

- The active flightplan is the flightplan currently used to guide the aircraft.
- The secondary flightplan is an alternative route toward the destination. It could consider for instance an alternative-landing runway on the destination airport, which has a significant impact on the target airport approach procedure.
- The temporary flightplan is an intermediate flightplan allowing the crew to enter a new flightplan and check for the modification before applying.

The flightplan task group is only composed of aperiodic tasks that correspond to the pilot's modifications to the pre-set flightplans.

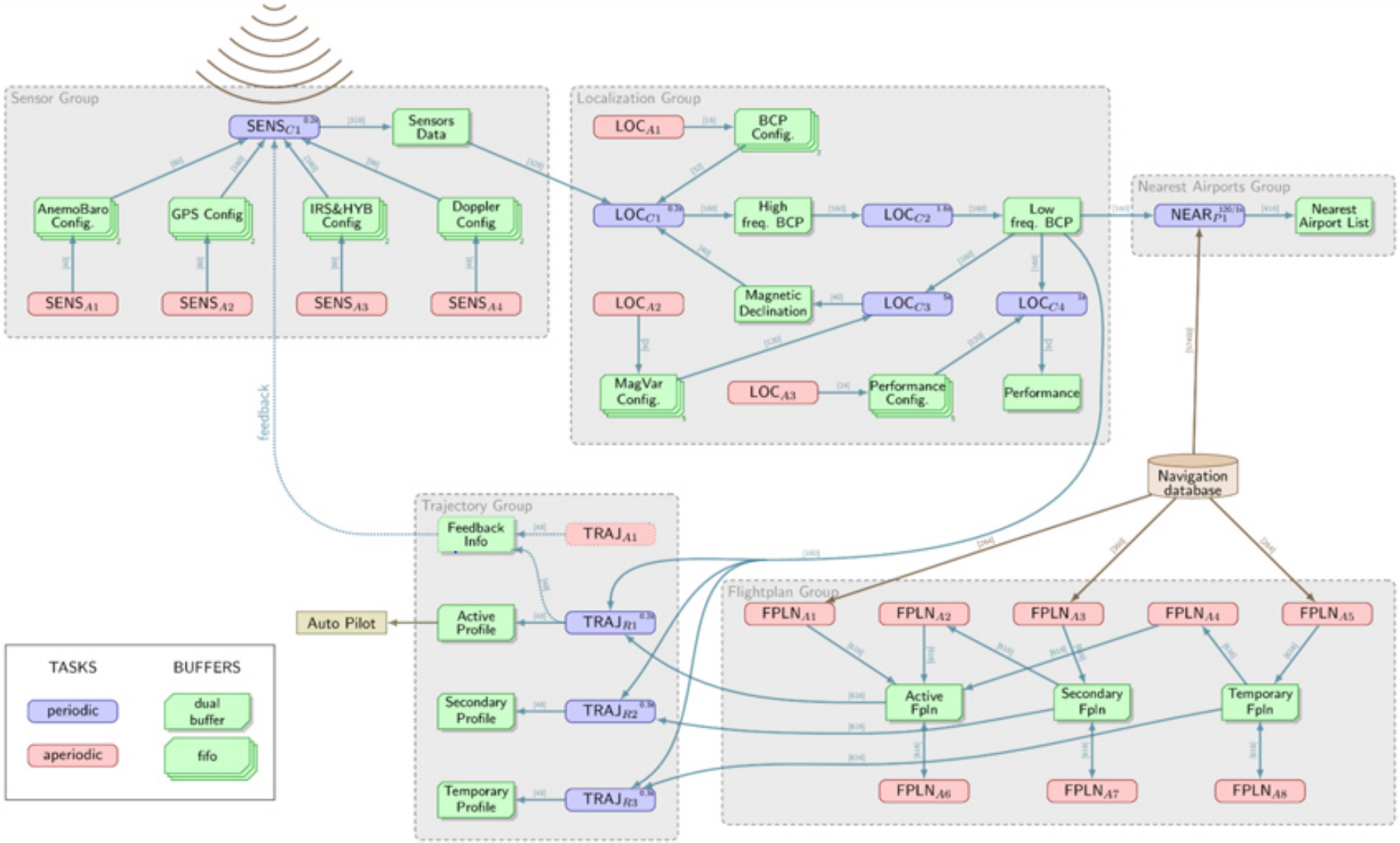

Figure 13: FMS Application Task flow graph

The **Trajectory** task group aims at computing both lateral and vertical profiles for the three flightplans set by the flightplan task. The lateral profile is composed of waypoints as well as leg information (path before, after and between the waypoints). The vertical profile provides altitude information (cruise altitude interceptions, crossing altitudes and slope angles) as well as performance information (estimated time of arrival, estimated fuel on board).

Trajectory computation is performed for each of the three above defined flightplans. The inputs of trajectory computation are both the flightplan and the best computed position (BCP) of the plane that comes from the localization task group. The computed trajectory tries to tangent the pre-set flightplan while respecting passenger wellness (limiting roll and pitch) as well as physical limitation of the plane actuators such as flaps. The trajectory information is later used by the plane autopilot to actually interact with these actuators.

The FMS application also embeds a large **Navigation Database** that does not fit in any cache structure. It is both linearly and regularly accessed by task from the Nearest airport task group, as well as randomly and sporadically accessed by tasks of the Flightplan task group. Accesses to this database in the main memory is very interference prone.

All the tasks composing the FMS have stringent **real-time requirements**. Table 3 and Table 4 respectively show the time requirements of periodic and aperiodic tasks composing the application.

Periodic Task	Period / Deadline
$SENS_{C1}$	200ms
LOC_{C1}	200ms
LOC_{C2}	1.6s
LOC_{C3}	5s
LOC_{C4}	1s
$TRAJ_{R1}$	200ms
$TRAJ_{R2}$	300ms
$TRAJ_{R3}$	300ms
$NEAR_{P1}$	1s

Table 3: FMS: Time requirements of periodic tasks

In the FMS, periodic tasks are characterized by an activation period as well as a deadline that always corresponds to the next activation period.

Aperiodic Task	Maximum activations	Deadline
$SENS_{A1}$	2 per 200ms	50ms
$SENS_{A2}$	2 per 200ms	50ms
$SENS_{A3}$	2 per 200ms	50ms
$SENS_{A4}$	2 per 200ms	50ms
LOC_{A1}	2 per 200ms	100ms
LOC_{A2}	5 per 5s	50ms
LOC_{A3}	5 per 1s	50ms
$FPLN_{A1}$	once at initialization	1s
$FPLN_{A2}$	1 per 1s	1s
$FPLN_{A3}$	once at initialization	1s
$FPLN_{A4}$	1 per 1s	1s
$FPLN_{A5}$	1 per 1s	1s
$FPLN_{A6}$	1 per 1s	50ms

$FPLN_{A7}$	1 per 1s	50ms
$FPLN_{A8}$	1 per 1s	50ms
$TRAJ_{A1}$	once at initialization	50ms

Table 4: FMS: Time requirements of aperiodic tasks

All aperiodic tasks are sporadic, and are characterized by a maximum number of activation per period of time. This period of time is usually defined by the period of the periodic task consuming the data produced by the aperiodic task. Aperiodic tasks also have to respect a deadline provided by Table 4.

We will develop both a baremetal and a single-partition PikeOS native version of the application.

5.2 Soft Real Time Low-critical Application: Bi-Quadratic Distributed Control System

The selected soft **low-critical** real-time application for the WP4 time-critical prototype is a control-command application implementing a bi-quadratic distributed control system. The tasks composing the applications, appearing in Figure 14, are:

- The **Generator** process is generating the input data. It either self-generates the data on the first pass, or iterates on the received data on the next passes.
- The **Splitter** process splits the data received from a FIFO to make it globally available to all the filtering tasks.
- The **LoPass** and the **HiPass** processes are applying some bi quadratic filtering to the data.
- The **Aggregator** process fuses the previously computed filtered data and sends it back as feedback to the generator task.
- The **Display** process finally displays the fused data.

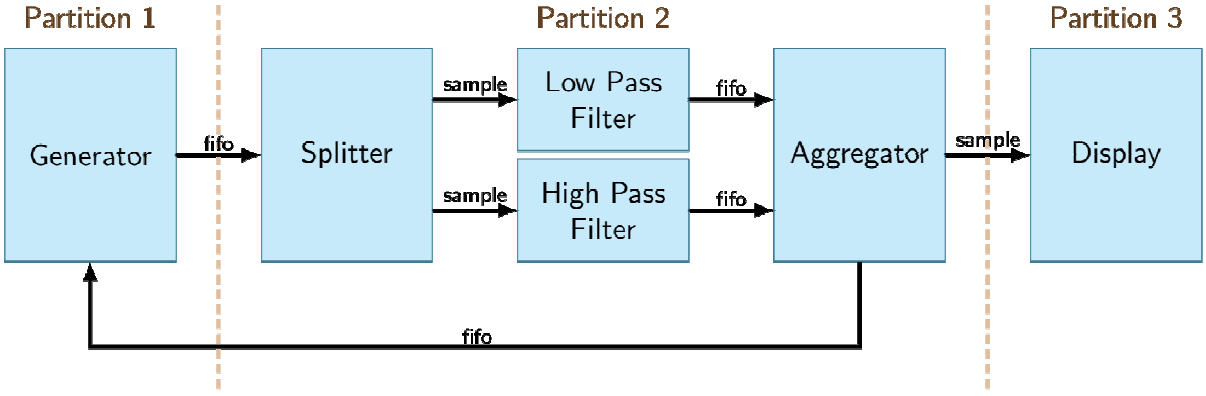

Figure 14: Partitioned BiQuad application implemented in PikeOS

Figure 14 illustrates the multi-partition PikeOS version of this application. It is composed of 3 different partitions and 6 PikeOS threads implementing the different tasks. This example exhibits the use of the possible communication mediums available in PikeOS, both for intra/inter-partition communication and for performing heavy load floating-point computation.

Several other implementations are also realized, including a bare metal version, and a single-partition version.

All the tasks composing the BiQuad application have soft **real-time requirements** expressed in term of data throughput. The quantity of data manipulated, the complexity of the computation, and the throughput requirements can all be parameterized in the application. We will select these requirements such that the high-critical application can be impacted by the generated interference without the RTE. The runtime engine will degrade the timing behaviour of the BiQuad application to ensure correct execution of the critical application.

5.3 Firm Real Time QOS-aware Application: Video broadcasting

The implementation of the last application composing the prototype has not started yet, but we already defined its specification. Contrary to the soft real-time application that is stopped if the resource budget is spent, the firm real time application reactively adapts to the available budget.

We selected a video broadcasting application based on the MJPEG format that allows us to vary the decoded frame rate accordingly to the resource available. The task graph of the application is presented in Figure 15.

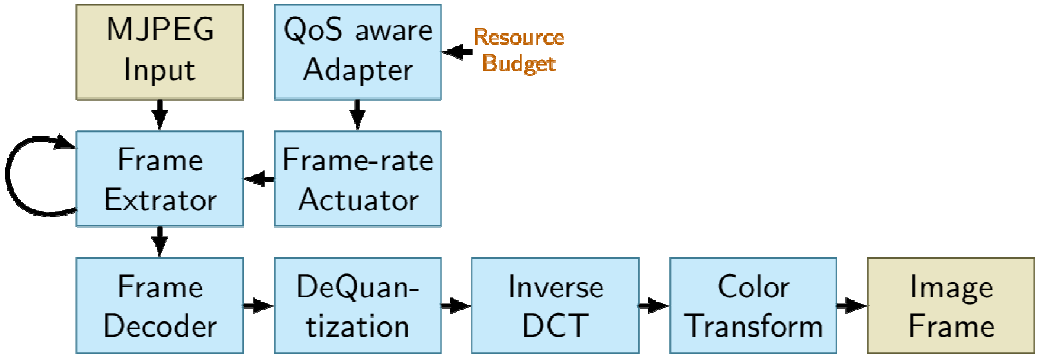

Figure 15: Firm Real Time Application Task Flow Graph

Before decoding each frame composing the video, this application will receive from the RTE the available resource budget, and decide if the current frame should be decoded or skipped. This way, rather than freezing the video because of more critical tasks, the framerate will be reduced, providing the user with a more acceptable experience.

As a baseline, this application will also be run without the QoS mechanism enabled, and we will compare the number of frame skipped with QoS versus the number of frame delayed and introduced latency without QoS (letting the RTE consider that the application is soft-real time).

The initial **timing requirements** of the MJPEG application correspond to decoding 25 frames per second, as per standard video broadcasting.

Chapter 6 Freedom from Interferences for mixed-critical systems

In automotive systems, Safety Analysis is based on the ISO 26262 standard [11]. The ISO 26262 provides design guidelines to accomplish at SW level “Freedom of Interference”, thus allowing to avoid that safety-relevant components and data could be corrupted by non-ASIL components.
In fact, the “Freedom of Interference” objective is to prevent propagation of a failure in one software partition to another software partition.
Automotive standard, like AUTOSAR, defines and designs safety mechanisms to guarantee “Freedom of Interference” (see chapter 6.3), these mechanisms are necessary on multicore system thought they affect system performances. Moreover, some of the issues addressed by ISO-26262, like "Freedom of interference”, can take a real advantage from the hardware separation (e.g. different private memories) provided by multi-core microcontrollers.
The SAFURE Automotive multicore use case (see D1.1) takes “Freedom of Interference” concept into account. In particular, Engine Control and Transmission Control are realized on the same device to reduce cost. The special safety requirements of the Engine and Transmission Control functionality must be covered by ensuring spatial and temporal isolation of the two parts. In order to achieve these requirements MAG, for SAFURE WP4, designed two optimized firmware drivers: TPROT and MPU (see chapter 6.3).

6.1 Safety and freedom from interferences mechanisms

In automotive systems, Safety Analysis is based on the ISO 26262 standard [11]. The standard consists of 9 normative parts and a guideline for the ISO 26262 as the 10th part. Here some extracts from ISO 26262 are provided, that describe different Safety Analysis area covered by ISO-26262, for details please refer to the full ISO 26262 documentation [11].

ISO-26262 provides design guidelines to accomplish at SW level “Freedom of Interference”, thus allowing to avoid that safety-relevant components and data could be corrupted by non-ASIL components.

Automotive Scenario related to SAFURE Project will implement protection mechanisms aligned to these ISO guidelines.

6.1.1 ISO 26262 – Timing Protection

ISO 26262-6, chapter 7 [11] covers software architectural design and describes how to analyse the dynamic design aspects (temporal constraints) of the software components. To determine the dynamic behaviour (e.g. of tasks, time slices and interrupts) the developer needs to consider:

- the different operating states (e.g. power up, shut down, normal operation, calibration and diagnosis).
- the communication relationships and their allocation to the system hardware (e.g. CPU and communication channels).

The determination of the dynamic behaviour of multicore systems is possible with a simulation approach. Temporal isolation is required to protect the execution time of critical tasks (software partitions) from unwanted interference caused by faults (including timing faults) such as blocking of execution, deadlocks, livelocks, incorrect allocation of execution time, and incorrect synchronization between software elements.

Complementing the runtime mechanisms for time isolation, we will also investigate fault-tolerant strategies and architectural approaches for fault isolation.

The normative regulation enumerates that these faults can be prevented by using traditional approaches like time triggered scheduling, cycling execution scheduling policy and fixed priority based scheduling. In this research project also dynamic scheduling approaches will be analysed.

Following annex D of ISO 26262-6 [11] freedom from interference by software partitioning has to be supervised by monitoring of processor execution time of software partitions in accordance with their allocation, program sequence monitoring and arrival rate monitoring.

6.1.2 ISO 26262 – Memory and exchange of information Protection

With respect to memory, the effects of faults such as those listed below can be considered for software elements executed in each software partition:

- corruption of content (it can cause a hardware reset),
- read or write access to memory allocated to another software element (it can cause an unpredictable behavior of the Control Unit).

Mechanisms such as **memory protection**, parity bits, error-correcting code (ECC), cyclic redundancy check (CRC), redundant storage, restricted access to memory, static analysis of memory accessing software and static allocation can be used to prevent faults.

With respect to the exchange of information, the causes for faults or effects of faults such as those listed below can be considered for each sender or each receiver:

- repetition of information,
- loss of information,
- delay of information,
- insertion of information,
- masquerade or incorrect addressing of information,
- incorrect sequence of information,
- corruption of information,
- asymmetric information sent from a sender to multiple receivers,
- information from a sender received by only a subset of the receivers, and
- blocking access to a communication channel.

6.2 AUTOSAR OS-Application and Protection Support

AUTOSAR specifications introduce OS-Applications as entities containers (TASKs, ISR2s, COUNTERs, ALARMS, Schedule Tables), that could resemble to processes (without memory virtualization).

The Operating System module is responsible for scheduling the available processing resource between the OS-Applications that share the processor. If OS-Application(s) are

used, all TASKs, ISRs, COUNTERs, ALARMs and Schedule tables must belong to an OS-Application. All objects which belong to the same OS-Application have access to each other. The right to access objects from other OS-Applications may be granted during configuration. An event is accessible if the TASK for which the EVENT can be set is accessible. Access means that these Operating System objects are allowed as parameters to API services.

The above are the fundamentals of Service Protection.

There are two classes of OS-Application:

1. Trusted OS-Applications: Are allowed to run with monitoring or protection features disabled at runtime. They may have unrestricted access to memory, the Operating System module's API, and NEED NOT have their timing behaviour enforced at runtime. They are allowed to run in privileged mode when supported by the processor.
2. Non-Trusted OS-Applications: Are not allowed to run with monitoring or protection features disabled at runtime. They have restricted access to memory, restricted access to the Operating System module's API and have their timing behaviour enforced at runtime. They are not allowed to run in privileged mode when supported by the processor.

Operating System module itself is a TRUSTED OS-Application.

The running OS-Application is defined as the OS-Application to which the currently running Task or ISR belongs. In case of a hook routine the Task or ISR which caused the call of the hook routine defines the running OS-Application.

There are services offered by the AUTOSAR OS which give the caller information about the access rights and the membership of objects and memory variables. These services are intended to be used in case of an inter-OS-Application call for checking access rights and arguments.

OS-Applications have a state which defines the scope of accessibility of its Operating System objects from other OS-Applications. Each OS-Application is always in one of the following states:

- Active and accessible (APPLICATION_ACCESSIBLE): Operating System objects may be accessed from other OS-Applications. This is the default state at start up.
- Currently in restart phase (APPLICATION_RESTART): Operating System objects cannot be accessed from other OS-Applications. State is valid until the OSApplication calls AllowAccess().
- Terminated and not accessible (APPLICATION_TERMINATED): Operating System objects cannot be accessed from other OS-Applications. State will not change.

Protection is only possible for Operating System managed objects. This means that:

- It is not possible to provide protection during runtime of OSEK Category 1 ISRs [12], because the operating system is not aware of any Category 1 ISRs being invoked. Therefore, if any protection is required, Category 1 ISRs have to be avoided. If Category 1 interrupts AND OS-Applications are used together then all Category 1 ISR must belong to a trusted OS-Application.

- It is not possible to provide protection between functions called from the body of the same Task/Category 2 ISR [12].

AUTOSAR specify four protection features:

- Service Protection: Protect the Object Access (TASK, ALARMs, Schedule Table, Resources) between OS-Application, if the permission is not explicitly granted.
- Memory Protection: Protect Global Data and Stacks of an OS-Application from possible corruption by Non-Trusted OS-Application.
- Stack Monitoring: On processors that do not provide any memory protection hardware it may still be necessary to provide a “best effort with available resources” scheme for detectable classes of memory faults. Stack monitoring will identify where a task or ISR has exceeded a specified stack usage at context switch time.
- Timing Protection: A timing fault in a real-time system occurs when a task or interrupt misses its deadline at runtime. AUTOSAR OS supply a mechanism based on budget that let the software understand which TASK or ISR2 is causing a deadline missing.

6.3 Freedom from Interferences OS-extension

For SAFURE, the Automotive Multicore Use Case prototype guarantees at firmware level the freedom from interferences compliant with ISO 26262. In particular, for SAFURE WP4 MAG has studied and developed two firmware drivers (AUTOSAR -like) to implement timing protection and memory protection to guarantee freedom from interferences between two applications that run on two different cores (chapter 6.2). This is an optimized alternative to RealTime OS supports.

6.3.1 Timing Isolation OS-extension

MAG has designed a FW component (TPROT) to realize Timing Protection without any support from a Real-time OS. From a Safety point of view, this allow to use Timing Protection (AUTOSAR -like) but with a QM RTOS (Part 3: Concept phase [11]).

AUTOSAR Timing Protection concept is built on the following monitors that will be adjusted in order to be used without any OS supports:

1. The execution time of Task/ISRs in the system

The measure of Task execution time can be performed without the need of OS, if this measure is triggered by the Task itself. The measure can be performed using a Timer Resource of the uC. Execution Budget violation can trigger Time Protection Error.

- Execution Budget: Maximum permitted execution time for a Task/ISR.

2. The blocking time that Task/ISRs suffers from lower priority Tasks/ISRs locking shared resources or disabling interrupts

The TPROT component can execute a measure of resources locked by lower priority tasks, if the measure is triggered by the lower priority task itself. Time violation can trigger Time Protection Error.

- Lock Time : Maximum permitted Interrupt Lock Time or Resource Lock Time

3. The inter-arrival rate of Task/ISRs in the system

TPROT will measure inter-arrival rate not in the strict meaning of Autosar (that will count each transition to READY STATE for each task and define a lower bound for it), but will just check that the time elapsed between two consecutive execution of the task is inside a determined upper bound. Again it is needed that the measure is triggered inside the Task.

- Time Period: The maximum time elapsed admitted between two successive activation of the same task.

6.3.1.1 TPROT Design Draft: Concept and API

TPROT introduce the concept of TimerSet that is a logical identifier bound to one or more uC timing resources. The number of uC resources needed may vary depending on the specific uC model. TimerSet will also encapsulate all the timing protection budgets and measures.

One task should always use the same TimerSet. Different Tasks should always use different TimerSet.

Some sample API:

TPROT_SetExecBudget (<TimerSet> Tx, <ExecutionBudget_uS> ExecBudget)

TPROT_SetTimePeriod(<TimerSet> Tx, <TimePeriod_uS> TimePeriod)

TPROT_SetLockBudget (<TimerSet> Tx , <LockBudget_uS> LockBudget)

TPROT_StartExecBudget (<TimerSet> Tx)

TPROT_StopExecBudget (<TimerSet> Tx)

TPROT_StartLockBudget (<TimerSet> Tx)

TPROT_StopLockBudget (<TimerSet> Tx)

TPROT_CheckExecBudget (<TimerSet> Tx) → Reset Execution Budget counting.

TPROT_CheckTimePeriod (<TimerSet> Tx) → Reset Time Period counting.

TPROT_CheckLockBudget (<TimerSet> Tx) → Reset Lock Budget counting.

TPROT_ClearTimerSet (<TimerSet> Tx)

TPROT_TimingProtection_Error (<ErrorId> E)

An example of usage for TPROT:

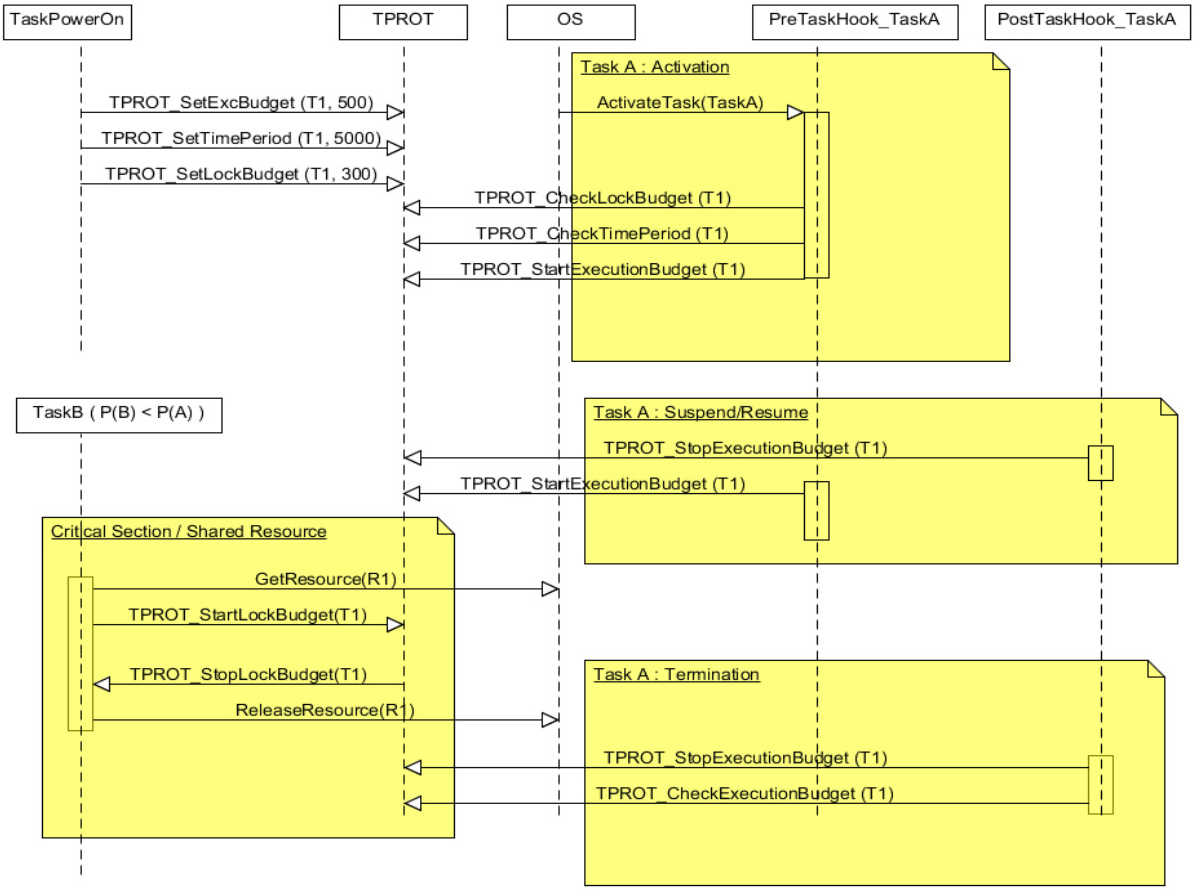

Figure 16: TPROT Design Draft (example explained with Sequence Diagram)

6.3.2 Memory Protection OS-extension

MAG has designed a FW driver to support memory protection at Task level, to guarantee that data structures inside the protected Task (or part of it) cannot be accessed by untrusted operations. This implementation makes use of microcontroller MPU device. This mechanism is ISO 26262 compliant and AUTOSAR-like.

Here is the Memory Protection concept described in AUTOSAR [13]:

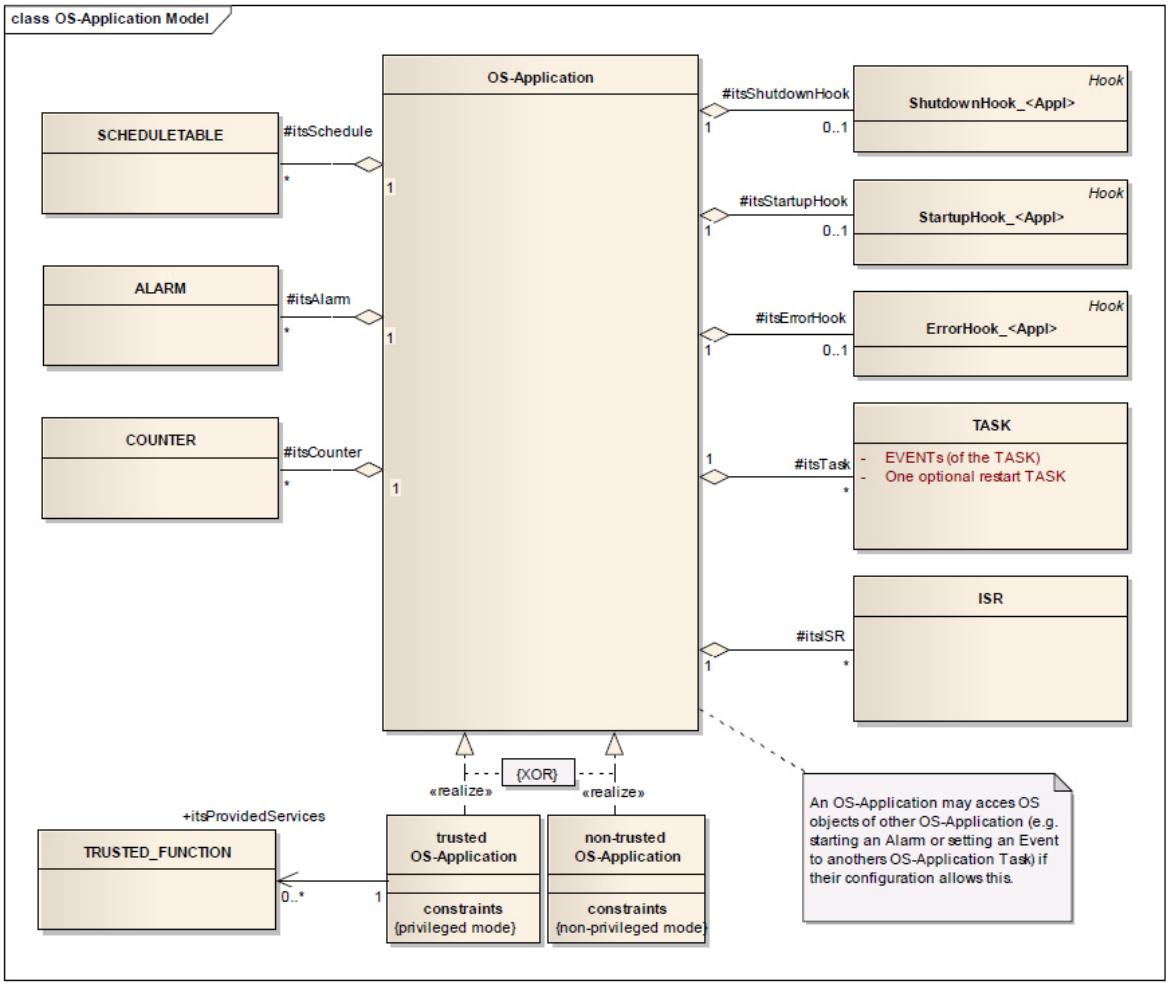

Figure 17: Memory protection applied at OS Application level.

6.3.2.1 Memory Protection Design Draft: Concept

Here is the concept implemented by MAG FW MPU Driver:

Suppose that the Figure 17 represents RAM partitioning (P0, P1 and P2) we want to obtain, giving the following requirements:

- Total of 256 KB of RAM.
- P0, P1, P2 are running on the same core.
- There is a real-time OS running. All the partitions P0, P1, P2 are preemptive.
- P2 is the default partition (see MPU Static Configuration section for details).

Figure 18: partitioning example.

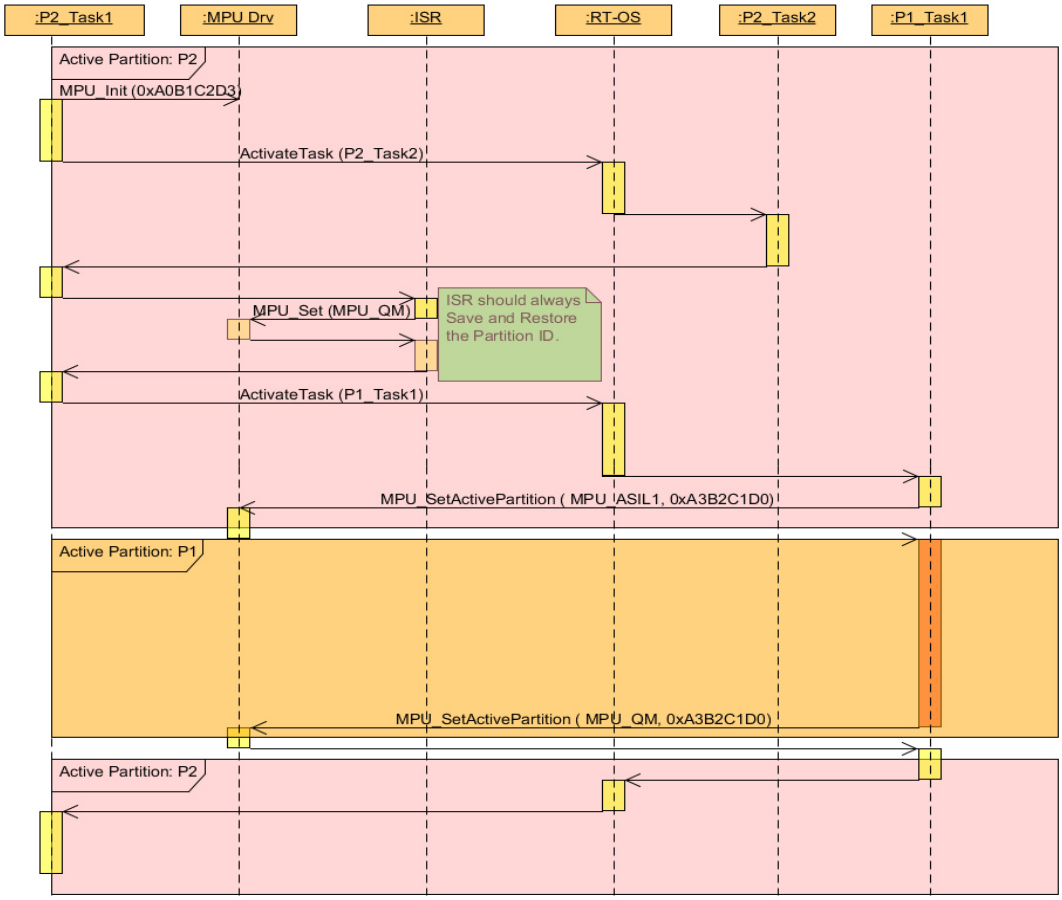

Figure 19: The diagram shows activation of P1 & P2 only, but the same apply when activating a task owned by P0.

Chapter 7 AUTOSAR OS

7.1 AUTOSAR OS support for mixed criticality

Currently, most of the AUTOSAR OS support for mixed critical application relies on a set of mechanisms (with reference to the 4.3 definition of the standard)

- mechanisms for timing isolation. Providing support for preventing timing faults from one application components from affecting other components
- mechanisms for monitoring. Providing support for checking violations of assumptions on the execution time of runnable, ISRs and critical sections
- mechanisms for predictable scheduling. Providing support for the implementation of scheduling policies that allow for predicatble interferences and blocking times.

Among these mechanisms, we are interested in those for which an open source implementation exists or can be obtained by extending the Erika OS [14] open source OSEK (AUTOSAR) system.

Hence, in the analysis of the services, we are especially interested in verifying their availability in Erika (for the architectures of interest for SAFURE). Of course other commercial implementation of AUTOSAR OS exist from several vendors in Europe and outside, but an open source kernel offers opportunities for better dissemination and experimentation by academics and the industry.

Mechanisms for timing protection

AUTOSAR OS guarantees a statically configured upper bound, called the Execution Budget, on the execution time of tasks and category 2 ISRs

AUTOSAR OS prevents guarantees a statically configured upper bound, called the Lock Budget, on the time that resources are held by tasks or Category 2 ISRs

AUTOSAR OS enforces an inter-arrival time protection to guarantee a statically configured lower bound, called the Time Frame, on the time between a task being permitted to transition into the READY state due to activation or release

Mechanisms for monitoring

AUTOSAR OS mechanisms for monitoring include hooks and events that are invoked or activated in correspondence to an attempt at policy violation.

Mechanisms for predictable scheduling

AUTOSAR OS mechanisms for predictable scheduling include scheduling tables and priority-based scheduling with ceiling mechanisms for local resource protection and locks for intercore resource protection (to be supplemented by a mechanism for global ceilings or an equivalent bounded blocking time mechanism). In their basic configuration both of them are not sufficient to provide isolation but must be supplemented.

7.2 AUTOSAR RTE generation for mixed-critical systems

The AUTOSAR RTE is the layer of automatically generated code that bridges the gap between the application SW components (and their code implementation) and the services of the basic SW, including the operating system, (most of) the drivers and the communication services.

The idea is that application developers should not use the OS (and scheduling) services directly, since before the composition of the system by the integrator, the runnable code is not assigned to any task or ECU and therefore cannot fully define its scheduling or scheduling conditions.

In mixed-critical systems the application SW components have additional characteristics that characterize them, including (time) criticality levels and the need for protection by isolation and a guarantee of their timing budgets or deadlines.

These requirements are expressed in the AUTOSAR component and runnable model, either directly, or by making use of the extensions developed in WP2.

Our planned development consists of the following:

A parser that analyses the ARXML code of a model with components a different criticality levels and build an internal representation of the AUTOSAR model with the proposed extensions.

The parser will be developed in Eclipse by leveraging the Artop framework [15] by the AUTOSAR consortium, or by defining a custom metamodel matching the AUTOSAR metamodel v4.3 and then using the Eclipse mechanisms for serialization and deserialization.

The second component is a code generator tool that takes the model of the application and automatically selects the OS services identified in the previous section for the protection of the timing characteristics of the critical tasks and for guaranteeing isolation.

The generator will be validated with a set of sample models, defined in Rhapsody.

7.3 Adaptive Autosar and future support for mixed-criticality

Adaptive policies for future support for mixed criticality include hierarchical scheduling support with possibly server policies on top of EDF.

Chapter 8 Summary and conclusion

This report is complementing the Alpha version of Operation System and Runt Time Engines prototype. We have reported on the current status and future work within work package four. The technologies developed for the final version of prototype will be implemented in the Telecom Use Case demonstrator in work package six.

Chapter 9 List of Abbreviations

BSP	Board Support Packages
EDF	Earliest Deadline First
TP0	Time Partition 0
FP	Fixed Priority
PSP	Platform Support Package
NDA	Non-disclosure Agreement
SEC	Security Engine
SFP	Security Fuse Processor
PBL	Pre-Boot Loader
RCW	reset configuration word
PBI	Pre-Boot Initialization
ISBC	Internal Secure Boot Code
ESBC	External Secure Boot Code
CST	Code Signing Tool
ITS	Intend To Secure
PKI	Public Key Infrastructure
SRK	Super Root Key
ECU	Engine control unit
PMC	performance monitoring counters
RTE	Run Time Engine
DTM	Dynamic Thermal Management
FMS	Flight Management System
BCP	Best Computed Position
QoS	Quality of Service
QM	Quality Management
RTOS	Real-time Operating System
ASIL	Automotive Safety Integrity Level
TPROT	Timing Protection FW Driver
MPU	Memory Protection Unit
ECC	Error-correcting Code
CRC	Cyclic Redundancy Check
ISR	Interrupt Service Routine
OSEC	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (German) English: "Open Systems and their Interfaces for the Electronics in Motor Vehicles"

Table 5: List of Abbreviations

Chapter 10 Bibliography

[1] ARM, «Fixed Virtual Platforms FVP Reference Guide,» [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.models.vplatforms/index.html.

[2] «https://en.wikipedia.org/wiki/Virtualization,» [Online].

[3] «www.openvirtualization.org/open-source-arm-trustzone.html,» [Online].

[4] «Earliest deadline first scheduling,» [Online]. Available: https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.

[5] SYSGO, PikeOS Fundamentals, PikeOS 4.0, Document Version 4.0-134.

[6] M. Kosinski e M. Krasikau, «PikeOS Training,» 2015.

[7] Freescale, «P4080 QorIQ Multicore Communication Processor,» 2014.

[8] Freescale, «HAB Code-Signing Tool User's Guide; Rev 2.2».

[9] Freescale, «Secure Boot For QorIQ Communication Processors».

[10] L. Schor, I. Bacivarov, H. Yang e L. Thiele, *Worst-Case Temperature Guarantees for Real-Time Applications on Multi-core Systems,* Real Time and Embedded Technology and Applications Symposium, 2012, pp. 87-96.

[11] International Organization for Standardization, *ISO 26262: Road Vehicles - Functional Safety,* 2011.

[12] «OSEK/VDX – Operative System,» [Online]. Available: www.osek-vdx.org.

[13] A. O. S. A. (AUTOSAR), «Specification of Operating System,» 2011.

[14] «Erika Enteriprise open source OS,» [Online]. Available: www.evidence.eu.com/products/erika-enterprise.html.

[15] «Artop - The AUTOSAR Tool Platform User Group,» [Online]. Available: www.artop.org.

[16] Cucinotta, T.; Checconi, F.;, «The IRMOS realtime scheduler,» 3 August 2010. [Online]. Available: http://lwn.net/Articles/398470/.