UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MATHEUS SCHUH

Porting of an avionics ARINC 653 compliant RTOS

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Trabalho realizado na École nationale supérieure d'informatique et de mathématiques appliquées de Grenoble (ENSIMAG) dentro do acordo de dupla-diplomação UFRGS – Grenoble INP.

Orientador:
Prof. Dr. Alexandre da Silva Carissimi (INF/UFRGS)

Co-Orientador:
Prof. Dr. Matthieu Moy (ENSIMAG/INP)

Porto Alegre
2016

CIP – Catalogação na Publicação

Schuh, Matheus
Porting of an avionics ARINC 653 compliant RTOS / Matheus Schuh. -- 2016.
72 f.

Orientador: Alexandre da Silva Carissimi.

Trabalho de conclusão de curso (Graduação) -- Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Curso de Engenharia de Computação, Porto Alegre, BR-RS, 2016.

1. Sistemas embarcados. 2. ARINC 653. 3. Arquiteturas paralelas. 4. Sistemas operacionais de tempo real. 5. Manycore. I. da Silva Carissimi, Alexandre, orient. II. Título.

Porte de um RTOS Aviônico baseado na norma ARINC 653

Matheus Schuh[1], **Alexandre Carissimi**[1], **Matthieu Moy**[2]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]École nationale supérieure d'informatique et mathématiques appliquées (ENSIMAG)
Institut polytechnique de Grenoble (Grenoble INP), France.

{mschuh,asc}@inf.ufrgs.br, Matthieu.Moy@grenoble-inp.fr

1. Introdução

Kalray é uma empresa pioneira na concepção de processadores *many-core*. Seu último lançamento, o Multi-Purpose Processor Array (MPPA®) Bostan conta com 288 núcleos programáveis em C/C++ com interfaces de Entrada/Saída (E/S) de alta velocidade. Essas características asseguram alto desempenho com baixo consumo (entre 10W e 20W) e resposta em tempo real.

Inúmeras aplicações podem se beneficiar de tais particularidades e, entre elas, estão os sistemas utilizados na indústria aviônica. O MPPA® oferece previsibilidade temporal, i.e. capacidade de calcular deterministicamente quanto tempo uma operação levará, e particionamento espacial, i.e. garantia de não-interferência entre espaços de memórias através de isolamento físico. Esses aspectos são cruciais para sistemas aeronáuticos.

Este resumo estendido irá apresentar o trabalho de porte de um Real-Time Operating System (RTOS) para a arquitetura do processador MPPA® Bostan. O documento está dividido em 6 seções, incluindo esta introdução. Na seção 2 é exposto o contexto e a motivação do projeto. Na seção 3 há uma análise de mercado, da metodologia de desenvolvimento de aplicações aeronáuticas e das soluções existentes em termos de sistemas operacionais (SOs). A seção 4 aborda os detalhes de implementação mais importantes e, por fim, na seção 5 os resultados do trabalho são mostrados seguidos pela conclusão na seção 6.

2. Contexto e Motivação

A Kalray foi fundada em 2008 por antigos funcionários da STMicroelectronics que possuíam experiência no desenvolvimento de arquiteturas paralelas e que acreditavam fortemente em seu potencial futuro. É uma empresa do tipo *fabless*[1], contando com uma equipe de hardware completa que realiza desde a concepção até a verificação de funcionalidade do processador.

Além do projeto de hardware, a empresa fornece também soluções de software visando integrar e explorar a capacidade de seu produto físico, entregando um pacote completo de acordo com as necessidade de cada cliente. Atualmente, os principais nichos de mercado atingidos são centros de processamento de dados (encriptação, compactação) e a indústria automobilística (processamento de imagem para veículos autônomos).

[1]Empresa que realiza o *design* de um circuito, externalizando sua produção.

A arquitetura do MPPA® Bostan é composta por 288 núcleos de 32, ou 64 bits, que recebem instruções do tipo *Very Long Instruction Word (VLIW)*. Desses núcleos, 256 são para uso geral e 32 para gestão de recursos. Os núcleos são ainda subdivididos em 16 clusters de cálculo com 2MB de *Shared MEMory (SMEM)* e 2 clusters de E/S com 1MB de SMEM e 2GB de *Double Data Rate (DDR)* para uso geral.

A comunicação interna no processador é realizada através de um *Network on Chip (NoC)* temporalmente previsível e eficiente. A comunicação externa é garantida através de interfaces de E/S dos tipos *Peripheral Component Interconnect (PCI)* e Ethernet. Uma visão geral da arquitetura do processador é apresentada na figura 1.

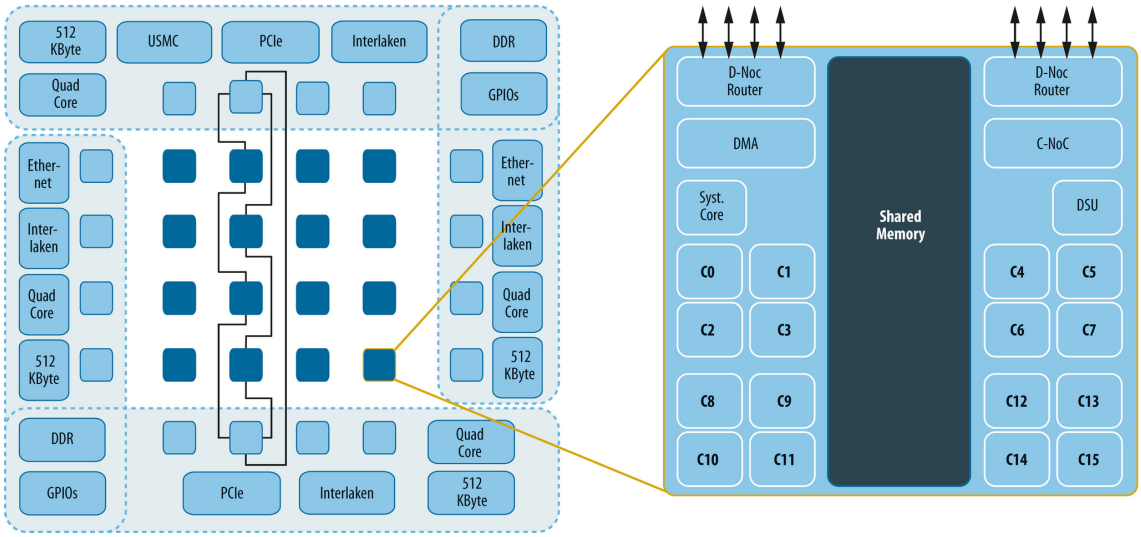


Figura 1. Visão geral do MPPA® Bostan com detalhe para um cluster de cálculo

O projeto, por estar inserido em um contexto aviônico, tem requisitos de um sistema crítico, onde falhas ou mal funcionamento podem representar mortes ou lesões e grande custo por perda de equipamento. Portanto, tanto o hardware, quanto o software, devem oferecer disponibilidade temporal, confiabilidade de execução, segurança contra falhas e proteção contra intrusões.

As características arquiteturais, previamente mencionadas, asseguram o MPPA® como um processador pronto para a computação crítica. Três aspectos são essenciais para isso:

- Computação determinística: duas operações realizadas sobre os mesmos dados de entrada terão a mesma sequência de instruções, executadas na mesma ordem.
- Tempos de resposta restritos: duas operações realizadas sobre os mesmos dados de entrada levarão o mesmo tempo.
- *Worst-Case Execution Time (WCET)*: capacidade de calcular o pior caso temporal de um sistema somando os piores casos individuais de seus componentes.

Além disso, elementos de hardware que tem efeito histórico, como a *cache*, podem ser desativados para evitar alterações temporais entre execuções. Ainda, a organização arquitetural do MPPA® em *clusters* permite a replicação de operações, disponibilizando redundância em caso de falhas.

Somando com o nível de criticalidade presente no hardware, a motivação principal do projeto vem da busca da indústria aeronáutica por um sistema que exponha uma camada de software com uma *Application Programming Interface (API)* padronizada, capacidade de particionamento temporal e espacial e que possa passar por um processo de certificação. Dessa maneira, o porte de um RTOS, que já atende os requisitos previamente citados, será feito para capacitar o MPPA® em termos de hardware e software para receber aplicações aviônicas que seguem a norma ARINC 653.

Na próxima seção explora-se em detalhe as exigências da camada de software revisitando como elas eram atingidas em sistemas aeronáuticos antigos e como deseja-se garanti-las atualmente.

3. Embasamento técnico

3.1. Metodologias de Desenvolvimento

Tradicionalmente a indústria aeronáutica utiliza uma metodologia denominada *Federated* para a concepção dos sistemas eletrônicos das aeronaves. Essa metodologia é baseada em unidades independentes e construídas especificamente para uma aplicação, e.g. controle do *display* da cabine de pilotagem. Tais unidades se comunicam através de um barramento que segue a norma *Avionics Application Standard Software Interface (ARINC)* 429 visando garantir a não-interferência e requisitos temporais do sistema.

Algumas desvantagens da metodologia *Federated* são: subutilização de recursos, visto que as unidades são especializadas e, possivelmente, carregam elementos que poderiam ser reaproveitados em outras partes do sistema; e o processo de certificação é realizado no sistema como um todo, sendo necessária a recertificação a cada troca ou adição de unidade.

Visando suprir tais deficiências a metodologia *Integrated Modular Avionics (IMA)* surgiu. Ela utiliza como base um SO onde partições de aplicação são executadas (equivalentes às unidades *Federated*). Uma partição é, portanto, um contêiner para uma aplicação, podendo abrigar múltiplos processos, porém, preservando a capacidade de isolamento temporal e espacial (em memória) entre os processos e outras partições. As aplicações aeronáuticas, e.g. controle do *display* da cabine, são assim construídas obtendo maior portabilidade e modularidade. A portabilidade vem do fato de que, como o SO é construído segundo uma norma específica, as aplicações desenvolvidas utilizando um sistema podem ser facilmente portadas para outro que segue a mesma norma. A modularidade vem do isolamento entre as partições

Essa modularidade permite que aplicações com diferentes níveis de criticidade sejam executadas em paralelo no mesmo sistema. Como consequência, todo o sistema pode ser implementado em um único *chip*, ao contrário de múltiplos sistemas eletrônicos integrados, existe uma grande economia em peso e em consumo energético. O processo de certificação, entretanto, é ligeiramente mais complicado, pois, o SO deve seguir estritamente as normas de concepção, bem como as aplicações. Uma vez que o longo processo de certificação, tanto para o sistema, quanto para as aplicações é concluído, os mesmo podem ser substituídos por outros sistemas e aplicações que também possuam certificação, sem necessidade de recertificação.

3.2. A norma ARINC 653

Com o desenvolvimento e a consolidação do conceito IMA, a necessidade de uma norma para estruturar o SO responsável por gerenciar todo o sistema era iminente. A norma ARINC 653 surgiu com o objetivo de preencher tal lacuna. Essa norma especifica basicamente dois aspectos do SO: sua interface e seus componentes.

A interface define como será a comunicação entre aplicações e o SO. Essa interface foi denominada de APplication/EXecutive (APEX), e é uma lista de funções para a gestão de partições, processos, tempo, memória e comunicação. Os componentes definem as estruturas básicas necessárias no SO para que ele seja capaz de fornecer a interface APEX, entre elas um escalonador customizável, possibilidade de comunicação intra e inter partições e gestão de recursos de hardware como memória e frequência.

A interface APEX é resumida e agrupada nos chamados grupos de serviço ARINC. Funções semelhantes compõem o mesmo grupo e essa organização auxilia tanto na implementação quanto nos testes de um SO baseado na norma ARINC 653.

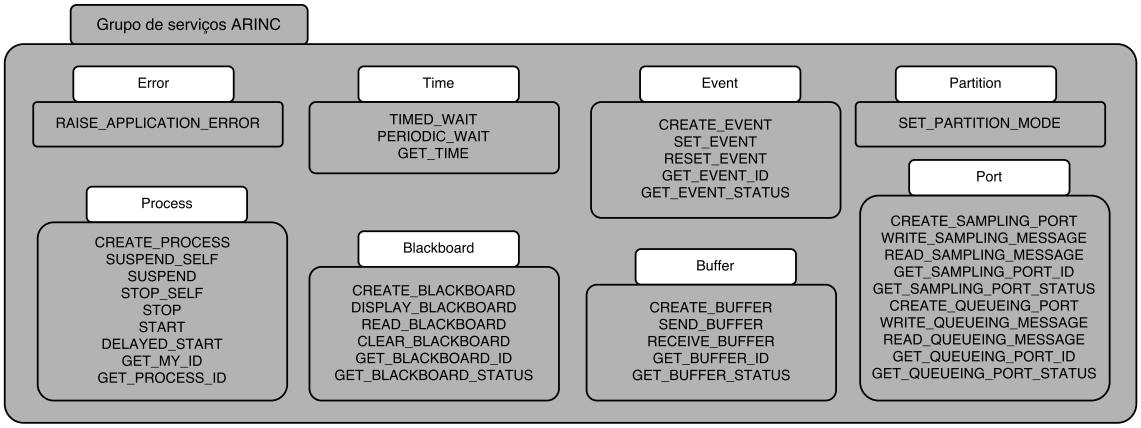


Figura 2. Grupo de serviços ARINC e suas funções

3.3. TiCOS

Uma análise dos SOs disponíveis no mercado foi feita para a definição de qual deles seria de fato portado para o processador MPPA®. Os aspectos mais relevantes foram ter código aberto, haver possibilidade de contato com os desenvolvedores e respeito estrito da norma ARINC 653.

O Time-Composable Real-Time Operating System (TiCOS) foi escolhido por satisfazer os aspectos citados anteriormente. Além disso, o TiCOS tem boa documentação, está disponível publicamente e possui colaboradores ativos. O TiCOS foi resultado da tese de doutorado de Andrea Baldovin e baseado em outro sistema chamado de POK. Sua implementação original é voltada para a arquitetura PowerPC da IBM que é *single-core*. Além disso, a adoção do TiCOS também foi uma recomendação de parceiros externos da Kalray neste projeto.

4. Implementação

4.1. Objetivo

Como citado, o TiCOS tem sua implementação original voltada para a arquitetura *single-core* PowerPC da IBM. O código é composto principalmente por `C` e *assembly* específico

da Instruction Set Architecture (ISA) PowerPC.

O objetivo deste trabalho é portar o TiCOS para a arquitetura MPPA® da Kalray, preservando o aspecto *single-core* do núcleo (*kernel*) do SO em um primeiro momento. A linguagem `C` para o código foi mantida graças ao suporte prévio do ambiente de software do processador. A maior mudança vem da alteração do código *assembly*, específico para a arquitetura MPPA®, e o uso de uma camada de virtualização que será apresentada nas próximas seções, o `mOS`.

4.2. Arquitetura

A arquitetura do TiCOS é dividida em duas camadas: aplicação e núcleo. A camada de aplicação é formada por um componente *core* que expõe ao usuário as funcionalidades do sistema e se apoia nas bibliotecas ARINC (contém as interfaces da APEX segundo a norma), *middleware* (responsável por tratar os serviços de comunicação) e *core* (responsável por tratar os demais serviços, e.g. criação de *threads*, controle de tempo). A comunicação entre a camada de aplicação (que é executada em nível de usuário) e o núcleo (executado originalmente em modo privilegiado) é feita através de chamadas de sistema.

A camada do núcleo é formada também por um componente *core* que é o ponto de entrada das chamadas de sistema e se comunica com o componente dependente da arquitetura (na implementação original denominado PowerPC *arch*). Esse último componente é o responsável pelo acesso aos recursos de hardware através de instruções específicas do processador.

O porte para uma nova arquitetura deve, em teoria, modificar apenas o último componente mencionado. Dessa maneira, um componente MPPA® *arch* foi adicionado e, entre a camada do núcleo e o hardware, foi introduzida a camada de virtualização composta pelo `mOS`. A figura 3 mostra a visão global da arquitetura do TiCOS.

4.3. O Hipervisor `mOS`

A Kalray optou por encapsular todo seu ambiente de software com um hipervisor que nada mais é do que uma camada virtual entre o hardware e qualquer aplicação, até mesmo um SO de terceiros.

O hipervisor `mOS` é um exonúcleo, um tipo específico de núcleo de SO extremamente minimalista e pequeno. Esse conceito foi introduzido pelo grupo de sistemas operacionais paralelos e distribuídos do MIT, em 1994. Uma de suas principais características é não forçar abstrações de hardware para as próximas camadas de software, construindo um *kernel* com poucas funcionalidades mas que entrega maior flexibilidade às aplicações. Um exonúcleo deve primordialmente garantir proteção, através da virtualização, e multiplexação de recursos de hardware.

Qualquer aplicação, ou SO, que utiliza um hipervisor é, portanto, considerada uma biblioteca, ou um SO hóspede, já que são executados em nível de usuário. Algumas características particulares do `mOS` da Kalray são: tamanho reduzido (apenas 32KB), presença de uma estrutura, denominada *scoreboard*, que armazena o estado virtual do sistema e uso de *hypercalls* que são chamadas de sistema com pré e pós processamento de dados.

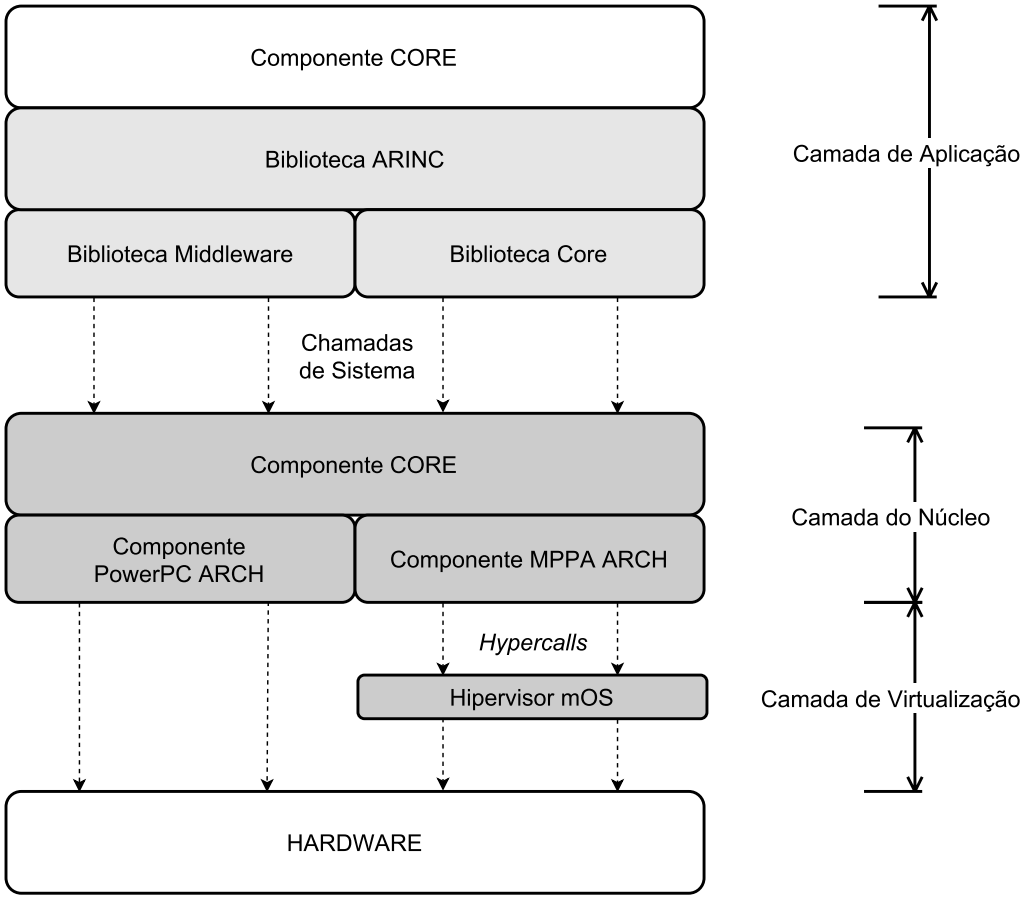


Figura 3. Visão arquitetural do TiCOS com o componente MPPA®

4.4. Metodologia de Desenvolvimento

Para este projeto a mesma metodologia de implementação foi utilizada para cada nova funcionalidade prevista pelo sistema original. Ela é descrita abaixo:

1. Análise do código do TiCOS e das estruturas presentes no processador PowerPC;
2. Estudo de estruturas equivalentes no processador MPPA®;
3. Identificação de funções existentes no hipervisor `mOS` que explorem essas estruturas;
4. Utilização da API do `mOS` ou implementação da funcionalidade.

As próximas seções apresentam detalhes de implementação e das estruturas de suporte de hardware presentes no processador, mostrando os principais pontos de interesse em um trabalho de porte de um SO.

4.5. Suporte de Arquitetura

O TiCOS necessita de alguns recursos de hardware comuns e presentes na maioria dos processadores para seu funcionamento. O primeiro deles, usado no isolamento temporal entre partições, é um registrador para o controle do tempo global do sistema juntamente com um registrador decrementador. O isolamento temporal garante que apenas uma partição estará em execução no sistema em um determinado *slot* de tempo. O

MPPA® possui uma estrutura no *scoreboard* que mantém um *timestamp* global e virtualiza o registrador físico decrementador de 64 bits. A API do `mOS` fornece acesso a ambos.

O segundo recurso é a definição de um contexto do estado arquitetural dos registradores para seu armazenamento e restauração. O MPPA® possui 64 registradores para uso geral e 50 registradores para funções específicas. A definição dos registradores importantes para o contexto foi feita em uma `struct` em `C` e macros *assembly* são utilizadas para salvar e restaurar o contexto.

O terceiro recurso é a gestão de memória através de uma *Memory Management Unit (MMU)* para o particionamento espacial. Embora o MPPA® possua tal suporte em hardware, ele é limitado, seu uso é complexo e desencorajado pelos desenvolvedores. A solução arquitetada foi separar a compilação do núcleo e das partições em endereços fixos, i.e. código absoluto, sem amarração dinâmica com a MMU. O ligador então se encarrega de empacotá-los em um objeto único (multibinário).

4.6. Casos especiais

Além das estruturas de hardware citadas, o TiCOS requer suporte extra de software em alguns casos especiais. Quanto ao primeiro dos casos, o sistema original foi concebido para ser *standalone*, i.e. não depender de nenhuma biblioteca externa para sua compilação. Assim sendo, o processo de compilação do sistema é feito usando a opção `--nodefaultlibs` do `gcc`. Para suprir a ausência de algumas funcionalidades básicas foi necessária uma implementação minimal da `libC`, bem como sua compilação e posterior ligação com o restante do sistema. Funções como `putchar` e `exit` foram escritas usando chamadas de sistema próprias do MPPA®. Rotinas aritméticas de baixo nível como `__divdf3`, `__modis3`, `__umodis3`, `divmodsi4` foram reaproveitadas de outros módulos já presentes no ambiente de software da Kalray.

O segundo caso vem da atrelação entre as chamadas de sistema e a ISA em questão, necessitando de um trabalho de porte tanto na instrução *assembly* utilizada, quanto no tratador das chamadas. É um trabalho vital para o funcionamento do sistema, visto que os serviços ARINC que necessitam acessar funcionalidades do núcleo do sistema terminam em chamadas de sistema.

Na ISA do MPPA® a instrução *assembly* é a `scall`. Ao ser executada, ela é interceptada pelo `mOS`, que redireciona o fluxo da aplicação para o tratador de chamadas de sistema. O tratador é responsável por salvar o contexto de execução, trocar as pilhas entre partição e núcleo e chamar as próximas funções do sistema que irão completar o fluxo de execução. O registro do tratador de chamadas de sistema é feito no *scoreboard* do `mOS` durante a inicialização do sistema.

O terceiro caso trata do suporte aos dois tipos de troca de contexto presentes no sistema original: dirigidas por interrupção e explícitas. Para isso, além de um registrador decrementador e de *timestamp*, é requerido o conhecimento do sistema de interrupções da arquitetura e o registro de um tratador, semelhante ao de chamadas de sistema.

A troca de contexto dirigida por interrupção ocorre no momento em que o registrador decrementador atinge zero e levanta uma interrupção, que interrompe o fluxo normal de execução do processador. Essa interrupção é então interceptada pelo `mOS` e redirecionada para o tratador que se encarregará da troca de contexto. Esse é um tipo

de escalonamento preemptivo de partições. O registro de tal tratador também é feito no scoreboard do `mOS` durante a inicialização do sistema.

A troca de contexto explícita ocorre quando uma partição decide esperar pelo próximo período de execução ou por um evento. A partição faz uma chamada de sistema que invoca o escalonador, que por sua vez retira a atual partição da execução. Esse é um tipo de escalonamento cooperativo, porém que ainda respeita o aspecto tempo real, visto que a próxima partição só será executada quando o *slot* de tempo reservada a ela começar.

5. Resultados obtidos

Antes de apresentar os resultados é importante mencionar o processo de validação utilizado e as dificuldades encontradas. Idealmente, ao desenvolver qualquer software visa-se isolar suas camadas e funções para posteriormente criar testes unitários. Mesmo com a arquitetura modular do TiCOS (seção 4.2), a utilização de testes unitários não foi possível, pela vinculação entre as camadas de núcleo e de partições de aplicação em único objeto.

Resta então como vetor de teste aplicações que serão compiladas e executadas juntamente com o TiCOS. As aplicações de teste tem duas origens: códigos exemplo fornecidos juntamente com o sistema original e códigos obtidos através de um gerador automático que tem como entrada arquivos `XML`.

Os códigos exemplo possuíam os estímulos necessários para avaliar a maioria dos grupos de serviço ARINC. Não estavam incluídos testes para os grupos EVENT, BLACKBOARD, BUFFER e PORT. Ainda, o sistema não estava sendo testado em termos de escalabilidade, isto é, o exemplo que mais estressava o TiCOS em gestão temporal e espacial possuía 3 partições com 1 processo cada.

Assim, um gerador automático de código foi usado para complementar os casos de teste. Foram criadas aplicações para estimular os serviços EVENT e PORT e aplicações com 4 partições e 1 ou 2 processos cada. O gerador foi necessário, pois o código do núcleo e das partições é parametrizável através de arquivos do tipo `deployment.h` e `deployment.c` que possuem diretivas de compilação e são de difícil alteração manual. Esses arquivos, além de *templates* para o código fonte das aplicações, são automaticamente criados pelo gerador com arquivos `xml` como entrada, que descrevem de maneira declarativa as partições, processos e comportamento esperado de uma aplicação.

Os critérios de avaliação para a execução dos testes foram:

- Funcionais, i.e. se a execução da aplicação está em conformidade com seu código fonte. Para isso uma análise visual baseada em mensagens de *debug* foi utilizada;
- Estruturais, i.e. estado arquitetural dos registradores durante a execução de serviços ARINC que podem modificá-los;
- Temporais, i.e. estado dos registradores de *timestamp* e de decremento em momentos chave da execução, como a troca de contexto.

Cada uma das aplicações teste explorava um ou mais conjunto de serviços ARINC fornecido pelo sistema, visando-o testar na sua integridade. A execução foi feita primeiramente em simulação *Instruction Set Simulator (ISS)*, que é mais rápida e facilita testes com funções de E/S. Posteriormente a execução foi feita em hardware, buscando avaliar os registradores físicos e aspectos temporais através do uso do `gdb`, o que não está com-

pletamente modelado na simulação. Após ou durante a execução os critérios de avaliação foram então aplicados para validar ou reprovar o teste.

Os resultados dos testes são apresentados na tabela 1. As células da tabela que apresentam o símbolo ✓evidenciam que o grupo de serviço ARINC correspondente foi validado através de uma aplicação executada com o TiCOS, em simulação, ou em hardware (dependendo da coluna), segundo os critérios previamente citados. As células que apresentam o símbolo ✗ evidenciam que o grupo de serviço não foi testado. Em especial os grupos *blackboard* e *buffer*, que fornecem comunicação intra partição, não foram testados em nenhuma das aplicações executadas.

Tabela 1. Resultados da validação em simulação e hardware

Grupo de serviço ARINC	Simulação	Hardware
ERROR	✓	✓
TIME	✓	✓
PARTITION	✓	✓
PROCESS	✓	✓
EVENT	✓	✓
BLACKBOARD	✗	✗
BUFFER	✗	✗
PORT	✓	✓

O produto final é o RTOS TiCOS portado com sucesso para a arquitetura MPPA®, com exceção de dois grupos de serviço que não possuíam alta prioridade no projeto segundo a especificação da Kalray. Além disso melhorias no processo de compilação foram feitas, para que todos exemplos presentes no diretório do sistema pudessem ser compilados de uma só vez e posteriormente executados através de comandos simples no MPPA®.

6. Conclusão

Os resultados da validação apontam que o porte do TiCOS foi feito com sucesso para o MPPA®. Esse trabalho foi de extrema importância na construção de uma prova de conceito da arquitetura para aplicações aviônicas e ressaltou suas características de criticidade e tempo real.

O projeto de porte de um SO é um ótimo tópico, pois incentiva a análise de código de terceiros, requer profundo conhecimento em `C` e *assembly* e familiaridade com a arquitetura para a qual o sistema está sendo portado. O uso do `mOS` como hipervisor trouxe acessibilidade aos recursos de hardware de maneira fácil, acelerando a curva de aprendizado e a realização de algumas tarefas. Entretanto, sua obrigatoriedade acarreta na perda da flexibilidade presente em sistemas *bare-metal*.

Como já dito anteriormente, o porte teve um aspecto *single-core* onde o núcleo e as partições são executados no mesmo *core*, isolados temporalmente e dispostos na SMEM em endereços fixos. Entretanto, um isolamento espacial garantido por estruturas de hardware é necessário para um processo de certificação. Trabalhos futuros podem explorar a MMU do MPPA® para obter tal característica.

No intuito de explorar a capacidade de processamento do MPPA® um porte a nível de *cluster* pode ser realizado, ainda em fase de estudo. A primeira proposta foi do

tipo Asymmetric multiprocessing (AMP), com o núcleo sendo executado em um *core* e as partições aplicações em *cores* distintos do mesmo *cluster*. Um porte a nível do MPPA® completo também está em estudo. Nesse caso, o núcleo seria executado em um *cluster* de E/S e as aplicações em cores distintos de *clusters* distintos, sendo o NoC responsável pela comunicação entre partições.

Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Final Study Project Report

Undertaken at **Kalray**

Porting of an avionics ARINC653 compliant RTOS

Matheus SCHUH
3A – SLE

22 February 2016 – 22 July 2016

Kalray SA
445 Rue Lavoisier
38330 Montbonnot Saint Martin

Internship responsible
Benoît de DINECHIN
School Tutor
Matthieu MOY

Acknowledgements

*I would like first to thank **Kalray** for offering me the opportunity to undertake my internship at the Montbonnot site, for welcoming me in its facilities and for giving me all the required material to accomplish my tasks.*

*I thank particularly **Benoît de Dinechin** for offering me this internship, for trusting me in the accomplishment of the given tasks and for supporting me in fullfilling them.*

*I am also grateful to **Matthieu Moy** for accepting to be the tutor of this project, for the advices given and the revisions of the report.*

*I also thank **Pierre Guironnet de Massas** and **Samuel Jones** for their constant support in all OS, MPPA® architecture and general doubts that came up throughout the internship.*

*Finally, I would like to thank all the **Core Software Team members** for receiving me during these 5 months, for helping me in doubts and difficulties and also for their kindness and patience with eventual french communication problems.*

Abstract

The avionics industry has a huge interest in improving their systems through better performance and lower power consumption while maintaining the safety requirements. Kalray's processor comes in handy and is able to meet these needs with a proper software environment.

The work presented in this document is focused on porting a Real-Time Operating System (RTOS) to Kalray's processor. This OS provides resource management and a programming interface that respects the ARINC 653 avionics standard.

The port was successfully accomplished as the majority of the examples are functional on simulation and on hardware. Even though, it is still a work in progress as some ARINC services are not completely working and optimizations exploiting the processor architecture can be made.

Keywords: Kalray, MPPA, ARINC653, embedded systems, C, assembly, processor, many-core, APEX, RTOS, TiCOS, POK.

Résumé

L'industrie avionique porte un grand intérêt à l'amélioration de leurs systèmes en ce qui concerne la performance et la consommation énérgétique, tout en respect les contraintes de sûreté. Le processeur Kalray permet de répondre à ces besoins par son environnement logiciel standard.

Ce document présente le travail de portage d'un système d'exploitation temps-réel (RTOS) au processeur de Kalray. Cet OS assure la gestion des ressources et une interface de programmation qui respecte la norme avionique ARINC 653.

Le portage a été accompli avec succès, vu que la plupart des exemples sont fonctionnels en simulation et en matériel. De toute façon, cela est encore un travail en progrès, car certains services ARINC ne marchent pas et des optimisations qui exploitent l'architecture du processeur peuvent toujours être faites.

Mot-clés: Kalray, MPPA, ARINC653, systèmes embarqués, C, assembleur, processeur, many-core, APEX, RTOS, TiCOS, POK.

Contents

List of Figures

List of source codes

Glossary

ALU Arithmetic and Logic Unit
APEX APplication/EXecutive
API Application Programming Interface
ARINC Avionics Application Standard Software Interface
ASIC Application Specific Integrated Circuit

BCU Branch and Control Unit
BMP Bound Multi-Processing
BSP Board Support Package

C-NoC Control NoC
CC Compute Cluster
CPU Central Processing Unit

D-NoC Data NoC
DDR Double Data Rate
DSU Debug System Unit

ELF Executable and Linking Format

FIFO First in, First out
FPGA Field Programmable Gate Array
FPU Floating-Point Unit

GNU Recursive acronym for GNU's Not Unix
GPR General Purpose Register
GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

I/O Input/Output
IMA Integrated Modular Avionics
IOC I/O Cluster
ISA Instruction Set Architecture
ISS Instruction Set Simulator

JTAG Join Action Group

LRU Least Recently Used
LSU Load/Store Unit

MAU Multiply-Accumulate Unit
MMU Memory Management Unit
MPPA® Multi-Purpose Processor Array

NoC Network on Chip

OS Operating System

PCI Peripheral Component Interconnect
PE Processing Element

RM Resource Manager
RR Round-Robin
RTC Real Time Controller
RTOS Real-Time Operating System

SFR Special Function Register
SMEM Shared MEMory

TiCOS Time Composable Operating System

VHDL VHSIC Hardware Description language
VHSIC Very High Speed Integrated Circuit
VLIW Very Long Instruction Word

WCET Worst-Case Execution Time

1 Introduction

Kalray is a pioneer company in the development of many-core processor architectures. Its latest processor release, the MPPA®2-256 Bostan, with 288 C/C++ programmable cores, provides a performance level similar to ASICs and high speed Input/Output (I/O) interfaces. These characteristics allow the so-called supercomputing on a chip with low power consumption and real-time response.

The applications that can benefit from such an architecture are the most varied: data centers, encryption acceleration and multimedia processing are some examples. Besides from these parallel intensive tasks, the avionics industry is particularly interested in the time predictability and space partitioning provided by the processor. The goal of the internship is to use the aforementioned features in order to port a RTOS compliant with the ARINC 653 specification to the Kalray's MPPA®. Thanks to this Operating System (OS) interface, clients may port modular avionics software to the MPPA® processor.

This document will begin by exposing an overview of the company and the work team. Then the internship's subject will be presented in detail: final goal, problematic, restrictions and work to be done. Furthermore, a technical section will follow containing the possible analysed solutions, a description of the alternative chosen and an evaluation mechanism of the developed product. Implementation and results sections are placed next, showing the work accomplished until this moment. Finally, a planning with the progress, a personal appraisal and a conclusion will be attached.

2 Context

2.1 The company

Kalray is an organization created in 2008 by Joël Monnier, former vice president of *STMicroelectronics*. It is specialized in the design of many-core processors targeting mainly the embedded computing market (a photo of such processor can be seen in figure 1). As many semiconductor companies nowadays, it is identified as fabless[1], having its processors production done in Taiwan.

More than 50 people are currently working in Kalray's offices. They are spread throughout Paris (financial headquarters), Montbonnot-Saint-Martin (near Grenoble), Tokyo (Japan) and recently Los Altos (USA) in the *Silicon Valley*.

Figure 1: Two Kalray MPPA®-256 processors in a board

2.2 The site

The Kalray Montbonnot-Saint-Martin site groups the most part of the company's workforce. It is situated in a region called Inovallée[2] that incubates more than 380 companies and laboratories. This privileged location has allowed multiple collaborative projects and the creation of a joint laboratory with the CEA, employing more than 30 engineers.

There are two main teams in the site:

- **Hardware:** the logical and physical design of the processor is done by this team, which is subdivided in:
  - Backend: in charge of the placement and routing process, in order to obtain a functional and optimized circuit layout. This design will finally be printed out on masks[3].
  - Frontend: responsible for the architecture design and the hardware description of the processor, written in VHSIC Hardware Description language (VHDL). A part of this team implements the VHDL code in a Field Programmable Gate Array (FPGA), providing a quick prototype interface to be used in internal projects.

[1] Company that externalizes the production process, being responsible only for a product's conception.

[2] A science park between Meylan and Montbonnot-Saint-Martin created in 1971 with the purpose of shorting the distances between industry and laboratories. Similar in concept with the *Silicon Valley*.

[3] A plate with transparencies that allow the light to pass in a defined pattern. In a chip production this is used in the photolitography process, engraving a layout on a silicon wafer.

- **Core Software:** all the software tools used to program the MPPA® are done by this team, which has two main work branches:
  - Core Development: creation of Kalray versions of GNU tools, such as `k1-gcc`, `k1-gdb` and `k1-objdump`. Deployment tools such as a Instruction Set Simulator (ISS)[4] and a hardware runner[5] are also developed and maintained by this branch.
  - Core Applications: creation of programs that demonstrate the processor abilities and performance. OS and libraries (such as the well known *libc*) are developed and ported by this branch.

Even though Kalray has been in the market for 8 years, the startup atmosphere remains. The communication and interaction between teams are extremely easy and the work dynamics is flexible. That means one person is not tied down to a group, being allowed to work in multiple projects at the same time.

2.3 The team

The Core Software team is lead by Céline Barraud and sums up to 20 engineers working, as already mentioned, with all the problematics linked to the MPPA® software environment.

My internship tutor is Benoît Dupont de Dinechin, who occupies the post of Chief Technology Officer, having more than 20 years of expertise in software engineering, processor design and compiler tools. As he is frequently busy with managing decisions, daily basic questions are addressed to Pierre Guironnet de Massas who possesses a deep knowledge of the Kalray environment and the process of porting an OS, specifically its kernel.

Both of these mentors work inside this team and the internship follow-up is done through a weekly email containing the tasks accomplished in the period and the goals to attain for the next week. This helps them verify the quality of the work and if a specific task may be blocking the progression.

[4] A simulation model that imitates a processor by reading instructions, executing them and maintaining a virtual state of such hardware.

[5] A tool that runs a given binary in a hardware system, in our case the MPPA® with its `k1-jtag-runner`.

3 The internship project

3.1 The MPPA processor

MPPA®2-256 Bostan is Kalray's latest processor. It implements a 32-bit/64-bit Very Long Instruction Word (VLIW)[6] core named `k1b`. The frequency of operation can vary between 4000MHz and 600MHz, which gives a power consumption variation between 10W and 20W. Its peak floating-point performances are 634 GFLOPS and 316 GFLOPS for single and double precision respectively.

In the following sub-subsections some important aspects of the processor will be detailed, as they are required for further comprehension of the document.

3.1.1 Global Architecture

Inside the processor there are 288 cores divided in 16 *Compute Clusters* and 2 *I/O Clusters (IOCs)* (an overview can be seen in figure 2).

Each CC is composed of 16 Processing Element (PE) cores and 1 Resource Manager (RM) core, all sharing a local Shared MEMory (SMEM) of 2 MB. The communication between CCs is achieved using the Network on Chip (NoC).

Each IOC is composed of 8 RM cores with multiple I/O interfaces (PCIe, Ethernet) and an external Double Data Rate (DDR) memory that normally can be accessed only by these cores.

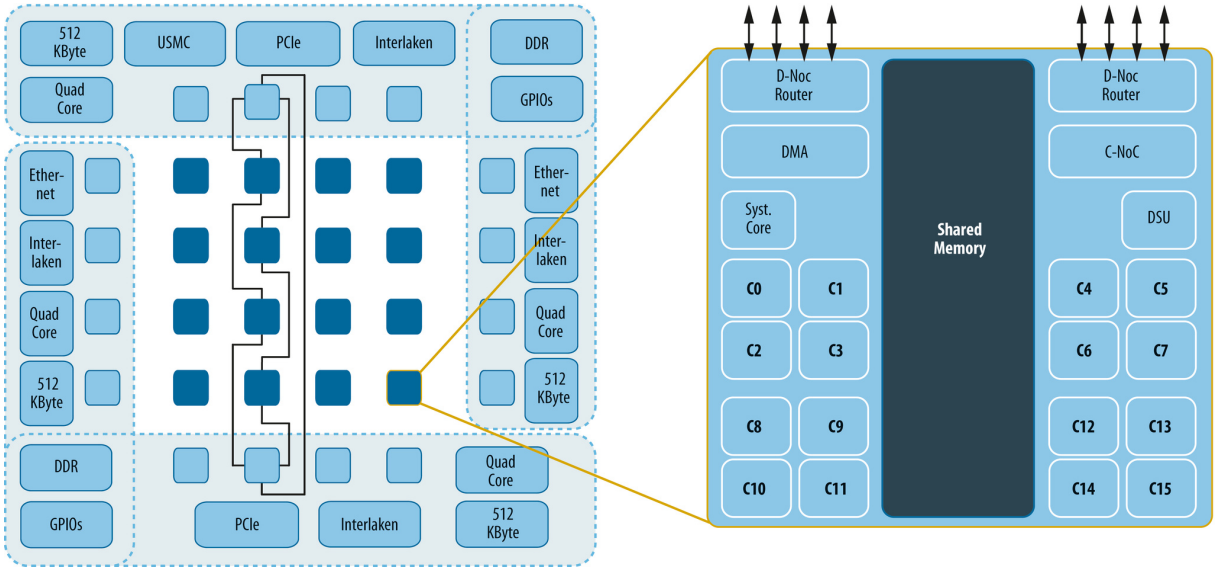


Figure 2: MPPA®2-256 Bostan Overview with a CC zoomed view

The processor was conceived originally to perform parallel calculation (for image and video processing) with high performance and low power consumption, overcoming in these terms a Graphics Processing Unit (GPU). Throughout the years, many other possibilities have appeared: cryptography, data storage, self-driving cars, real-time systems, etc.

Isolating the DDR access and limiting the CC communication to the NoC interface are important features for these applications. It makes the MPPA® more energy-efficient and

[6] Processor architecture capable of executing multiple instructions at the same time as long as they are not dependant.

prevents unexpected timing interferences between the clusters. Another great advantage is that the programming environment remains a C/C++ standard with no proprietary language learning requirements.

NoC The MPPA® contains a dual NoC that allows communication and synchronization between the clusters. It is composed of a Data NoC (D-NoC) designed for heavier data traffic, capable of remote writing and a Control NoC (C-NoC) which supports fast synchronization barriers.

The topology is based on a 2D torus[7] augmented with direct links between IOC nodes and NoC extension links to other MPPA® processors or an external FPGA. The two NoCs are identical concerning the bi-directional links, router arbitration, the topology and the route encoding, differing at their interfaces and by the size of First in, First out (FIFO) queues. The MPPA® NoC totalizes 32 nodes, one per CC and 8 per IOC (c.f. figure 3).

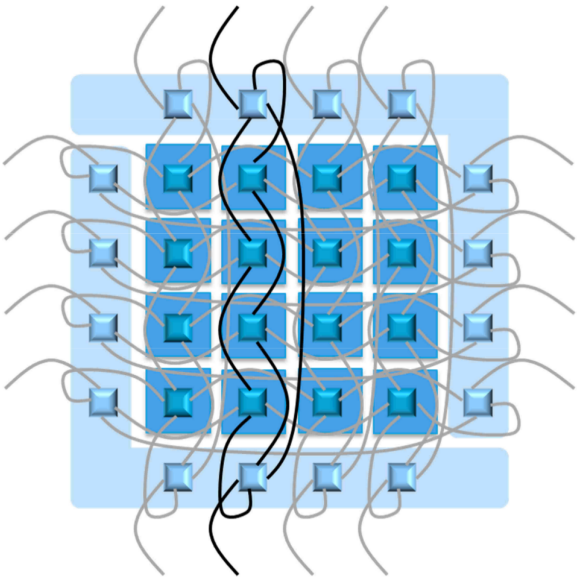


Figure 3: MPPA® NoC in detail

An important aspect for time-criticality is the NoC latency. The D-NoC which is dedicated to streaming data transfers (possible biggest delays) has been designed to operate with guaranteed services, thanks to non-blocking routers and flow regulation at source code. The routers multiplex flows originating from different directions, each with its own FIFO buffer, allowing interference on a node only if they share a link to the next one. In addition, a Round-Robin (RR) arbitration is done between the packets in these FIFOs.

All these features induce a minimal amount of perturbation on the data flow and have allowed Kalray to develop a linear programming formulation to compute the application minimal bandwidth requirements, absence of FIFO queue overflow in the routers and fairness of bandwidth allocation between the different flows.

3.1.2 The k1b VLIW core

The Kalray k1b core implements the previously mentioned 32-bit/64-bit VLIW architecture with a 7-stage instruction pipeline[8]. This allows the execution of bundles[9] containing up to 5 instructions.

The core also features two Arithmetic and Logic Units (ALUs), a Multiply-Accumulate Unit (MAU) combined with a Floating-Point Unit (FPU), a Load/Store Unit (LSU), and a Branch and Control Unit (BCU). Another important aspect to be mentioned is the endianess[10] of the architecture: *byte-addressable bit-little-endian*, meaning that it is legal to access bytes of a word and that the bit 0 of any data is always the least significant bit.

[7] Network topology to connect processing nodes in a computer system

[8] Hardware resource that splits the execution of an instruction into multiple steps, allowing parallelism and faster Central Processing Unit (CPU) throughput

[9] Group of instructions that can be executed in parallel

[10] Order of the bytes in a computer memory stored word. Usually big-endian or little-endian.

Pipeline An overview of `k1b` core pipeline can be seen in figure 4. The 7 stages are explained as follows:

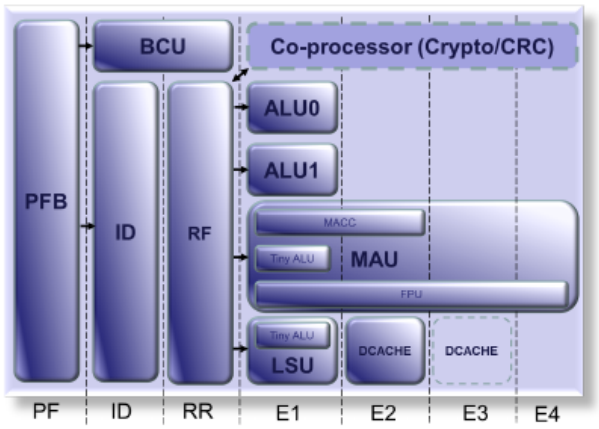


Figure 4: Kalray `k1b` pipeline

- **PF** (Prefetch): pre-load instructions to the core
- **ID** (Instruction Decode): fetch, align, decode and dispatch an instruction
- **RR** (Register read): read the necessary operands for an instruction
- **E1, E2, E3, E4**: execution stages; all executions begin in E1; results of instructions finished before E4 are available for bypassing

All the functional units, such as the ALU or the LSU, have their own internal pipeline structure that can be seen with more details in [1] and are out of the scope of this document.

Registers The MPPA® provides three types of register files:

- **R**: 64 General Purpose Registers (32-bit)
- **P**: 32 General Purpose Register pairs (64-bit)
- **S**: 64 Special Function Registers (32-bit)

General Purpose Registers (GPRs) can be used to store all data types that fit within their width: integers, addresses, boolean conditions, single or double precision floating-point. Special Function Registers (SFRs) are used for program and system control and as status registers. An introductory overview of the registers and their functions can be seen in figures 14 and 15. For further information please refer to [1].

MPPA® Instruction Set Architecture (ISA) As already mentioned, the MPPA® implements a VLIW core, leading to a very particular instruction set, completely different from the standard x86 ISA.

Due to the pipeline and parallel operation units, these instructions are regrouped into bundles in which they are encoded in ordered words called syllables. In assembly language a valid bundle is, therefore, an ordered subsequence of:

```
BCU
ALU0
ALU1
MAU
LSU
;;
```

Following the MPPA® assembly rules, each instruction is separated from the next by an end of line. Two semicolons ";;" mark the end of a bundle. In the above listing the acronyms represent instructions that can be executed in the respective unit. For example, the `BCU` could be replaced by an `IGOTO`, the `ALU` could be replaced by an `ADD`, and so on. For further details of the ISA and valid bundle operations please refer to [1].

3.2 Motivation

The MPPA® has, as shown, a unique architecture. Managing all of its features and being able to write complex programs in bare-metal[11] is rather challenging. To solve this problem, the services provided by an OS are of great help.

Kalray software engineers have already ported multiple OSs to the MPPA® such as RTEMS and even Linux. They have also developed their own simplified systems called μtask[12] and simpleOS[13]. Third-party OSs have also been ported by partner companies such as eMCOS by the japanese eSOL and ERIKA Enterprise by Evidence Srl.

Even though the last two mentioned ports are from RTOSs, none of them have been projected exclusively for the avionics market. Therefore, they do not necessarily respect the ARINC653 specification, leading to the following problems when providing such non-compliant OS to avionics costumers:

- The general purpose APplication/EXecutive (APEX)[14] interface for application software is not available and the development will have to be done using a different API. This may cause compatibility issues with older programs and a productivity decrease due to the learning curve of a new API.
- The concept of partition may not be implemented, leading to applications that are not spatially and temporally isolated.
- The final software produced has the same (or a lower) level of certification than the OS, leading to usage restriction.

This problematic has brought up the urge of an RTOS available for the MPPA® that is compliant with the ARINC653 specification, and preferentially open-source. The defence and avionics market may then have their attention drawn to the possibilities offered by Kalray's processor. Of course the process of porting an OS to a new architecture is not easy and many aspects must be analysed before the implementation, particularly:

- Different ISA, forcing the rewriting of assembly parts of the kernel

[11] A computer running without an OS.

[12] Thread support functions with an Application Programming Interface (API) mirroring the classic POSIX pthread implementation.

[13] Quite similar to μtask but also providing scheduling primitives between threads.

[14] A special type of API defined by ARINC 653 specification. More details can be seen in section 4.2.1.

- The boot, initial steps and core configuration must be adapted but remain similar
- Memory map changes with new peripherals
- Possible optimizations can be made with a new ISA and they should be made

3.3 Objectives

The final goal of the internship is to obtain an ARINC 653 OS on the MPPA® Compute Clusters, starting from an open-source single-core implementation. The main tasks to be accomplished can be listed as follows:

1. Analyse and choose an ARINC 653 compliant OS which will be then ported
2. Translate or re-implement the architecture dependant kernel files
3. Adjust the remaining kernel core files for the MPPA® architecture
4. Adjust the OS libraries as the underlying modifications may have consequences for them
5. Optimize the system (single-core to many-core perspective change)
6. Simplify the system as the Part 4 is a subset of the original ARINC 653 standard

The tasks 5 and 6 are optional and may be executed depending on the remaining time after a single-core port has been successfully accomplished. A more detailed technical view of the tasks can be seen in section 5.3, while a planning using a Gantt chart can be seen in section 8.

Concerning the quality and robustness of the final result, it should meet the criteria specified in section 5.5, which can be summarized in: having a functional system; respecting the ARINC 653 standard, performance and timing requirements; and being integrated in Kalray's repository. These criteria are based on the ultimate objective of the internship: an open-source RTOS that can be exploited by avionics costumers and used to develop applications using Kalray's environment.

4 Technical aspects

4.1 Time-Critical Computing on the MPPA

Time-Critical applications are defined as the association of time constraints with information manipulation activities such as acquisition, processing, transport, storage, coordination and delivery [2]. Meeting time constraints requires having a suitable computing model of an application and a computing platform whose OS or run-time software, architecture and implementation support at least the following properties:

- *Deterministic computations*: given the same system environment, inputs and event timing, computation results should be the same.
- *Deterministic and predictable response times*: given the same system environment; inputs and event timing, computation output should take the same predictable time.
- *Composable execution and communication lines*: updates of the system functionality should have commensurable effects on timing properties.

The main issue with many-core platforms and time-critical applications is ensuring timeliness[15] of computation and communication, given the logical (e.g. code critical sections) or physical interference (e.g. memory hierarchy) of tasks that execute concurrently [3].

4.1.1 MPPA general architecture features for Time-Critical Applications

The astuteness to ensure that the MPPA® is suitable for these type of applications is to, first of all, guarantee the timing constraints at the core and CC level, then combine the multiple clusters and finally connect them to external interfaces through the NoC and synchronisation capabilities.

This is possible because the core architecture makes timing analysis more precise than on classical multicore devices with shared caches (c.f. 4.1.2), while the NoC communication can be configured to meet certain limits on the rate and latency of data transfers (c.f. 3.1.1).

4.1.2 MPPA core architecture features for Time-Critical Applications

The VLIW architecture of MPPA®'s core was implemented with the elimination of timing anomalies[16] in mind. The following properties assure their absence:

- All memory access instructions exist in cache and uncached variants.
- Instruction and data caches implement the Least Recently Used (LRU) replacement policy.
- Data caches are write-through / write-around and are complemented with a write buffer
- The effects of possible hazards or stalls in instruction/execution pipeline are only transient for time-critical applications
- There is no branch prediction

[15] The state of being punctual, having time precision

[16] Situation where a local worst-case execution time does not contribute to the global worst-case [4].

- There is no hardware support for branch prediction or out-of-order execution, which could lead to non-deterministic or data-dependent timing.

Consequently, the Kalray VLIW core can be classified as fully time-compositional, i.e. with no timing anomalies. An accurate static timing analysis can thus be done at the core level using predictable and composable execution times, following local worst-cases.

4.1.3 MPPA practical usage in Time-Critical Applications

Some examples of applications that can benefit from the aforementioned MPPA® features are:

Control-Command Applications High safety and security requirements, typical in avionics, transports, medical or industrial domains. The software is written in a model-based programming that generates `C` code which can be deployed locally in different CC without hardware or software interference between them (except for planned global interactions).

Mixed-Criticality Applications Composed by individual programs with different criticality requirements (hard real-time, soft real-time, streaming or interactive). The goal of such applications is to increase the system utilisation, taking advantage of underused hardware resources and allocating them to less critical programs. The spatial partitioning between the cores and natural boundaries provided by the CC help the deployment of this kind of application. Moreover, there is hardware support for fast barrier synchronizations across cores that allows advanced mixed-criticality execution techniques.

Latency Constrained Application High performance with deterministic or even real-time execution requirements that can only be met by parallel processing. Typical classes of such applications include: computer vision applied to self-driving vehicles, industrial robotics and online processing of large-scale physical instrumentation. The MPPA® architecture provides high performance Ethernet (10 Gb/s), PCIe (Gen3) and Interlaken (10 Gb/s) directly connected to the NoC that guarantees timing services. Furthermore, parallel execution patterns can have its worst-case execution time easily calculated when running inside a CC.

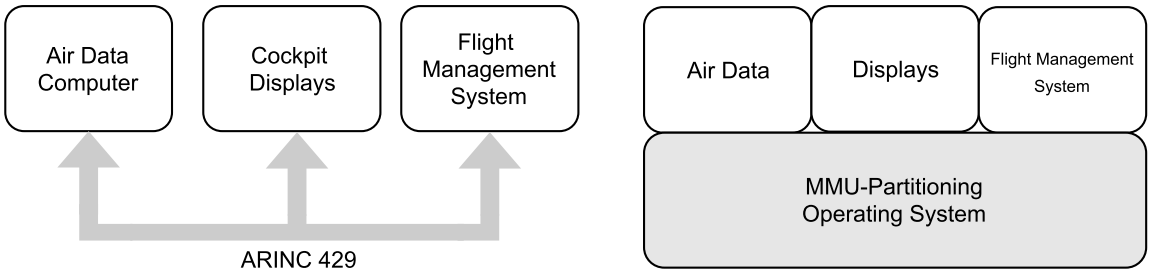
4.2 State of the Art

4.2.1 ARINC 653 standard specification

Aviation industry has been transitioning from the classical federated[17] structure to Integrated Modular Avionics (IMA). This new concept allows avionic subsystems to be grouped inside a limited (or even unique) set of processing units, managed by a dedicated RTOS. As long as the OS provides a proper API and meets the safety requirements, IMA has several advantages over a federated structure: excellent software re-use, portability and modularity, reducing costs of a re-certification process. Nevertheless, as IMA is still a modern methodology, it has a higher complexity of design and certification.

The ARINC 653 standard is closely connected to the IMA concept. Developed by aviation experts to provide "the baseline environment for application software used within IMA and

[17] Avionic subsystems with a dedicated microprocessing unit for each functionality, communicating through ARINC standard 429 (c.f. figure 5a)



(a) Federated System (b) IMA

Figure 5: Diagram exposing the differences between federated and IMA systems

traditional ARINC 700-series avionics". Its primary objective is to define a general purpose APEX interface between the OS and application software. The specification includes interface requirements between application software and OS and a list of services that allow application software to control scheduling, communication and status of processing.

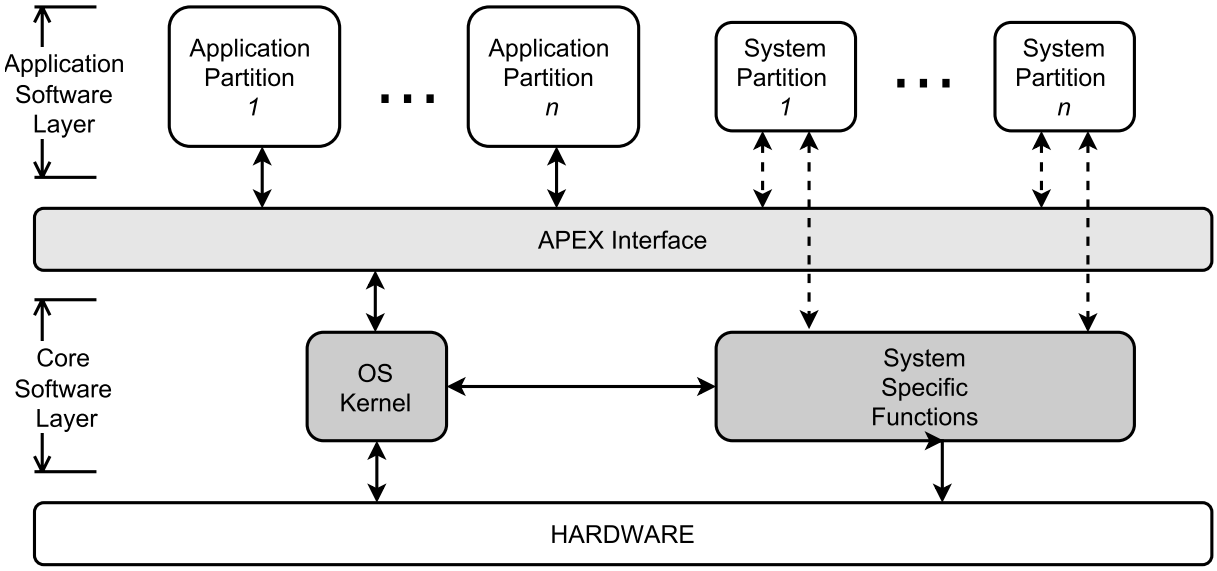


Figure 6: Generic structure of an ARINC 653 compliant system

The key concept behind the ARINC 653 specification is the **partition**. It is nothing more than a container for an application assuring that its execution is both spatially and temporally isolated. Partitions are divided in two types: applications partitions and system partitions. The former executes avionics applications and exchanges data with the system through the APEX interface. The latter is optional and may provide services not available in the regular APEX, such as device drives or fault management.

ARINC 653 Part 1 Initial version of the ARINC 653 standard, published October, 2003 [5] after several drafts and modifications. Defines the core aspects and required services, particularly:

- Time and space partitioning
- APEX interface, of which the main components are:
  - partition management,

- process management,
- time management,
- memory management,
- inter-partition communication,
- intra-partition communication,
- health monitoring.

The whole set of ARINC services is a heavy task to implement and requires a complex system to support it. Therefore, simplifications have been suggested and a supplement to the standard was created.

ARINC 653 Part 4 on the MPPA A supplement of the ARINC 653 standard, released June, 2012 [6] prepared to support controllers and relatively simple avionics. It is a subset of services specified in ARINC 653 Part 1, simplifying the interface. In the subset services, partition scheduling is restricted to only one partition time window within the partition's period. Process management uses a dual-process model with at most two processes within a partition.

The internship reported in this document is part of the CAPACITES project which determines that the OS ported to the MPPA® should be compliant at least with the ARINC 653 Part 4. Moreover, the specification states that the application partitions should execute on CCs, while the system partitions, if there is any, will run on IOCs.

As the Part 4 profile prescribes, application partitions only have two processes, a periodic foreground process and an aperiodic background process. As a result, there is no need to implement the intra-partition communication and synchronization objects such as blackboards, events and semaphores. Inter-partition communication and synchronization is provided by message queuing and message sampling, which will be supported by the NoC (c.f. paragraph 3.1.1) interfaces of the MPPA®.

The final proposition is then to run one ARINC 653 Part 4 partition per PE core in the CCs, in a configuration known as Bound Multi-Processing (BMP). This type of configuration allows the developer to bind any process (partition) and all of its associated threads (processes) to a specific core while using a common OS across all cores. An illustration of this proposition can be seen in figure 7.

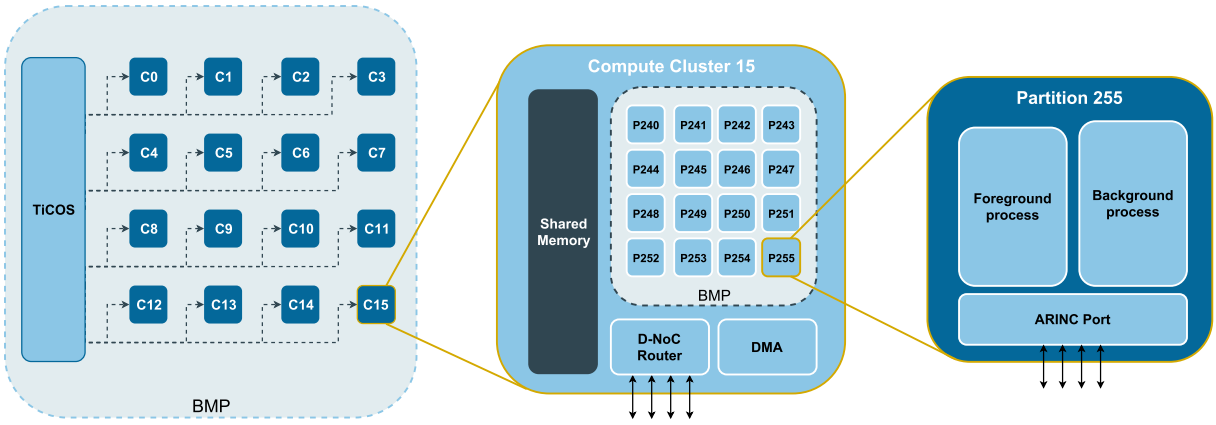


Figure 7: ARINC 653 Part 4 proposed implementation on the MPPA®

4.2.2 Available compliant OS

In this sub-subsection possible OS options for the porting process will be presented. A brief introduction of each system is given, the decision, however, is postponed to the section 5.1 where proper justification concerning the choice is provided.

TiCOS TiCOS is a time-composable real-time operating system developed within the framework of the PROARTIS project supporting the ARINC653 software specification and originally targeting the PPC[18] architecture. TiCOS is based on POK, a light weight operating system implementing the ARINC653 standard and distributed under the BSD license.

Its source code is publicly available, along with documentation and some introductory examples at the following link:

`https://github.com/UPD-RTS/TiCOS`

INRIA AOSTE OS Open-source implementation of an RTOS with ARINC 653 personality based on the aforementioned POK OS. Developed by the team AOSTE, hosted by the INRIA laboratory, that conducts research mainly on real-time embedded systems.

Its creation was motivated by the deficiencies of POK and also to obtain practical systems to be tested in AOSTE's framework of Worst-Case Execution Time (WCET). It is not publicly available so the usage of this RTOS requires the collaboration of INRIA in providing its source code, documentation and support.

[18] PowerPC, a ISA created by the 1991 Apple-IBM-Motorola alliance.

5 Solution undertaken

5.1 OS choice

The OS chosen to be ported was TiCOS. Its public availability, relative stability (been in development for 4 years) and proximity between the internship tutor B. Dinechin and its developers were the main points taken in account for the decision. A GitHub repository was created for this project inside the same organization (UPD-RTS) as the original version. More details in subsection 5.4.

The high level architecture of TiCOS can be seen in figure 8. There are two main layers: kernel layer and application layer. The first is responsible for implementing the OS services, while the latter is composed by the user application, the ARINC library layer, the core library layer and finally the middleware library layer. The libraries services are implemented through system calls to functions inside the kernel layer.

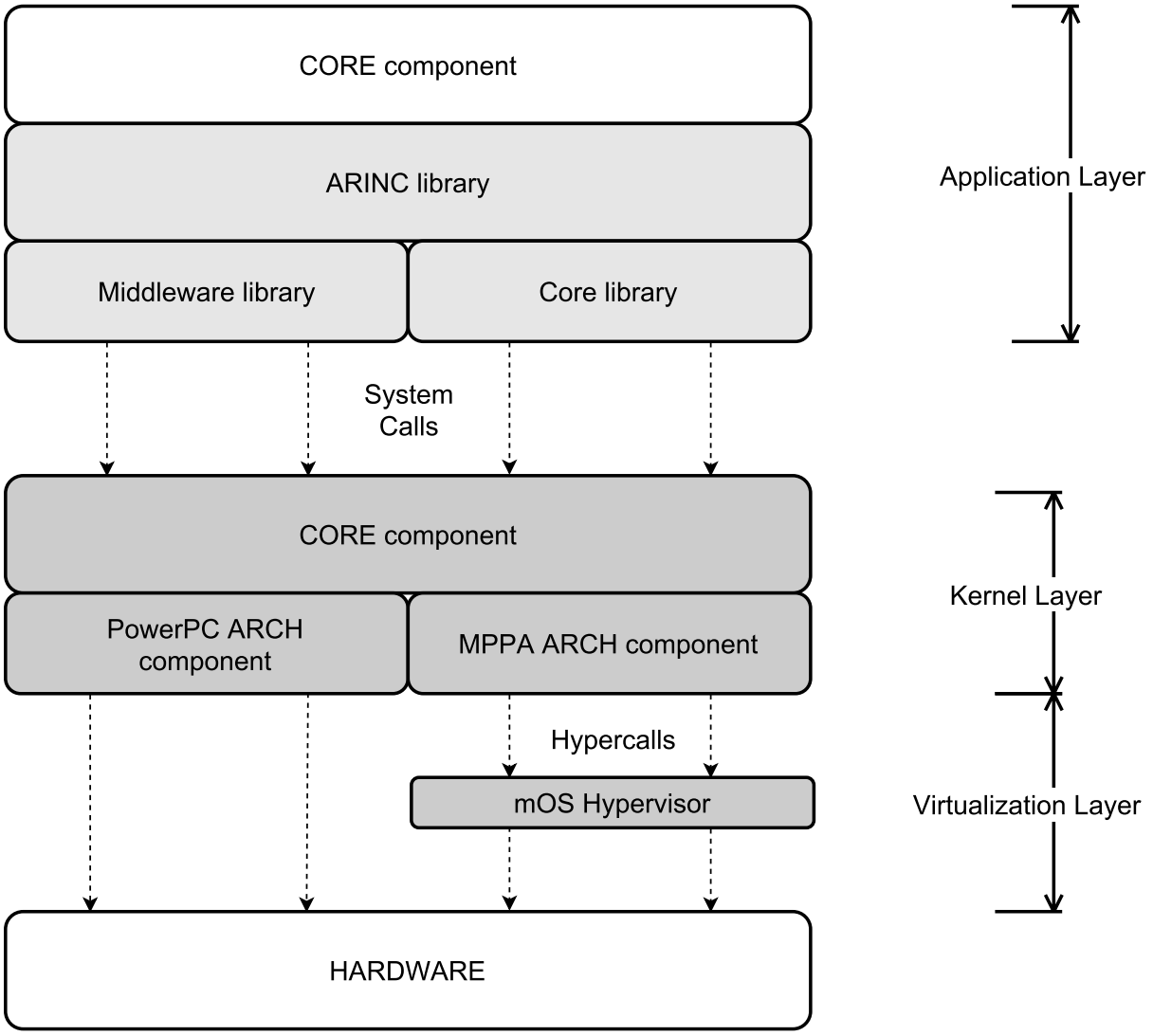


Figure 8: TiCOS high level architecture with MPPA® entity

Normally this mechanism would make the system start executing in supervisor (privileged) mode and call the proper kernel functions. The MPPA® port of TiCOS will embed in its arch component (as we can see in figure 8) a virtualization layer containing an exokernel hypervisor

called **mOS**. This system will be detailed in the next subsection, but in this case its main influence is that the OS will always be in user mode and the kernel functions will end up calling mOS that runs in supervisor mode.

5.2 mOS

Kalray's Hypervisor, called mOS, is a lightweight exokernel that provides two main services: processor and I/O virtualization, and strong hardware partitioning.

The exokernel concept was introduced by the MIT Parallel and Distributed Operating Systems group in 1994. Its principle is to force as few hardware abstractions as possible to further software layers, constructing a kernel that does apparently nothing and gives freedom to applications. An exokernel is supposed to be tiny and only ensure protection and multiplexing of resources, in opposition to microkernels and monolithic kernels that perform way more tasks and provide high-level abstractions.

mOS essential characteristics are:

1. All the services commonly provided by operating systems are delayed to the user-space execution level. As an example, it provides neither a scheduler, a memory allocator nor virtual memory support. Moreover, its implementation is fully static and lock free.
2. Its sole role is to ensure proper usage of the underlying hardware, control and manage access rights and ownership.
3. It has a very limited memory footprint (32 KB).
4. It allows the implementation of any complex system services in user-space. As an example, a user-space operating system may implement fully custom virtual memory support.

A key feature that eases the port of an OS is the virtualization layer provided by the hypervisor that exposes a symmetric 16 core CC (instead of an asymmetric 16PE + 1RM) with direct access to the NoC (hides also the RM interface in order to use the NoC). It also provides a virtualized view of the `k1` processor core, modifying its architectural states and behaviour.

An OS that runs with mOS support is called guestOS or libOS due to its user-space status. The interface between such system and the hypervisor is composed of hypercalls, asynchronous remote services and a shared memory region called scoreboard.

Scoreboard A structure that allows both mOS and a potential libOS to read and write information. It is composed of a set of architectural registers corresponding to the processor core, NoC, memory management, etc. They are possibly organized in their own sub-scoreboard and are updated asynchronously or synchronously by the hypervisor. There is a scoreboard per CC shared by all the cores of the partition.

Hypercalls Mostly related to updates of the architectural state of the virtual core. They are divided in fast, regular and slow types, according to register use and efficiency. An exhaustive list is provided in the internal documentation and is out of the scope of this document.

Asynchronous remote services They are important because in the MPPA® processor some services, mainly related to the NoC interface configuration, cannot be issued directly by the PE. These requests are packaged by the hypervisor in helpers, which sends them to the RM, waits for the response and raises an event when the request is ready.

The last important aspect concerning the hypervisor is the partition setup. It relies on a *binary descriptor* that allows static hardware resource allocation for an application. It also allows the application to configure its initial MMU setup. Finally, it contains the address of the scoreboard within the user binary and the application entry point. Currently, the hypervisor, user application and binary descriptor are packaged together in one ELF file and loaded into the cluster SMEM as a single unit.

5.3 Porting steps

This section will describe more in depth the services and functionalities implemented on each of the previously presented TiCOS layers. It may be seen as a guideline or checklist for the development covered in section 6. The document presents the port of an RTOS originally targeting a mono-core system and, therefore, a single-core port is the first natural stage (c.f. subsection 3.3). Nonetheless, optimizations for a many-core architecture and simplifications regarding the ARINC 653 Part 4 specification are suggested at certain moments.

5.3.1 Kernel Layer

The kernel of TiCOS is made of three main components: an architectural dependant part (called *arch component*) and two architectural independent parts (called *core component* and *middleware component*).

The arch component is the most delicate part of the port. It accesses directly architectural functionalities to implement kernel services, requiring assembly and architectural knowledge. Particularly, TiCOS implements in this component: Board Support Package (BSP), system entry point, system calls, memory management, context switch, timer and interruption support (porting details can be seen in subsection 6.1). The core component does not need to directly use the underneath architecture, requiring potential adjustments for memory virtualization or user application loading, for example (porting details can be seen in subsection 6.2). The middleware component (inter-partition communication) should not require modifications.

As a whole, the kernel layer provides the following services:

- **Partition support:** the concept of partition is defined in sub-subsection 4.2.1 and TiCOS must implement time and space isolation among partitions, granting to each one its own space with no shared memory.
- **Port support:** ports are the ARINC way of inter-partition communication. Since partitions do not share memory, the kernel is responsible for copying messages from source ports to destination ports.

The ARINC 653 Part 4 allows the simplification of the scheduling algorithm and also the absence of lock objects used for intra-partition communication, as there are just two processes per partition. A minimal implementation of **libc**[19] is embedded in the kernel layer and will need to be MPPA® ported due to its architectural dependant parts (porting details can be seen in subsection 6.3).

[19] Library that contains common `C` functions like `printf` and `memset`, for example.

5.3.2 Library Layer

The OS library layer exposes TiCOS services through user-level functions. These functions end up in system calls to kernel services presented in the previous subsubsection. The library is divided in three parts:

- **Core library:** contains the OS's core functionalities used to implement the ARINC library. The functions contained here are not part of the ARINC 653 specification and should not be used directly as the application will not be portable.
- **Middleware library:** contains the functions that implement the majority of the functionalities offered by the ARINC library. Nevertheless, it must not be accessed directly in an user application.
- **ARINC library:** contains the functions responsible for creating and managing the structures presented in subsubsection 4.2.1. Most of them make use of the two previous libraries.

Due to the simplifications introduced by the ARINC 653 Part 4 some services do not need to be implemented, particularly the ones related to ARINC events, blackboards and buffers. Furthermore, almost the whole library layer should remain untouched during the port, except for little adjustments through `#ifdefine` directives. For the same reason the prefix `pok_` is maintained in functions names, as they are still legacy from the POK OS. A general overview of the library layer is presented in figure 16 (ARINC 653 Part 1) and 17 (ARINC 653 Part 4).

5.3.3 Build chain

The compilation process of TiCOS is quite complex, having the phases described below.

Kernel compilation Two files are used to generate the kernel's executable: a configuration file (`deployment.h`) containing pre-compilation directives that configure the kernel and a linker script (`kernel.lds`). The build process creates the link object `kernel.lo` putting together all the objects forming this layer (c.f. figure 9). The contents of each object was described in sub-subsection 5.3.1, although its was not mentioned the subdivision of the *arch component*, formed by `prep.lo` (BSP specific functions) and `ppc.lo` (all PowerPC specific code).

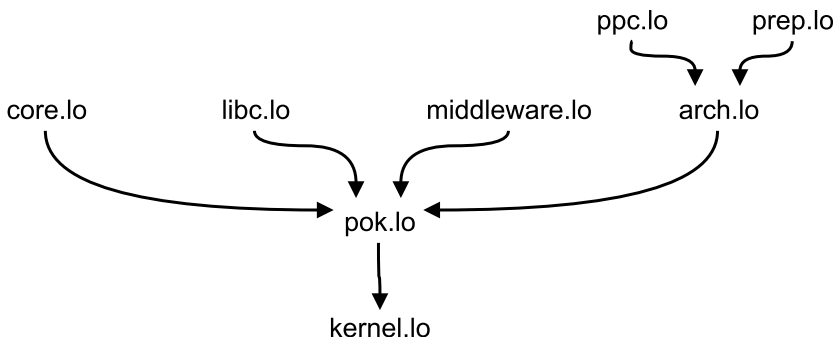


Figure 9: Visualization of the kernel build process and its compilation objects.

Partitions compilation Each partition ought to run in the system is compiled separately into an Executable and Linking Format (ELF) file containing user code and TiCOS library. Once again, two files are used to generate a partition's executable: a configuration file (`deployment.h`) to compile TiCOS library statically and a linker script (`partition.lds`) that joins the user code (`main.c`, `activity.c`) and the shared library (`libpok.a`).

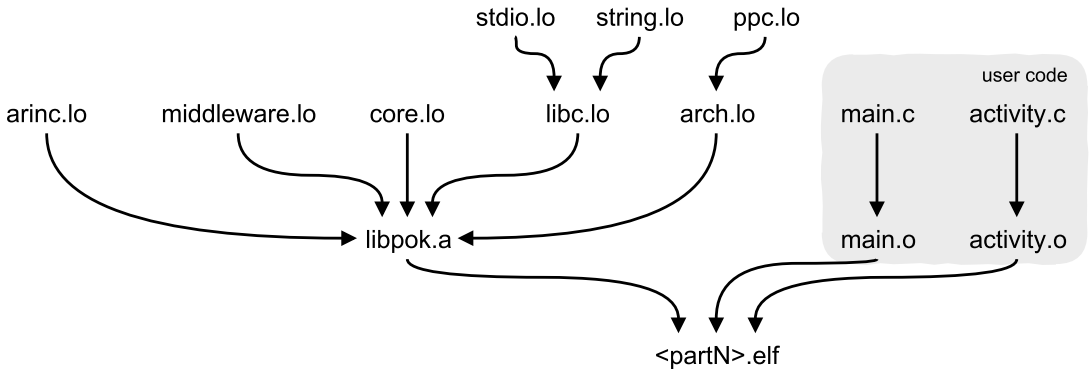


Figure 10: Visualization of a partitionN build process and its compilation objects.

System compilation This phase consists on integrating kernel and partitions into one executable. Each partition ELF is padded to respect alignment and then all partitions are assembled in a single binary called `partitions.bin`. Afterwards, a file containing the sizes of each partition (`sizes.c`) is generated, compiled into `sizes.o` and embedded with `partitions.bin` using `obj-copy`. Finally, `sizes.o` and `kernel.lo` are linked together to create `pok.elf`.

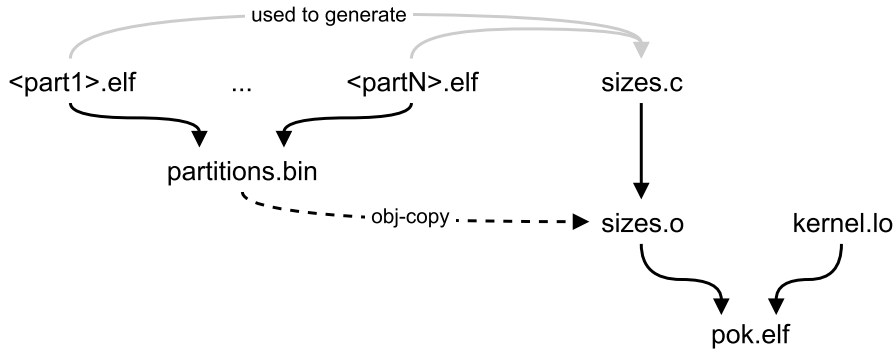


Figure 11: Visualization of a partitionN build process and its compilation objects.

All the link objects named `ppc.lo` in the above figures are the product of architectural-dependant layers that will need port to the MPPA® architecture. Moreover, configuration files that predefine directives, need to include Kalray's compilation tools. The linker scripts that define memory organization at execution time will also need to be modified to include mOS and other MPPA® particularities.

An important part of the building process is also embedding functional user applications with the OS. The pre-configured examples that come with TiCOS will need to be adapted to the ARINC 653 Part 4 standard and also the MPPA® features. They are the main test vectors of the system as a whole. The validation process will be discussed more in depth in subsection 7.1.

5.4 Employed tools

All the machines at Kalray have as main OS Linux because the tools and drivers of their products work only under this environment. Additionally, many essential tools for developers in general are available exclusively on Linux. The distribution used is CentOS 7, the same used in Ensimag machines, making the adaption even easier. Usage of Windows is possible through servers in order to exploit the Office tools or store important files.

As already introduced in subsection 2.2 a great number of GNU tools were ported by Kalray. They make the MPPA® processors usable through standard programs and file formats that developers are already familiarized with, speeding up the learning curve progress. The 3 main tools used were:

- `k1-gcc`: `gcc` port that allows the compilation of executable files for the MPPA® (may call `k1-ar` to compile assembly code and `k1-ld` to link files)
- `k1-readelf`: `readelf` port that allows to obtain human readable information from a compiled ELF program
- `k1-gdb`: `gdb` port used to execute an application in debug mode, exposing useful information at run time to track down errors

The suite of Kalray tools, already compiled, is easily obtainable from an internal machine through an application called `kalrayEnv`. By giving a `git` commit ID or branch the application downloads the compiled suite for the machine architecture and Linux distribution. Afterwards, launching the command `kenv switch` modifies the working environment and lets all the Kalray tools available and already in the path, including the simulator and hardware runners.

The project was developed in a MPPA® Developer Machine using an SSH connection from a local machine. Concerning the project versioning, a GitHub repository was created inside the same GitHub organization were TiCOS (and other ports) were developed. The source code and current progress is available in the following address:

`https://github.com/UPD-RTS/mppa-TiCOS`

5.5 Evaluation protocol

The first criterion of success is the functional aspect of the solution. The port of TiCOS should work on the MPPA® and ease the development of user applications based on given examples. Moreover, the RTOS must implement the services defined the ARINC 653 Part 4 standard while still respecting the security, space and time partitioning aspects.

A second criterion is TiCOS performance and power consumption, given the MPPA® embedded features for such measurement. The results can help in the validation of timing requirements and provide useful numbers that may be used to draw clients' attention.

Finally, the integration of this project in Kalray's `git` repository with the standard runtime OSs provided with the mOS hypervisor, or even as a standalone system is the third criterion. This would show that the project has accomplished its objectives and meets the system quality requirements of Kalray.

Obviously, this is a progressive evaluation protocol, meaning that the aforementioned criteria will be observed progressively, as the project goes forward and more core functionalities or ARINC services are added and tested.

6 Implementation

In this section, the work done to extend TiCOS in order to support the MPPA® architecture is presented. The single-core porting process was carried following the steps introduced in section 5.3 leading to an incremental development of the operating system layers:

1. **Kernel:** divided in three sublayers: **arch**, **core** and **libc**.
2. **Library:** divided in three sublayers: core, middleware and ARINC but treated together in this section.

The development technique used allowed to focus firstly on the most architectural dependant layer, dealing with specific and concise functions that together form the main functionalities of the OS. More effort and time were dedicated to this stage as important decisions regarding the porting are concentrated here. Furthermore, the low-level porting is not merely an assembly-translation of code, but an attentive look to identify dispensable operations and possible optimizations.

Once this stage was finished, all the architectural dependencies were already ported to the MPPA® processor and the changes performed in the next layers could be targeted to the services proposed by the OS.

6.1 Architectural modifications

As shown in section 5.3.3 the architectural layer of TiCOS is compiled in a link object naemd `arch.lo`. Its source files are machine related and mainly assembly code, which implies in non-reusability of the original PowerPC code. Therefore a new directory named `arch/mppa` has been created containing all MPPA®-dependent code, replacing the `ppc.lo` object in figure 9 with `mppa.lo`.

6.1.1 Clock

The Kalray k1b core, introduced in section 3.1.2, provides a Real Time Controller (RTC) in form of two 32-bit timers `$t0v` and `$t1v` that contain the current value of the timers, and are reset to `0x0`. These timers can be chained in order to provide a 64-bit timer.

In addition to that, two 32-bit timers reload value registers `$t0r` and `$t1r` are available and are also reset to `0x0`. They are directly related to how the MPPA® RTC works.

A timer, if enabled, is decremented on each tick of the clock divider, until it reaches 0. Upon the next tick, the timer value is loaded with its reload value ($t0v ← $t0r or $t1v ← $t1r) When such an underflow occurs, depending on the configuration of the Timers Control Register ($tcr), a pulse on the corresponding interrupt line is generated.

The `timer.c` file contains the software implementation of such hardware characteristics, using the hypervisor. In particular, two mOS functions are worth mentioning:

```
1 static __inline__ void
2 mOS_timer_general_setup(void)
3 {
4         uint32_t tcr = _K1_DEFAULT_CLOCK_DIV; /* All other fields 0 */
5         __k1_club_fast_hypercall1_noret (MOS_VC_SET_TCR, tcr);
6 }
```

mOS_timer_general_setup sets the $tcr aforementioned register with the _K1_DEFAULT_CLOCK_DIV and is called at system initialization.

```
1  static __inline__ int
2  mOS_timer64_setup(unsigned long long int value, unsigned long long int reload,
3                                                  unsigned char itDisable)
4  {
5          mOS_transtype64_t t_value, t_reload;
6
7          t_value.dword   = value;
8          t_reload.dword  = reload;
9
10         return __k1_club_syscall5 (MOS_VC_SETUP64_TIMER,
11                 t_value.low,
12                 t_value.high,
13                 t_reload.low,
14                 t_reload.high,
15                 (int) itDisable);
16 }
```

mOS_timer64_setup is used to set the chained 64-bit timer with its value, reload value (usually the same) and a control char to disable or enable the interrupt associated with the timer. Both functions end up in hypercalls (or syscalls) as the majority of mOS API.

In order to have a global system timer, the Debug System Unit (DSU)[20], provides an independent timestamp for each PE, that is cycle-accurate, updated at each tick and stored in a particular memory address. The function used to exploit this resource is

```
static inline uint64_t mOS_dsu_ts_read(void);
```

which, once again, finishes in a syscall that performs a memory access, core-aware, at the address corresponding to

```
&mppa_trace[__k1_get_dsu_id()]->timestamp
```

where mppa_trace is a particular zone of the DSU and __k1_get_dsu_id() returns the current working core, assuring to retrieve the appropriate timestamp value.

6.1.2 Thread context

The original version of the OS implemented two types of data structures that represented a thread's context: volatile_context_t and context_t. This was due to the different types of registers provided by the PowerPC architecture: *volatile* (may be modified by functions), *non-volatile* (must be preserved by functions) and *dedicated* (must be used only for specific purpose, similar to SFRs). Moreover, these structures were allocated on the top of the thread's stack. Together they formed a multiple of a quadword since, according to the PowerPC EABI [7], a stack frame has to be quadword aligned.

The MPPA® thread context is slightly different:

- There is no need to have two different structures holding *volatile* and *non-volatile* registers as the MPPA® architecture does not impose these saving constraints.

[20] A shared memory space between all cores (at cluster level), used for debug purposes.

- MPPA® architecture and, in particular mOS, do not specify any alignment to stack frames and neither obligate to place the thread's context on the stack.

The data structure containing the processor context is presented in listing 1.

Listing 1: C structures representing the MPPA® context.

```
typedef struct {
    union {
            __k1_uint32_t regs[64]; // 64 GPR
            __k1_uint64_t force_align[32];
    };
    // System function registers (There are 64, 50 used and few important for the ctx)
    __k1_uint64_t spc; //shadow program counter
    __k1_uint64_t sps; //shadow processing status
    __k1_uint64_t ra; //return address
    __k1_uint32_t cs; //compute status
    __k1_uint32_t lc; //loop counter
    __k1_uint64_t ps; //processing status
    __k1_uint64_t ls; //loop start address
    __k1_uint64_t le; //loop exit address
} __k1_context_t;

typedef struct
{
    __k1_context_t k1_base_ctx;
    __k1_uint32_t ssp; //shadow stack pointer
    __k1_uint32_t sssp; //shadow shadow stack pointer

} context_t;
```

As we can see in the listing there is a `__k1_context_t`, which is actually defined in low-level Hardware Abstraction Layer (HAL) code, just included by TiCOS code. It contains the 64 32-bit wide GPRs and the most important SFRs to store in a context (all of them presented in section 3.1.2).

The OS context uses this MPPA® base context and adds two virtual registers, `ssp` and `sssp` that are managed by mOS in case of nested traps/interruptions. The total size occupied by `context_t` is `0x130`, directive-defined and used in other parts of the system.

In addition to that, two assembly macros are used when dealing with the context:

```
.macro _vk1_context64_save to
.macro _vk1_context64_restore from
```

They are also defined in HAL code and are helpers to save/restore a full context in accordance with the `context_t` type. The arguments to/from must be registers that contain the memory address where to save/from where to restore the context.

A pointer to the current executing thread's context is now maintained by the system and updated by the scheduler, called `pok_current_context`. It is used in some assembly files to save and restore a thread's context before and after performing scheduling decisions, in the occurence of a system call, etc.

6.1.3 Memory management

The original version of TiCOS made use of PowerPC virtualization functionalities to manage the memory. Even though the MPPA® contains a Memory Management Unit (MMU) and, thus, offer the possibility of virtualization, sizing rules and static configuration required by mOS in order to exploit the MMU are complex and not compatible with the application flexibility offered by the OS.

For this reason, address range checks, done by the OS at precise moments, were chosen as memory management system. This technique allows to divide the memory in parts, each one with different privilege levels.

For instance, the thread's context can be moved from the top of the thread's stack to a dedicated OS structure allocated in memory, containing all the user application threads contexts. Using the memory protection technique this array of contexts is stored in a protected memory area with kernel priviilege.

The linker script and the boot process of TiCOS are what define the memory map of the system. Firstly, mOS is booted, discovers what is the current running core and start using a boot stack reserved specifically for that core. During the booting phase each partition is loaded into memory and for each partition a main thread is created (with its stack) having as responsibility creating and starting all the partition's processes. Each ARINC process is mapped to a thread, having, consequently, its stack as well. Moreover, an idle and kernel thread are created for system use. In summary, the memory to be allocated is:

- A stack for the kernel (mOS boot stack and mOS kernel stack)
- A stack for the idle thread
- A stack for each partition's (main thread)
- A stack for each partition's process

The kernel linker script (see listing 2) pre-reserves the space for the boot and kernel stacks inside mOS scoreboard. It also defines global labels that can be used to access these addresses later.

Listing 2: Kernel linker script snippet

```
1  BOOT_STACK_SIZE = DEFINED(BOOT_STACK_SIZE) ? BOOT_STACK_SIZE : 0x100;
2  KERNEL_STACK_SIZE = DEFINED(KERNEL_STACK_SIZE) ? KERNEL_STACK_SIZE : 0x400;
3
4  MPPA_ARGAREA_SIZE  = 0x1000;
5  .scoreboard ALIGN(0x100) : AT ( ALIGN(LOADADDR(.bsp_config) + SIZEOF(.bsp_config), 0x100))
6  {
7      _scoreboard_start  = ABSOLUTE(.);
8      KEEP(*(.scoreboard))
9      _scb_mem_frames_array = ABSOLUTE(.);
10     . += (INTERNAL_RAM_SIZE >> 15);
11     _scoreboard_end    = ABSOLUTE(.);
12     _scoreboard_boot_stack_end = ABSOLUTE(.);
13     . += (BOOT_STACK_SIZE*16);
14     _scoreboard_boot_stack_start = ABSOLUTE(.);
15     _scoreboard_kstack_end = ABSOLUTE(.);
16     . += (KERNEL_STACK_SIZE*16);
17     _scoreboard_kstack_start = ABSOLUTE(.);
18     MPPA_ARGAREA_START = ABSOLUTE(.);
```

```
19      . += MPPA_ARGAREA_SIZE;
20  }
```

In particular, the label `_scoreboard_boot_stack_start` is used to initialize the stack pointer's register (`$r12`) in the entry point of the OS (c.f. listing 3) and `_scoreboard_kstack_start` is used to switch from the user stack when entering an interrupt handler or syscall hander (c.f. listings 5 and 8).

Listing 3: System entry point

```
1  .section .pok_boot, "ax", @progbits
2  .align 8
3
4  .global _data_start
5  .global _vstart
6  .proc _vstart
7  .type _vstart, @function
8  _vstart:
9          make $r14, _data_start          # r14 is used at some compilation modes
10         ;;                              # ;; ends instruction bundle
11         get     $r5 = $pcr              # Get processor id
12         ;;
13         extfz   $r2, $r5, 15, 11        # between 0 - 15 (PE), 16 (RM) in $r2
14         ;;
15         cb.eqz  $r2, __proceed          # if it's PE0 it jumps to proceed
16         ;;
17         scall   MOS_VC_IDLE1            # otherwise the PE waits the end of
18         ;;                              # the booting process
19 __proceed:                              # Defines stack for each PE in the scoreboard
20         make    $r3, BOOT_STACK_SIZE    # Normally BOOT_STACK_SIZE = 0x100
21         ;;
22         ctz     $r3, $r3                # 8 (number of zeros in $r3) stored in $r3
23         ;;
24         sll     $r2, $r2, $r3           # shift proc ID << $r3
25         ;;
26         add     $r2 = $r2, 8            # calculates how much the boot stack pointer
27         ;;                              # should be moved for the current PE
28         neg     $r2, $r2
29         ;;
30         make    $r12 = _scoreboard_boot_stack_start - 8
31         ;;
32         add     $r12 = $r12, $r2        # move stack pointer
33         ;;
34         call    pok_boot                # C boot function
35         ;;
36         wpurge
37         ;;
38         make    $r0 = -1                # should not endup here!
39         ;;
40         scall   1
41         ;;
42         hfxb    $cs, $r1
43         ;;
44         idle1
45         ;;
46         goto    -8                      # should never return
```

```
47          ;;
48 .endp _vstart
```

Also, through the kernel's linker script a label `_pok_heap_start` is defined, identifying the start of the heap section. When the kernel is loaded into memory the `_pok_heap_start` pointer indicates where to start allocating threads stacks. The space allocation works as follows:

- For system threads:
  - The kernel thread performs an allocation for its context (size `0x130`) starting from the `_pok_heap_start` pointer.
  - The idle thread does a stack and a context allocation, both starting from the `_pok_heap_start` pointer.
- For each partition:
  - An amount of memory large enough to hold all the text and data sections of the partition is allocated (i.e. the pointer `_pok_heap_start` is moved forward)
  - An amount of memory large enough to hold all of its threads stacks is allocated.
  - From this allocated memory, some is reserved for the main thread's stack.
  - The needed space (`0x130`), starting from the `_pok_heap_start` pointer, to hold the main thread's context is allocated.
- For each partition's user thread:
  - Some space in the partition's memory is reserved for the thread's stack (on top of the main thread's stack)
  - The needed space (`0x130`), starting from the `_pok_heap_start` pointer, to hold the thread's context is allocated.

The size of the kernel stack is set by default while the size of user threads stacks and idle thread stack can be configured using the directives `USER_STACK_SIZE` and `IDLE_STACK_SIZE`.

The applications threads stacks are placed in partition's memory starting from its bottom, one after the other, the main thread being the first. The memory layout created by TiCOS follows the structure depicted in figure 18.

6.2 Core modifications

The architectural modifications are reflected slightly to the core layer, as it is still bounded to the kernel and directly exploring the features from the underneath level. Most of its code has not changed though, and no new architecture-dependant directory was created. To implement the needed modifications two pre-compilation directives are employed: `POK_ARCH_PPC` and `POK_ARCH_MPPA`. They are used inside of the code to generate a configured kernel for PowerPC architecture and MPPA® architecture, respectively.

6.2.1 Partition loader

The partitions compilation process was introduced in section 5.3.3. As the MMU is not used and the concept of virtual address does not exist, the partitions linker script must be personalized and calculate non-conflicting starting addresses for each partition.

Listing 4: Partition linker script template

```
ENTRY(main)
SECTIONS
{
    . = SEGMENT_START(".data", DATA_ADDRESS);
    .data : {
        *(.rodata .rodata.*)
        *(.data)
        *(SORT(.init_array.*) .init_array)
        *(SORT(.fini_array.*) .fini_array)
        *(.bss)
    }

    . = SEGMENT_START(".text", TEXT_ADDRESS);
    .text : AT ( ALIGN (0x100)) {
        *(.text .text.*)
    }

    . =  0x1F0000;
    _stack = .;
    _end = _stack; PROVIDE (end = .);
}
```

The macros `DATA_ADDRESS` and `TEXT_ADDRESS` are relative to the kernel configuration (i.e. where the pointer `_pok_heap_start` will be placed) and also incremental, regarding the number of partitions.

Once the partitions are compiled and archived together (`partitions.bin`) they are copied into the object `sizes.o` with the following command:

```
$(OBJCOPY) --add-section .archive2=partitions.bin sizes.o
```

The kernel linker script puts the section `.archive2` inside `.rodata` protecting the partitions objects. During the booting process of the system, these objects are retrieved from this read-only section and loaded into memory, being assisted by the sizes calculated in `sizes.c`. The function responsible for this process is:

```
void pok_loader_load_partition (const uint8_t part_id, uint32_t offset, uint32_t *entry)
```

The argument `offset` of each partition was calculated with `base_addr` of the memory region reserved for the partition (c.f. 6.1.3). Without the concept of virtual address and knowing statically where the partitions will be placed, this argument is always `0` in the MPPA® version.

6.3 LibC implementation

As TiCOS is a standalone OS, supposed to be compiled successfully without any default library, its kernel must implement at least a minimal subset of the standard `libc`. In particular two architectural dependant functions were implemented and are presented below.

```
int putchar(const int x) {
        unsigned long long int ret;
        ret = __k1_club_syscall2 (4094, (volatile int) &x, 1);
        return (int) ret;
}
```

The function `putchar` is usually the final subcall of a `printf` and performs a `scall 4094`. The hardware or simulator catches this and searches for two arguments at `$r0` and `$r1`: what to print (`&x`) and its length (fixed at 1, as it is only one char at a time).

```
void exit(int level) { mOS_exit(1, level); }
```

The function exit uses the mOS API to perform a `scall 4095` (the first argument, fixed at 1, is emit_4095). The argument `level` is put in `$r0`. The hardware or simulator catches this and exit the application domain with the return value present in `$r0`.

Furthermore, some additional low level math functions were taken from internal or open-source code, such as `__divdf3`, `divmodsi4`, `__modsi3` and `__umodsi3` but are relatively well known `C` implementations and out of the scope of this report.

6.4 Library modifications

The ARINC, middleware and core libraries allow the user code to call operating system services. It is compiled separately from the kernel, yet provides access to kernel functionalities through the system call mechanisms. The services provided by the OS did not change hence this layer is left almost untouched, apart from the way system calls are performed.

6.4.1 System Calls Implementation

A library layer helps in the paradigm of kernel/user space separation providing user code access to kernel functions. A way to safely jump into kernel code must be developed. Hopefully the MPPA® architecture is similar to PowerPC and ARM, for example, in this aspect, providing a special instruction named `scall`. The listing 5 presents its use.

Listing 5: MPPA® system call implementation

```
.text
.globl pok_syscall2
.globl pok_syscall3
.globl pok_syscall4
.globl pok_syscall5
.globl pok_syscall6
.globl pok_syscall7
pok_syscall2:
pok_syscall3:
pok_syscall4:
pok_syscall5:
pok_syscall6:
pok_syscall7:
    scall $r0
    ;;
    ret
    ;;
```

In bare mode the `scall` instruction performs the system call with the number passed as argument, while disabling the interrupts. When using the hypervisor, this instruction behaves differently.

mOS catches the `scall`, ignoring its argument, and jumps to a code portion where it will prepare the register set to handle the syscall. It performs a light context save, make `$spc = &scall_handler` and `$sps = $sps_tweaked`, where tweaked means mainly disabling the interrupts. The light context saved is restored and an `rfe` instruction is executed, moving multiple registers: `$ps = $sps`, `$sps = $ssps`, `$pc = $spc` and `$spc = $sspc`. The execution then jumps to the system call handler. This address is registered at booting phase by the function

```
mOS_register_scall_handler((mOS_exception_handler_t) &_system_call_ISR);
```

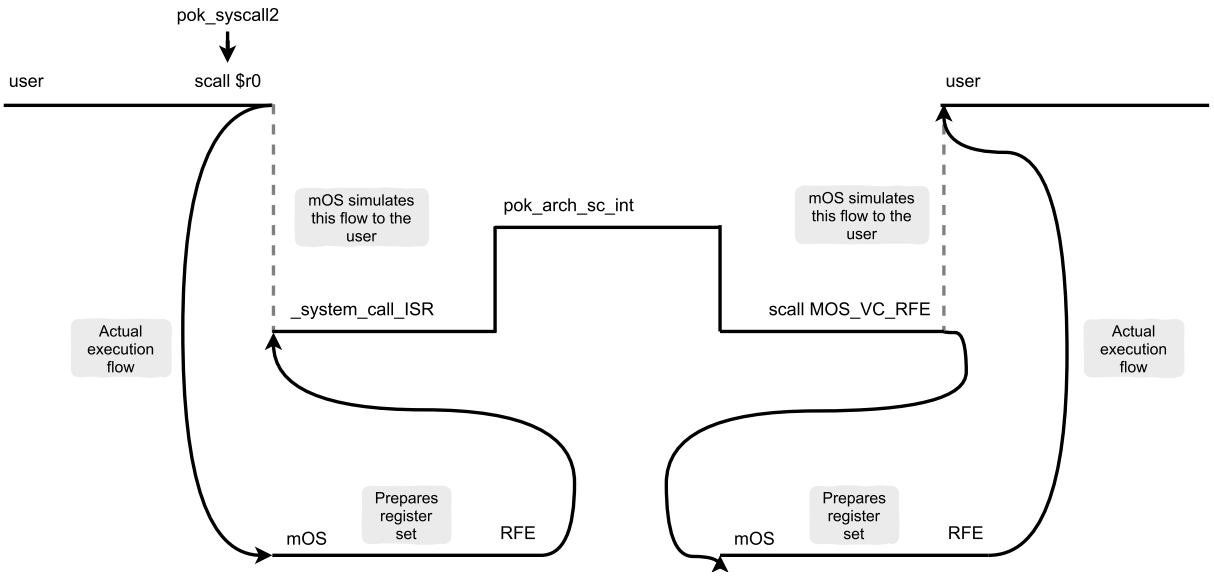


Figure 12: System call execution flow with mOS interference

The system call handler assembly code can be seen in listing 7. It prepares the OS and the calling thread, then jumps into the C function `pok_arch_sc_int` that identifies the kernel function to call and pack its arguments.

6.5 Context Switch

TiCOS has two mechanisms of context switching: *interrupt-driven* or *explicit*. The first is done through the configuration of the RTC, described in subsection 6.1.1, to expire at specific time slots where scheduling decisions must be taken. The latter is oriented to the cases of yield or run-to-completion: when a periodic thread has finished its execution and must wait the next period or when an aperiodic thread waits for an event.

6.5.1 Interrupt-driven context switching

In order to properly configure the MPPA® interruption system several actions are performed in booting phase. The first one is registering the address of the interruption handler, similarly to what is done for the system calls:

```
mOS_register_it_handler(_interval_ISR);
```

After that the function pok_bsp_time_init is called, which initializes the bsp_handlers with the specific function to call when a timer interruption arises, do the timer general configuration and associates a timer interruption with a priority level. Then it calculates the first interruption of the system.

```
pok_ret_t pok_bsp_time_init ()
{
        int err;

#ifdef POK_NEEDS_DEBUG
        printf ("[DEBUG]\t TIMER_SETUP: Freq:%d MHZ, Div:%d, Shift:%d\n",
                                POK_BUS_FREQ_MHZ, POK_FREQ_DIV, POK_FREQ_SHIFT);
#endif
        bsp_handlers_init();

        mOS_configure_int (MOS_VC_IT_TIMER_0, 1 /* level */);
        mOS_configure_int (MOS_VC_IT_TIMER_1, 1 /* level */);

        bsp_register_it(pok_arch_decr_int, BSP_IT_TIMER_0);
        bsp_register_it(pok_arch_decr_int, BSP_IT_TIMER_1);

        mOS_timer_general_setup();

        time_inter = (POK_BUS_FREQ_HZ /POK_FREQ_DIV) / POK_TIMER_FREQUENCY;
        next_timer = time_inter;
        time_last = get_mppa_tb();
        last_mppa_tb = time_last;

        err = pok_arch_set_decr(next_timer);
        return err;
}
```

Finally, the function pok_arch_preempt_enable is executed, calling several mOS functions that prepare the interrupt lines, the $ps register and ends by enabling the interruption system.

```
pok_ret_t pok_arch_preempt_enable()
{
        // clear pending flags
        mOS_it_clear_num(MOS_VC_IT_TIMER_0);
        mOS_it_clear_num(MOS_VC_IT_TIMER_1);

        /* Activate stack switch for interrupts and exceptions */
        _scoreboard_start.SCB_VCORE.PER_CPU[0].SFR_PS.isw = 1;
        _scoreboard_start.SCB_VCORE.PER_CPU[0].SFR_PS.esw = 1;

        mOS_set_it_level(0);
        mOS_it_enable();

        return (POK_ERRNO_OK);
}
```

With all these settings configured, when a timer interrupt is raised the control flow jumps to the interrupt handler registered in mOS scoreboard: _interval_ISR (c.f. listing 8). The execution flow that follows is exposed in figure 13 with commentaries about the role of each function on the left side.

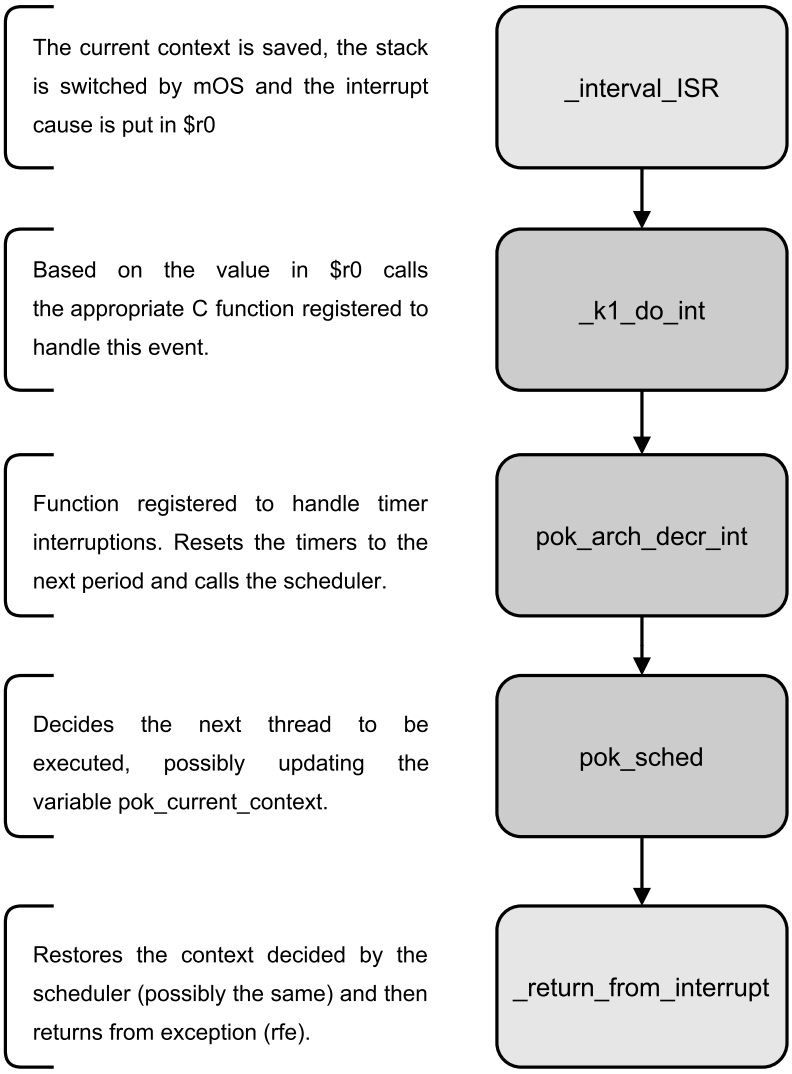


Figure 13: Interruption execution flow

6.5.2 Explicit context switching

The previously mentioned cases of explicit context switch (wait for next period, wait for event) end up calling the `pok_context_switch` function. After a scheduling decision, the function receives as parameter the address of the previous context in `$r0` and the address of the new context in `$r1`. It is a legacy function that has maintained its interface, but only `$r1` is used for the moment as the previous context should be already saved.

Listing 6: MPPA® context switch

```
1  .global pok_context_switch
2  .proc pok_context_switch
3  .type pok_context_switch,@function
4  pok_context_switch:
5          add $r12, $r12, -16
6          ;;
7          copy $r13, $r1
8          ;;
9          _vk1_context64_restore $r13
10         ;;
11         add   $r12, $r12, 16
```

```
12          ;;
13          scall MOS_VC_RFE
14          ;;
15  .endp pok_context_switch
```

7 Obtained results

This section will explain the validation process, the execution method and differences between ISS simulation and real hardware execution. TiCOS porting results are then exposed and the last subsection is dedicated to possible extensions for the system.

7.1 Validation

As introduced in section 5.3 the examples that came out-of-the-box with TiCOS were the main test vectors used. Nevertheless, the test base was expanded with the generation of additional examples. The complete list is presented below:

- **Partition/Thread examples:** minimalistic set of examples intended to introduce a new user to the system and also test its scalability regarding the number of threads and partitions. The kernel generated object incorporates the O(1) scheduler, libC and essential code for a minimal working system.
  - `arinc653-1part`: 1 partition made of 2 periodic threads
  - `arinc653-2parts`: 2 partitions each made of 2 periodic threads
  - `arinc653-3parts`: 3 partitions made of 1 periodic thread
  - `arinc653-4parts`: 4 partitions made of 1 periodic thread
  - `arinc653-4parts-2threads`: 4 partitions each made of 2 periodic threads
- **Intra-communication examples:** more sophisticated set of examples adding into the kernel the lock-objects, used in intra-partition communication mechanisms such as events and blackboards.
  - `arinc653-1event-O1`: single partition made of 2 periodic and 2 sporadic threads synchronizing on an ARINC event
  - `arinc653-1event-O1-split`: single partition made of 2 periodic and 1 sporadic thread synchronizing on an ARINC event using O1-split scheduler
- **Inter-communication examples:** even more complex set of examples adding the ARINC port services, used in inter-partition communication mechanisms such as sampling and queuing.
  - arinc653-sampling-queueing: 2 partitions each made of 4 periodic threads, 1 sampling port and 3 queuing ports.

These examples were run on ISS simulation and hardware with different purposes which will be explained in subsections 7.2 and 7.3, respectively. In both platforms the following acceptance criteria was followed:

- Functionality, i.e. if the application execution is coherent with its source code. A visual analysis based on punctual console prints were used for this;
- Structural integrity, i.e. architectural state of registers during the execution of ARINC services which can modify them;

- Temporality, i.e. state of timestamp and decrementer registers in key moments of the execution, such as the context switch.

Therefore, in order for an application, and thus an ARINC service group, to be considered validated and properly ported, it must pass the criteria previously mentioned.

The `arinc653-1part` example code and execution trace have been put in the subsection B.3 of the appendix to provide a better visualization of an user code and how the system behaves running it.

7.2 Simulation

As mentioned in section 2.2 an internally developed and maintained simulator is used called Kalray-1 Instruction Set Simulator. It can be invoked by `k1-mppa` or `k1-cluster`, according to where the binary file is supposed to be executed: IOC + CC or just CC, respectively.

The whole deployment command used to run TiCOS final ELF is:

```
k1-cluster -s libsyscall.so --march=k1b --cycle-based -- pok.elf
```

where `-s` specifies the syscalls library, `--march` specifies the MPPA® architecture version (in this case bostan) and `--cycle-based` which enables cycle accurate simulation. Two debug arguments can also be provided: `--profile` which generates an assembly execution trace to analyse and/or `-D` which waits for `k1-gdb` to attach.

The simulation approach has the advantage of being available on any machine that has the Kalray sotfware toolchain installed and eases the execution of I/O commands, as they do not perform the whole I/O MPPA® path, i.e. the normal execution flow is halted to require the host computer to execute the I/O command. This abstraction helps with debug and log messages while developing the system.

On the other hand, the ISS simulator does not contain some physical registers and even `mOS` virtualized registers and the timestamp register usually differs from the hardware execution. These aspects may cause runtime inconsistency and inability to perform the validation of some registers.

7.3 Hardware

Again as mentioned in section 2.2 the hardware execution can be deployed using a `x86` host and the Peripheral Component Interconnect (PCI) or in standalone mode through the Join Action Group (JTAG) connection. TiCOS is a standalone system and thus the command used is:

```
k1-jtag-runner --board=developer --exec-file=Cluster0:pok.elf
```

where `--board` selects the board type, in this case a developer machine, and `--exec-file` loads a specific file (`pok.elf`) on the given TAP (`Cluster0`). The option `-D` can be used to attach a `k1-gdb` and is the only hardware debug method available.

The hardware approach lets the developer have full control of the register set and completely validate an application in terms of timing. The downside of this method is the need to reduce the I/O communication as it affects heavily the execution of the program.

7.4 Final Product

The results of the execution of the applications previously presented are in the table below, organized by the ARINC service groups.

Table 1: Results of simulation and hardware validation

ARINC Service Group	Simulation	Hardware
ERROR	✓	✓
TIME	✓	✓
PARTITION	✓	✓
PROCESS	✓	✓
EVENT	✓	✓
BLACKBOARD	✗	✗
BUFFER	✗	✗
PORT	✓	✓

The table cells that contain the symbol ✓ show that the corresponding ARINC service group was validated through an application run on Time Composable Operating System (TiCOS), on the simulator or using the hardware (depending on the column), according to the acceptance criteria. The cells that contain the symbol ✗ show that the corresponding ARINC service group was not validated. Particularly, the *blackboard* and *buffers* groups, that supply intra partition communication, were not tested in any of the executed applications.

While executing the tests the urge for a better build process appeared. Before, if a user wanted to compile and run an example, it should go to the examples folder, choose the desired example, compile it with a make command and then execute it.

This ended up limiting the user and forcing him to tap the same command several times. Now, the root folder of the project has a makefile that builds all examples, puts them in a folder named `build-all` and generates a custom makefile to run them: `make simu` to run the example using the ISS or `make hard` to run the example using MPPA® hardware.

The final product is, therefore, the ported single-core TiCOS for the MPPA® with the majority of ARINC services validated, except for two low priority groups. In addition to that the build process was improved to ease and speed up the validation step of new examples.

7.5 Future work

The first stage of the work accomplished in the internship is a single core port, where TiCOS kernel and application partitions are running on CC0, PE0. Partitions and processes are allocated in the SMEM and virtually isolated. The next possible steps to accomplish are:

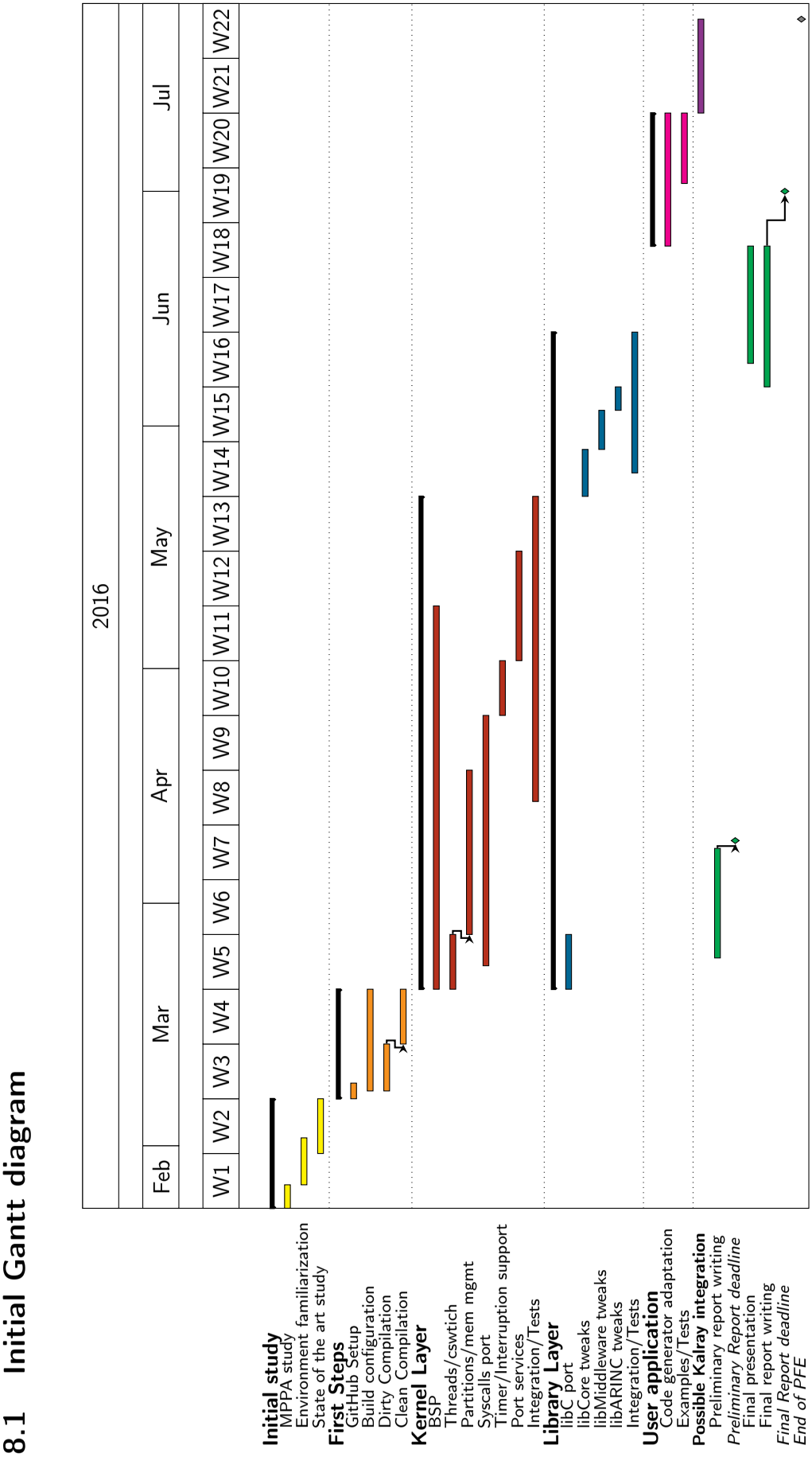
1. **MMU utilization:** use the hardware MMU to provide a proper isolation between the partitions and processes allocated in the SMEM. This is currently under development and should be integrated in the repository soon.

2. **Compute Cluster port:** TiCOS kernel running independently, possibly on PE0. The application partitions would run on distinct PEs of the same CC. This solution is still in study phase as it is unclear where the TiCOS kernel is supposed to run. Nonetheless, its development cost is not too elevated once the single core port is well done.

3. **MPPA® port:** TiCOS kernel running on IOC. TiCOS application partitions running on distinct PEs of distinct Clusters (spatially isolated). Inter-partition communication mechanisms using the NoC. This solution is still in study phase as it is even more unclear if the kernel should run on an IOC and the development cost to distribute the system on the chip and use the MPPA® NoC library is elevated.

8 Progression

8.1 Initial Gantt diagram



8.2 Forecast justification

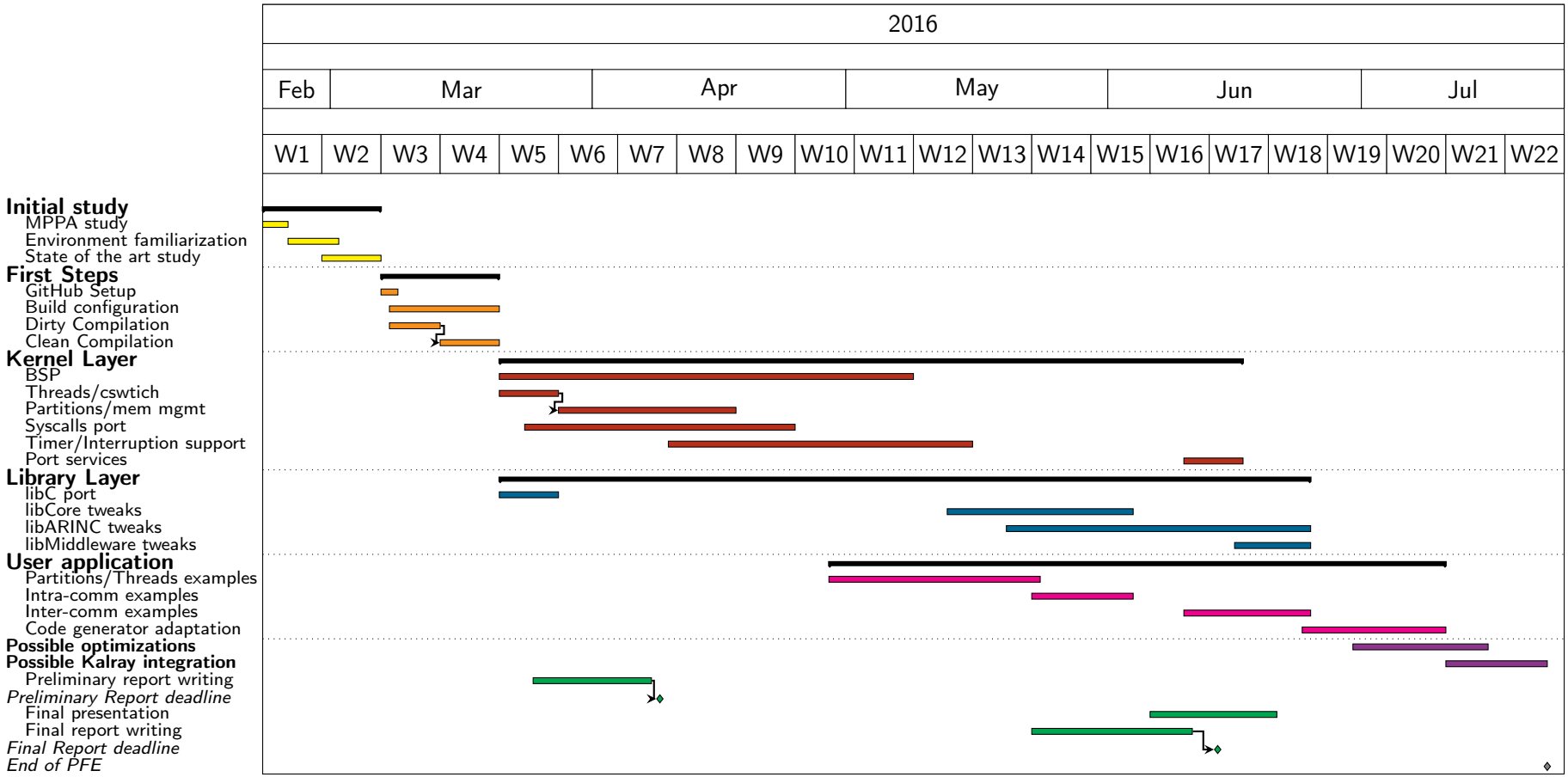
The first stage consists in understanding Kalray's MPPA® architecture, specifically the core, ISA and low-level software that has already been developed. After that, a period of familiarization with the work environment is planned, which includes the Kalray tools suite, development servers and internal git workflow. This stage ends with a state of the art study, analysing the ARINC standard and compliant OSs to define the porting choice and how it will be done on the MPPA®.

The second stage involves the GitHub setup and initial work. This will allow to verify the feasibility of the project and how the development of the next stages are supposed to happen. Firstly, the MPPA® tools need to be incorporated in the OS build configuration. Then two kinds of compilation will be done, aiming a bootable system with minimal functionality: dirty compilation, with all the MPPA® libraries included, and clean compilation with no standard library included, just the hypervisor mOS.

The next 3 stages are the actual OS port with all the tasks of each layer described in the diagram. The kernel layer being the heavier task, more time has been assigned to it. It is important to notice also that at the end of each stage a good amount of time is dedicated to the integration of all tasks and to general testing. There is also a possible Kalray integration where the internship final code can be pushed into Kalray's official repository.

To conclude, the final assignments are related to the report and presentation, having their particular deadlines which are to be observed.

8.3 Final Gantt diagram



8.4 Final planning analysis

This final gantt diagram reflects more realistically the work that has been accomplished. The preliminary diagram suggests that the kernel, library and user application development, integration and tests were independent and sequential.

After the first report deadline, this process was seen as unmaintainable and unreal due to the way the OS was compiled. Performing individual tests for each layer and their functionalities was very difficult because of their interdependency. A development with an incremental approach was then adopted, with the kernel and library layer work being more tied up to the type of application being executed in the system.

The Partition/Threads examples are relatively simple programs with single/multiple partitions that contains one or more threads scheduling themselves. It required the development of all kernel functions but the ARINC port services, as well as some libCore and libARINC tweaks. The intra-communication examples used the same structure but dealt with different functions from the library layer. The inter-communication examples required the development of the aforementioned port services and some adjustments in the libMiddleware.

Some time is still reserved before the end of the internship for possible optimizations, the integration process and adaptation of the code generator to include particular features of the MPPA® port.

9 Internship appraisal

The port of an operating system unveils a lot of requirements, difficulties and opportunities to learn. Some of them are the comprehension of external code (many times not well documented), deep knowledge in C and assembly (from different architectures) languages and understanding how an OS kernel works. Kalray has a private toolchain with their own version of GNU tools for compilation, disassembly and debug, demanding familiarity with their options and possibilities.

The concepts of a hypervisor and an exokernel, in this case mOS, are advanced operating systems mechanisms that forced the learning of details about the MPPA® architeture. The exokernel may hides or extends the complexity of the architecture and knowing what was virtualized, what was physical and its influence in the communication structure and system calls was vital for the porting process. In addition, the MPPA® is completely different from the typical Intel or MIPS architecture, normally seen and studied. The ISA, processor organization and deployment system have a quite slow learning curve, specially concerning undocumented assembly code.

As a final internship result, TiCOS was successfully ported to the MPPA®, except for some minor low priority ARINC services. It does have the limitation of still being a single core system, even though the processor was conceived to be parallel. Nevertheless, considering the available internship time, the single core port was already a great achievement. The future works on TiCOS parallelization are of sum importance to exploit all that the system and the MPPA® have to offer in terms of performance and safety.

10 Conclusion

The work presented here covered the TiCOS single core port to the MPPA®. Several areas were addressed in depth throughout this document: development of operating systems, parallel computing architecture, advanced C language concepts, a new ISA and the compilation flow of a well structured project.

The rigorous software specification, in this case ARINC 653 increased the cost and importance given to each coded line. As this program may be used for real avionic purposes, a responsibility facet, not always incorporated in regular projects, was present.

As already stated, the future work that focus on the MMU utilization and parallelization at cluster level which will then be extended for the whole MPPA® carries a lot of importance. This initial work created a functional basis that can now be extended to be safer and faster.

References

[1] K. H. Team, *The Kalray k1b VLIW Core Architecture and Reference Manual*. Kalray, 2011.

[2] J. D. Northcutt and E. M. Kuerner, *Network and Operating System Support for Digital Audio and Video: Second International Workshop Heidelberg, Germany, November 18–19 1991 Proceedings*, ch. System support for time-critical applications, pp. 242–254. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992.

[3] B. D. de Dinechin, *MPPA Bostan Processor for Time-Critical Applications*. Kalray, 2015.

[4] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, p. 966, 2009.

[5] "653-1 avionics application software standard interface." `http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=496`. Accessed: 2016-05-23.

[6] "653p4 avionics application software standard interface, part 4, subset services." `http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=1841`. Accessed: 2016-05-23.

[7] I. Microelectronics, "Developing powerpc embedded application binary interface (eabi) compliant programs," 2011.

A MPPA additional content

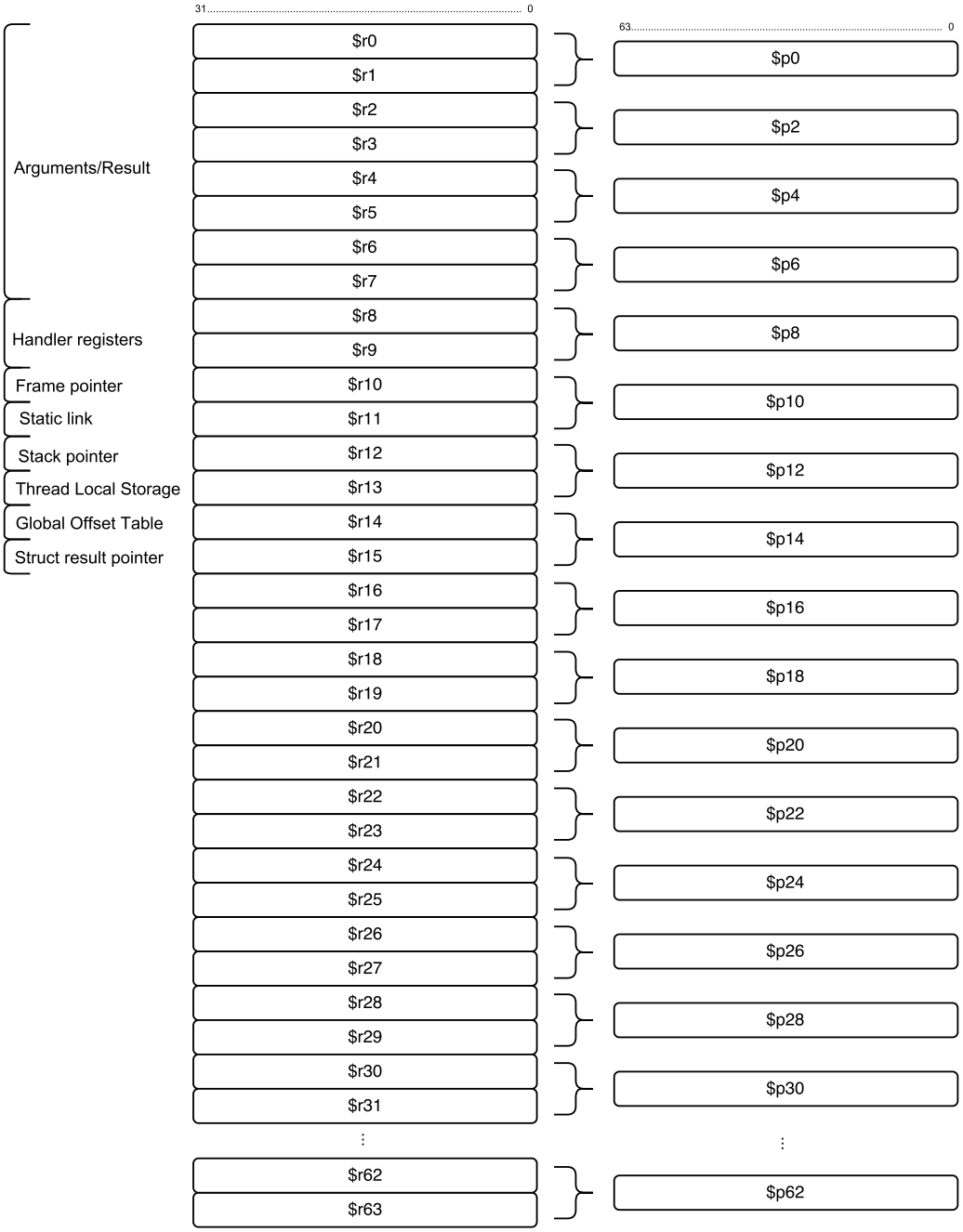


Figure 14: GPRs and GPR pairs with their respective usage convention

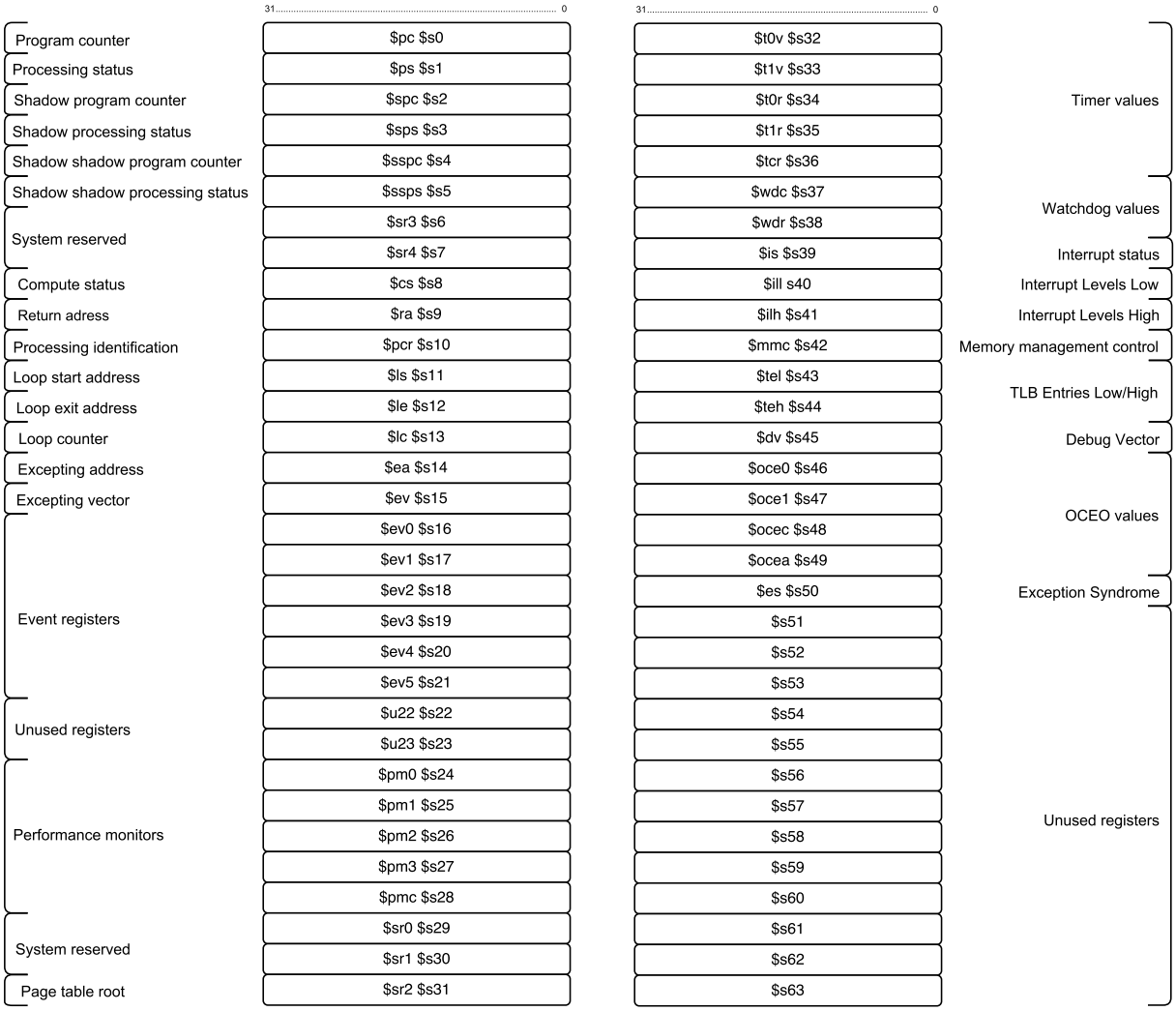


Figure 15: SFRs and their respective functions

B TiCOS additional content

B.1 Listings

The following listings present the handlers for system calls and interruptions, introduced in subsections 6.4.1 and 6.5.1, respectively.

Listing 7: MPPA® system call handler

```
.section .locked_text, "ax", @progbits
.align 8
.global _system_call_ISR
.proc _system_call_ISR
.type _system_call_ISR, @function
_system_call_ISR:
        add $r12 = $r12, -16                    # create scratch area
        ;;
        copy $r40, $r0                          # save registers $r0-$r7
        copy $r41, $r1                          # because _vk1_context64_save
        copy $r42, $r2                          # modifies them
        copy $r43, $r3
        ;;
        copy $r36, $r4
        copy $r37, $r5
        copy $r38, $r6
        copy $r39, $r7
        ;;
        make $r16 = pok_current_context         # retrieves current context
        ;;
        lw $r17 = 0[$r16]
        ;;
        _vk1_context64_save $r17                #saves it
        ;;
        make $r12 = _scoreboard_kstack_start;   # stack switch
        ;;
        copy $r0, $r40                          # restore registers $r0-$r7
        copy $r1, $r41
        copy $r2, $r42
        copy $r3, $r43
        ;;
        copy $r4, $r36
        copy $r5, $r37
        copy $r6, $r38
        copy $r7, $r39
        ;;
        add $r12 = $r12, -16                    # create nested scratch area
        call pok_arch_sc_int                    # jump to C code
        ;;
_return_from_sc:
        add $r12 = $r12, 16                     # destroy nested scratch area
        ;;
        make $r16 = pok_current_context
        ;;
        lw $r13 = 0[$r16]
        ;;
        sw 0[$r13] = $r0
        ;;
        _vk1_context64_restore $r13             # restore context
```

```
50          ;;
51          add $r12 = $r12, 16                     # destroy scratch area
52          ;;
53          scall MOS_VC_RFE
54          ;;
55  .endp _system_call_ISR
```

Listing 8: MPPA® interrupt handler

```
1   .section .locked_text, "ax", @progbits
2   .align 8
3   .global _interval_ISR
4   .proc _interval_ISR
5   _interval_ISR:
6           add     $r12, $r12, -16                 ## create scratch area
7           ;;
8           sd 0[$r12] = $p16;
9           make $r16 = pok_current_context         # load memory adress of
10          ;;                                      # current context
11          lw $r17 = 0[$r16]
12          ;;
13          _vk1_context64_save $r17                # save it
14          ;;
15          ld $p16 = 0[$r12]
16          make    $r0, _scoreboard_start
17          get     $r3, $pcr
18          ;;
19          extfz   $r1, $r3, 15, 11                # retrieve the current PE in $r1
20          ;;
21          sll     $r1, $r1, 8
22          ;;
23          add     $r2, $r2, $r1
24          ;;
25          lw      $r0 = MOS_VC_REG_PS[ $r2 ]      # retrieve the current $ps in $r0
26          copy    $r1, $r12                       # retrieve the stack pointer in $r1
27          ;;
28          srl     $r0 = $r0, 28
29          add     $r12, $r12, -16                 ## create nested scratch area
30          call    __k1_do_int                     ## jump to C code
31          ;;
32  _return_from_int:
33          add     $r12, $r12, 16                  ## destroy nested scratch area
34          ;;
35          make $r16 = pok_current_context         # retrieve the elected context
36          ;;
37          lw $r13 = 0[$r16]
38          ;;
39          _vk1_context64_restore $r13             # restore this new context
40          ;;
41          add $r12 = $r12, 16                     # destroy scratch area
42          ;;
43          scall   MOS_VC_RFE
44          ;;
45  .endp _interval_ISR
```

B.2 Diagrams

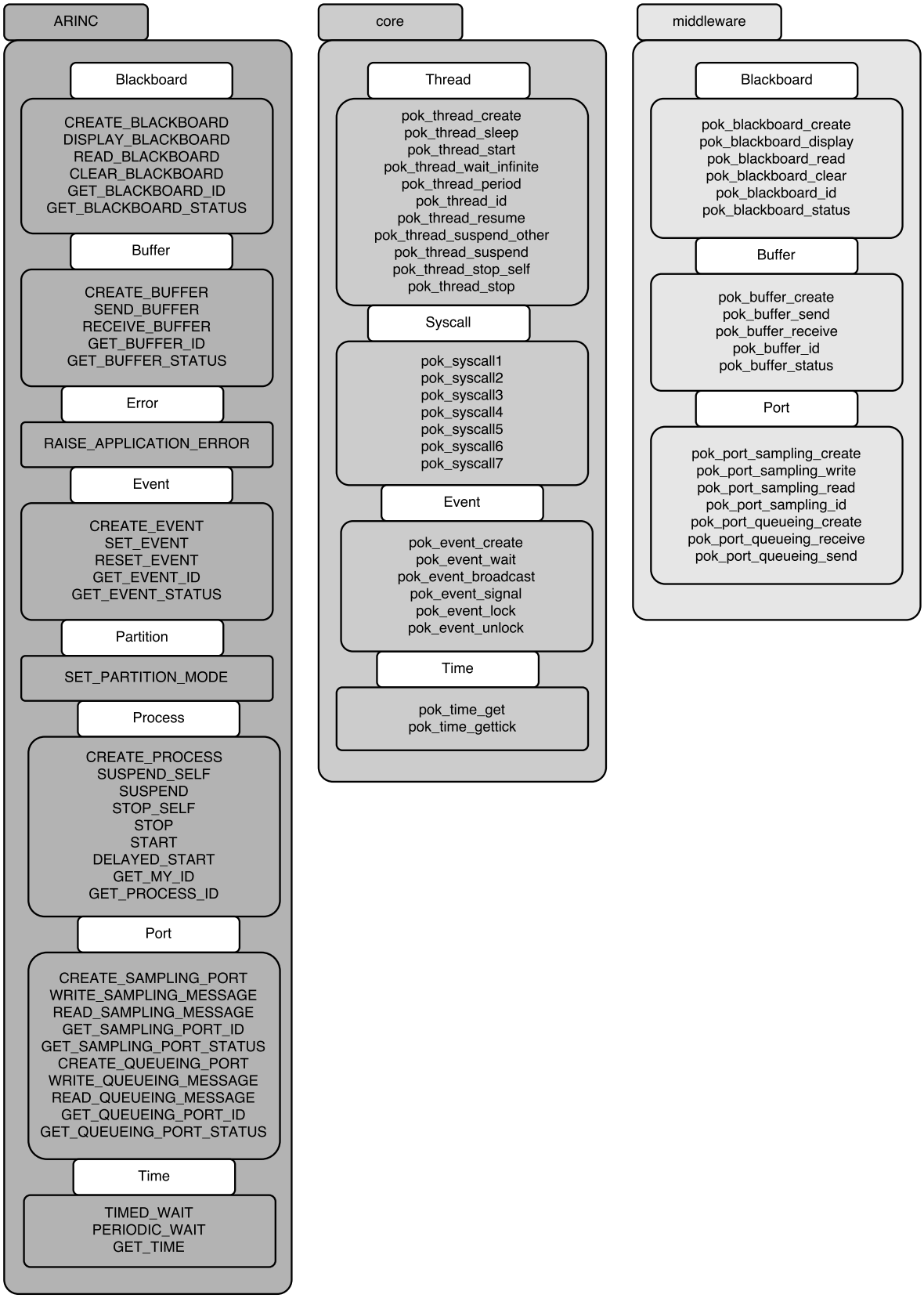


Figure 16: Structure of the ARINC 653 Part 1 library layer for user applications

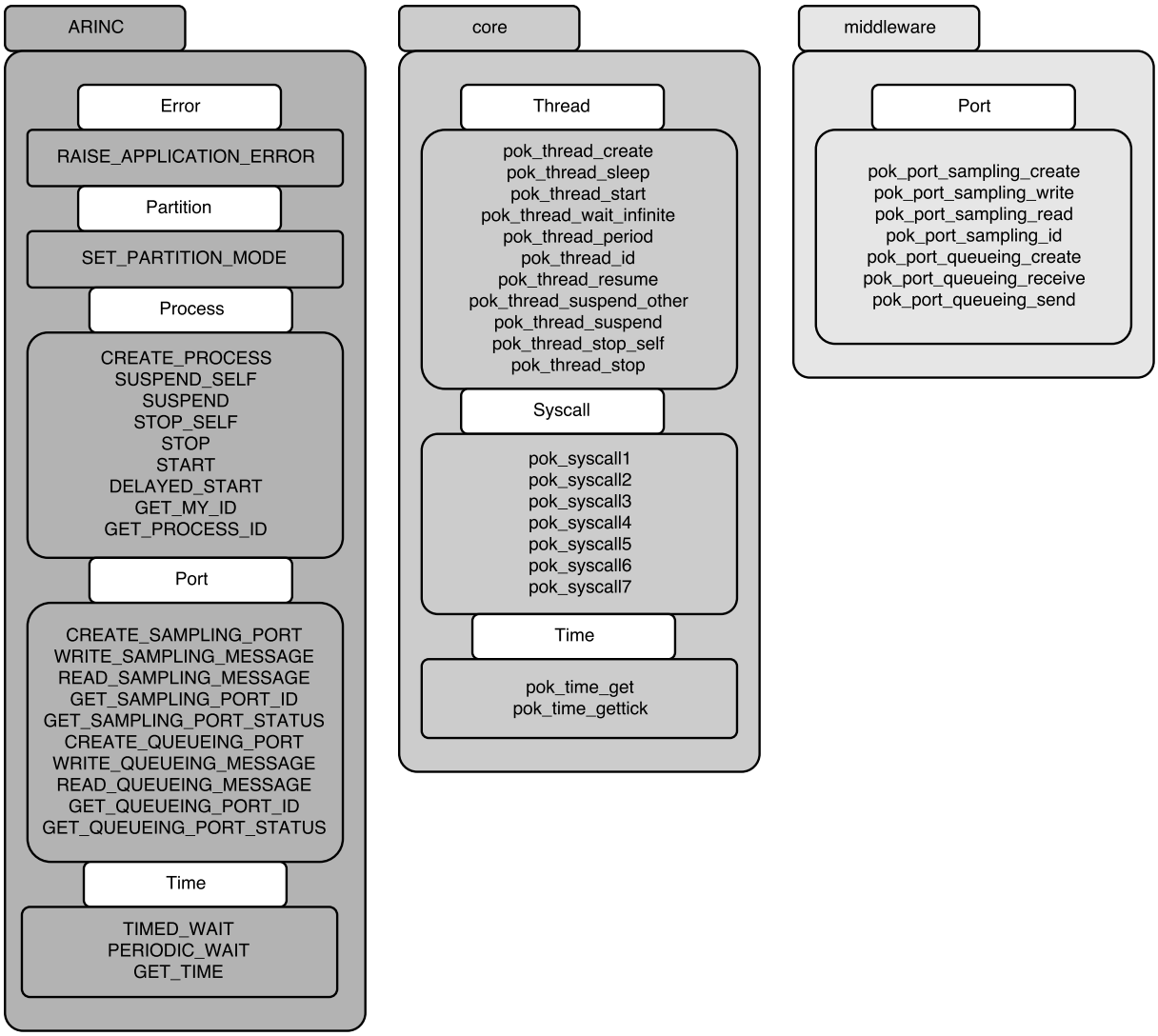


Figure 17: Structure of the ARINC 653 Part 4 library layer for user applications

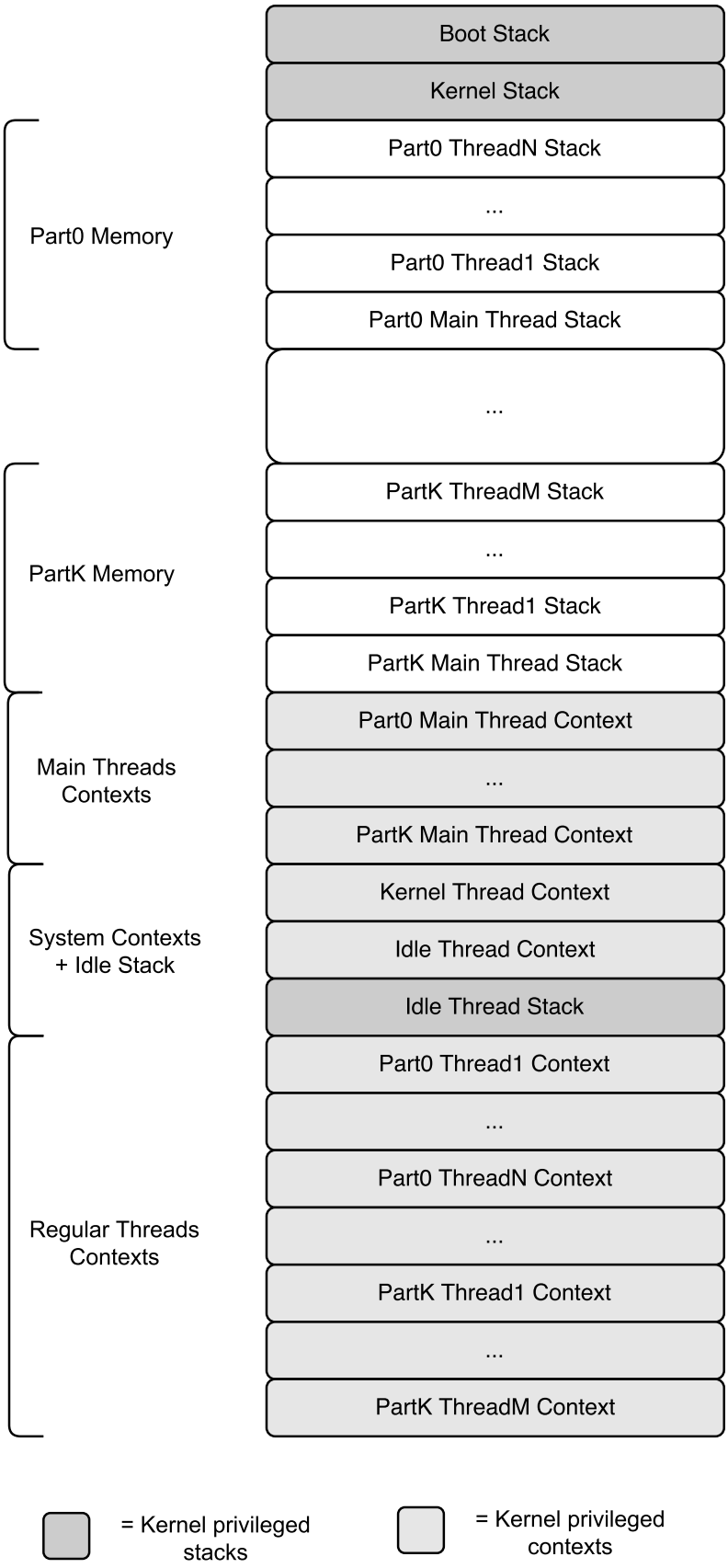


Figure 18: TiCOS memory map

B.3 Example arinc653-1part

The file `activity.c` describes the two threads that will run inside the `part1` scope.

Listing 9: part1/activity.c

```
#include <libc/stdio.h>
#include <arinc653/types.h>
#include <arinc653/time.h>
#include "deployment.h"

void* thr1_1_job ()
{
        RETURN_CODE_TYPE ret;
        while (1)
        {
                printf("Partition n. 1 - Thread n.1\n");

                /*****************************************************************/
                /* Message processing code should be placed here                 */
                /*****************************************************************/

                PERIODIC_WAIT (&(ret));
        }
}

void* thr1_2_job ()
{
        RETURN_CODE_TYPE ret;
        while (1)
        {
                printf("Partition n. 1 - Thread n.2\n");

                /*****************************************************************/
                /* Message processing code should be placed here                 */
                /*****************************************************************/

                PERIODIC_WAIT (&(ret));
        }
}
```

The file `main.c` configures the two threads, start them and then puts `part1` in normal mode, which makes the partition and processes runnable from the scheduler point-of-view.

Listing 10: part1/main.c

```
#include "activity.h"
#include <libc/stdio.h>
#include <arinc653/types.h>
#include <arinc653/process.h>
#include <arinc653/partition.h>
#include "deployment.h"

PROCESS_ID_TYPE arinc_threads[POK_CONFIG_NB_THREADS];

int main ()
{
```

```
12
13          PROCESS_ATTRIBUTE_TYPE tattr;
14          RETURN_CODE_TYPE ret;
15
16          printf("part1 - Main thread\n");
17          tattr.PERIOD = 100;
18          tattr.BASE_PRIORITY = 7;
19          tattr.ENTRY_POINT = thr1_1_job;
20          CREATE_PROCESS (&(tattr), &(arinc_threads[1]), &(ret));
21
22          tattr.PERIOD = 100;
23          tattr.BASE_PRIORITY = 9;
24          tattr.ENTRY_POINT = thr1_2_job;
25          CREATE_PROCESS (&(tattr), &(arinc_threads[2]), &(ret));
26
27
28          START (arinc_threads[1],&(ret));
29          START (arinc_threads[2],&(ret));
30
31          SET_PARTITION_MODE (NORMAL, &(ret));
32          return 0;
33  }
```

The execution trace that follows show the system from boot to user code. It is important to notice the interruptions arrival (`[DEBUG] DEC interrupt`), the main entry point (line 30) and the user threads being scheduled (lines 48,52; 59,62; etc) together with the idle thread (lines 44; 55; 66; etc). The program continues indefinitely as there is no call to exit.

Listing 11: arinc653-part1 example execution

```
1  [DEBUG] Pok arch init finished.
2  [DEBUG] MALLOC: Space reserved starting at: 0x00050000, with size: 40000
3  [DEBUG] Partition 0 loaded at addr phys=|50000|
4  pok_create_space: 0: 50000 40000
5  Loading partition code from 2c000 to 50000, for a size of : 1668
6  Copied last partial block from 2c000 to 50000 ( size = 1668 )
7  Loading partition code from 30000 to 70000, for a size of : 15c
8  Copied last partial block from 30000 to 70000 ( size = 15c )
9  Loading partition code from 34000 to 1f0000, for a size of : 0
10 Copied last partial block from 34000 to 1f0000 ( size = 0 )
11 [DEBUG] Setting up main thread of partition: 0, addr: 50000
12 [DEBUG] MALLOC: Space reserved starting at: 0x00090000, with size: 130
13 [DEBUG]  Creating context for partition 0, ctx: 0x00090000, entry: 50000
14 [DEBUG]  Creating context for partition 0, stack: 8fff0
15 [DEBUG] pok_thread 0 created in partition 0
16 [DEBUG] Partition 0: index_low 0, index_high 3
17 [DEBUG]  Creating KERNEL thread, id: 3
18 [DEBUG] MALLOC: Space reserved starting at: 0x00091000, with size: 130
19 [DEBUG]  Creating IDLE thread, id: 4
20 [DEBUG] MALLOC: Space reserved starting at: 0x00092000, with size: 1130
21 [DEBUG]  Context size 130
22 [DEBUG]  Creating system context 4, ctx: 0x00093000, sp: 92ff0
23 POK kernel initialized
24 [DEBUG]  TIMER_SETUP: Freq:74 MHZ, Div:37, Shift:0
25 TIME LAST: 3273, TIMER: 5000, TIME NEW: 8273, TIME CUR: 3273, DELTA: 5000
26 POK boot finished
27 [DEBUG]  DEC interrupt:1
```

```
Switch from partition 0 to partition 0
TIME LAST: 8273, TIMER: 50000, TIME NEW: 58273, TIME CUR: 8523, DELTA: 49750
        part1 - Main thread
[DEBUG] MALLOC: Space reserved starting at: 0x00094000, with size: 130
[DEBUG]  Creating context for partition 0, ctx: 0x00094000, entry: 500c0
[DEBUG]  Creating context for partition 0, stack: 8dff0
[DEBUG] pok_thread 1 created in partition 0
[DEBUG] Partition 0: index_low 0, index_high 3
[DEBUG] MALLOC: Space reserved starting at: 0x00095000, with size: 130
[DEBUG]  Creating context for partition 0, ctx: 0x00095000, entry: 500e8
[DEBUG]  Creating context for partition 0, stack: 8bff0
[DEBUG] pok_thread 2 created in partition 0
[DEBUG] Partition 0: index_low 0, index_high 3
SET_PARTITON_MODE : partition[0] MODE = NORMAL
DEBUG::Activation of thread 0 updated to 0
switch from thread 0, sp=0x90000
switch to thread 4, sp=0x93000 entry point = 0x1b388
[DEBUG]  DEC interrupt:11
Switch from partition 0 to partition 0
TIME LAST: 58273, TIMER: 50000, TIME NEW: 108273, TIME CUR: 58548, DELTA: 49725
Partition n. 1 - Thread n.1
DEBUG::Activation of thread 1 updated to 21
switch from thread 1, sp=0x94000
switch to thread 2, sp=0x95000 entry point = 0x500e8
Partition n. 1 - Thread n.2
DEBUG::Activation of thread 2 updated to 21
switch from thread 2, sp=0x95000
switch to thread 4, sp=0x93000 entry point = 0x1b388
[DEBUG]  DEC interrupt:21
Switch from partition 0 to partition 0
TIME LAST: 108273, TIMER: 50000, TIME NEW: 158273, TIME CUR: 108355, DELTA: 49918
Partition n. 1 - Thread n.1
DEBUG::Activation of thread 1 updated to 31
switch from thread 1, sp=0x94000
switch to thread 2, sp=0x95000 entry point = 0x500e8
Partition n. 1 - Thread n.2
DEBUG::Activation of thread 2 updated to 31
switch from thread 2, sp=0x95000
switch to thread 4, sp=0x93000 entry point = 0x1b388
[DEBUG]  DEC interrupt:31
Switch from partition 0 to partition 0
TIME LAST: 158273, TIMER: 50000, TIME NEW: 208273, TIME CUR: 158355, DELTA: 49918
Partition n. 1 - Thread n.1
DEBUG::Activation of thread 1 updated to 41
switch from thread 1, sp=0x94000
switch to thread 2, sp=0x95000 entry point = 0x500e8
Partition n. 1 - Thread n.2
DEBUG::Activation of thread 2 updated to 41
switch from thread 2, sp=0x95000
switch to thread 4, sp=0x93000 entry point = 0x1b388
```