A Time-Composable Operating System for the Patmos Processor

Marco Ziccardi

Kongens Lyngby 2013
M.Sc.-2013-73

Technical University of Denmark
Applied Mathematics and Computer Science
Matematiktorvet, building 303B, DK-2800 Kongens Lyngby, Denmark
Phone +45 4525 3031, Fax +45 4588 1399
compute@compute.dtu.dk
www.compute.dtu.dk M.Sc.-2013-73

Summary

The aim of this thesis is to port the TiCOS operating system to the Patmos processor. TiCOS is a light-weight operating system developed to obtain composability and analyzability and targeting single-processors. Patmos is a time-predictable single-processor developed in the framework of the T-CREST project. The aim of the T-CREST project is to develop a time-predictable multi-processor able to meet both the requirements of safety and processing capacity for modern real-time and embedded systems.

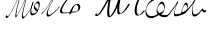
Preface

This thesis was carried out at the department of Applied Mathematics and Computer Science, at the Technical University of Denmark, in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis was developed in the framework of the Time-Predictable Multi-Core Architecture for Embedded Systems (T-CREST) project, a Specific Targeted Research Project (STREP) of the European Union's 7th Framework Programme, whose goal is the creation of a time-predictable multi-core architecture for embedded systems. In order to do so the T-CREST group is developing, amongst other things, the processor, the Network on Chip (NoC) and the compiler. The T-CREST project is a collaboration of many industrial and research organizations and the architecture, as a research project, is continuously evolving. The research and dynamic nature of T-CREST architecture made of my master thesis project a challenging and meaningful experience. On the other hand my thesis aided in identifying industrial requirements that the architecture was missing, which caused me to further my understanding of the T-CREST architecture itself, extending it with the needed features.

The thesis consists of the adaptation of an existing time-composable operating system for the T-CREST processor (Patmos). In order to make this possible the need for extensions to the processor architecture was identified and the relevant features implemented in the software simulator.

Lyngby, 31-July-2013



Marco Ziccardi

Acknowledgements

I would like to thank my supervisor Martin Schoeberl for driving my project work in the right direction. I would also like to thank all the T-CREST group at the Technical University of Denmark and Stefan Hepp from the Vienna University of Technology, special thanks go to Florian Brandner for helping me every time I needed. Finally, I would like to thank Andrea Baldovin, Enrico Mezzetti and Tullio Vardanega for the endless amount of valuable advices.

Contents

Chapter 1

Introduction

In this chapter we introduce some key aspects of Real-Time Systems (RTS), starting from the computational model. We then discuss *timing analysis* and Worst-Case Execution Time (WCET) calculation. After that we introduce *time-composability* and how important that property is. In Section 1.4 the role of a Real-Time Operating System (RTOS) is described and some examples are presented. In the end the T-CREST project and platform are briefly described.

1.1 Real-time Systems

A real-time system receives inputs and sends outputs to the hardware, working under strict constraints on its response time [Ben06]. Real-time systems (RTS) are almost everywhere in our everyday life, applications of real-time systems can be found in [Mal09]:

- *Industrial applications*: key industrial systems often require a RTS, for example process control and automation are usually RTS-managed
- *Medical*: a RTS is hidden behind almost all medical diagnostic machines

- *Peripherals*: devices like laser printers or digital cameras contain embedded real-time systems
- *Automotive and Transportation*: automotive assistant systems such as automated pedestrian detection are implemented through real-time systems. Likewise the safety system of a train is a RTS
- *Telecommunication applications*: real-time systems are used in modern cellular systems
- *Aerospace*: real-time systems in aerospace industry have to deal with the big amount of fresh data collected from the environment and the scarce downlink bandwidth
- *Avionics*: several systems on an aircraft perform real-time operations and have to be coordinated. Examples are the communication and the navigation systems
- *Defence*

A simplified model for a RTS is shown in figure 1.1. A **sensor** has to convert environmental inputs to electrical signals while an **actuator** is a device that takes an electric signal and converts it to physical action on the environment. **Conditioning units** are responsible for translating the electric signal coming from a sensor to an electric signal that could be used by a computer and likewise they translate computer electric signal to something understandable by an actuator. An **interface unit** translates signals between the two talking ends (CPU and conditioning unit) and eventually takes care of buffering [Mal09].

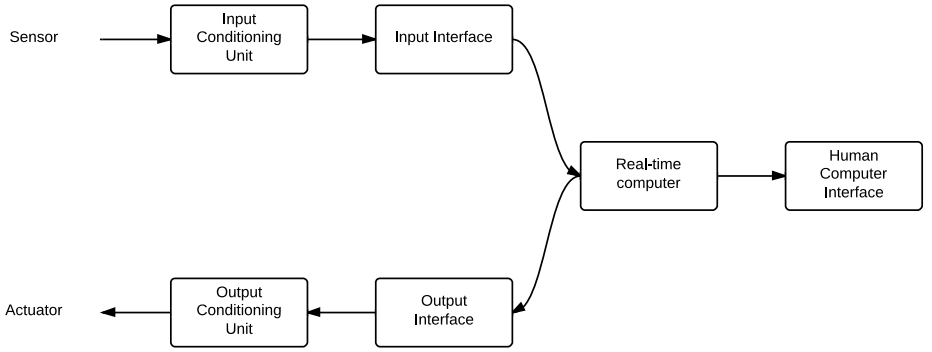


Figure 1.1: A real-time system model (from [Mal09])

1.1.1 Task Models

A real-time application can also be viewed as a *taskset* τ, a static set of tasks [DB11]. Each real-time task is characterized by a set of attributes specifying its timing behaviour [Ben06]:

- *Release time*: time when the task is ready to be executed
- *Worst case execution time* C_i
- *Response time* R_i: amount of time between the task release time and the end of its execution
- *Deadline* D_i: maximum allowed response time

In the literature two simple task models are mainly used [DB11]:

- *Periodic task model* [LL73]: tasks perform an ideally infinite sequence of invocations (*jobs*). Jobs arrival time is periodical: between two invocations always occurs the same amount of time, we call this amount period T_i
- *Sporadic task model* [Mok83]: each task's invocation may occur at any time once a fixed interval T_i since the last invocation passed

In the above task models intra-task parallelism is not allowed: only a single job of a task can be active at a time.
T_i indicates the period in the periodic model and the minimum inter-arrival time in the sporadic one. There are three different types of deadlines D_i with respect to the value of T_i:

- *Implicit deadlines*: deadlines correspond to the periods, $D_i = T_i$
- *Constrained deadlines*: deadlines are less than or equal to the periods $D_i \leq T_i$
- *Arbitrary deadlines*: deadlines can be less than equal to or greater than the periods

Tasks and therefore real-time systems can be classified according to the consequences of a deadline miss [Mal09] into:

- *hard*: the task has to produce a result within its deadline otherwise the whole system is considered to have failed
- *firm*: the task is required to produce a result within its deadline. Eventual late results are discarded
- *soft*: the task has an associated deadline which, however, is not absolute but expressed as an average response time required

The developed real-time operating system is targeting hard real-time systems.

1.2 Timing Analysis

The maximum execution time is a key attribute of a task and it is commonly called Worst Case Execution Time (WCET) [WEE+08]. The calculation of the WCET is crucial in the development and validation of a real-time systems. *Timing analysis* is the process of computing execution time bounds [WEE+08], which not necessarily correspond to the `WCET`, as shown in figure 1.2.

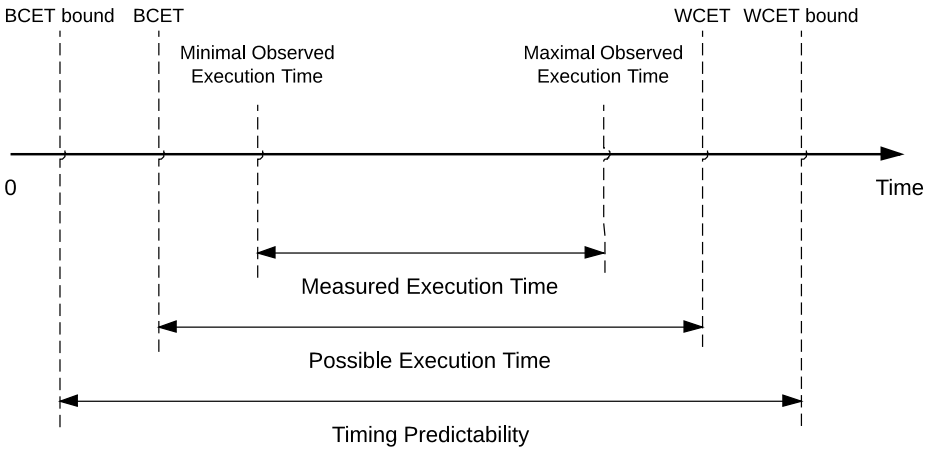


Figure 1.2: Relationship between measured execution times, WCET and WCET bound

Timing analysis has to face several problems when computing the WCET for a task. It has to consider each path of the control flow through the task and its time to be executed on the specified hardware. The computation of the execution time of a path is harder when the underlying hardware makes use of

history-dependent components such as caches.

1.3 Time composability

Complex systems are often built integrating independent components. A composable architecture assures that the components can be integrated together without changing their behaviour [KO02]. In real-time systems we refer to *time-composability* as the property of being composable in the time domain; that is, the timing characteristics of a component do not change upon composition of that component in a larger system.
Modern real-time systems development relies on the assumption that the timing behaviour of the whole system can be obtained by composing the WCET of its components [BMV13]. When developing a system, assuming composability has an innate characteristic is not always correct: several issues arise from modern hardware which affects not only single component's timing behaviour but even time-composability. As a non-time-composable hardware feature we can think about the use a cache when two functions are executed: each function's timing behaviour is probably not going to be preserved [LRL10]; the same reasoning can be made for bigger components such as tasks. Moreover, not only hardware can interfere with time-composability but even the execution of the operating system's services has an influence on the execution of application's tasks. Using a time-composable operating system is highly desirable when developing systems with strict timing-behaviour constraints [BMV13].

1.4 Real-time Operating Systems

An Operating System (OS) is a software working as an interface between computer hardware and application programs, controlling the execution of other applications [Sta08]. An OS tries to hide hardware complexity from the programmer and in order to do so it provides several services [Sta08], like:

Executing a program. That is, performing several operation like loading program's code or initializing hardware

Accessing I/O devices. Each hardware resource may need dedicated instructions, the OS tries to provide a uniform interface avoiding the programmer from dealing with these details (*drivers*)

Providing core functionalities. In order to avoid each programmer from implementing them, assuring that these functionalities are correctly implemented

Controlling accesses to the system

Controlling accesses to data

Detecting errors and handling them

An OS is usually developed relying on a hardware interface called Instruction Set Architecture (ISA), which is the set of instructions the underlying hardware is able to execute. The OS uses these instructions (some of them are reserved to the OS itself and not accessible by user programs) to provide its services. Likewise the operating systems offers two other interfaces:

- *Application Binary Interface (ABI)*: the ABI is the system call interface to the OS and it defines the rules for applications portability
- *Application Programming Interface (API)*: the API offers higher level functions to perform system calls. Developing applications around an API ensures their portability to systems supporting the same API

A computer system is a set of resources (like CPU and memory), therefore the OS can be seen as a program developed to manage these resources. An OS is usually made of several components. The key part of an operating system is called *kernel*. The kernel contains the most used services of the OS and is permanently placed in memory during system's execution. An OS uses an abstract representation of a computer program, usually called process. The role of the OS as a resource manager is to assign resources to processes so that they can execute. In order to correctly manage resources, operating systems implement several features:

- *Interrupts handling*: interrupts provide the operating system a way to react to external events. Interrupts are used to manage I/O devices. An interrupt makes the computer suspend the current execution and jump to an operating system's routine called Interrupt Service Routine (ISR) responsible for handling the exception associated to the interrupt
- *Memory management*: each process needs some memory for its execution. The OS has to grant each process an adequate amount of memory and has to grant also that every process accesses this memory correctly: for example a process should not access other processes memory. *Memory protection* allows the OS kernel to avoid any illegitimate access

- *Multitasking*: on modern computers several programs are executed at the same time. The OS has to grant each program to proceed in its execution arbitrating a possibly single CPU. A component of the kernel is called *scheduler* and is responsible for allocating the CPU to processes. In order to perform this allocation, the OS must be able to take control over a running process. This may happen after a process explicit request or due to an external event (interrupt). When gaining control of the CPU the OS has to save the current execution state of the process in order to restore it later (we call this execution state *context).* After having saved the context the scheduler has to select the next process to execute, whose context is going to be restored. The just explained process is called *context switch* and is a key feature implemented by an OS

A real-time system consists of one or more applications, each containing one or more communicating tasks. In order to manage the complexity of such systems a Real-Time Operating System (RTOS) is used. As a normal OS, a RTOS has to handle resource arbitration, and more precisely: time sharing, (virtual) memory allocation and inter-task communication. However, the real-time nature of the OS imposes more requirements. First of all a RTOS has to consider time as a key parameter [Tan07]. More features of a RTOS are listed in [SBWT87]:

- A RTOS must support different types of task, from small and high rate tasks to larger and less frequently executed ones
- A RTOS should not rely on a simplified task model and therefore it should be able to handle both periodic and sporadic tasks
- Inter-task communication is time-critical and different tasks may assume different models of communication
- Different systems may have different requirements on the operating system. Some systems may need full RTOS functionalities such as complex scheduling policies or space partitioning while others may need a more lightweight OS, so an RTOS should be configurable according to the system's needs

1.4.1 Memory Management

As previously stated memory management and inter-task communication are key features offered by a RTOS. Both memory allocation and inter-task communication mechanisms implemented by a RTOS have to assure that tasks timing

constraints are not going to be violated due to other task's interference. More precisely, in the case of memory allocation a good and widely adopted way of guaranteeing no interference between tasks is not allowing shared memory using tasks space partitioning, which can be implemented through memory virtualization or memory protection. Moreover a RTOS has to provide to tasks ways of communicating between each other without violating deadlines constraints, not allowing shared memory means that communication services have to be implemented by the RTOS.

1.4.2 Scheduling Algorithms

Several scheduling algorithms have been developed to share a single-processor between tasks. Scheduling algorithms can be classified in different ways. Scheduling may be:

Static: based on offline information

Dynamic: based on run-time information

A scheduling algorithm can also be classified as:

Preemptive: a task computes for a certain amount of time then the control comes back to the operating system which selects an other task to be assigned the processor

Non-preemptive: a task executes until completion or until it yields control to another task

We call a scheduler *work-conserving* if it never lets the processor idle when there is a task ready to run.
Several algorithms have been developed for scheduling tasks on a single-processor, both static and dynamic. *Rate-monotonic* [LL73] is a static priority algorithm assigning each task a priority proportional to its *request rate* (defined as the reciprocal of its period). On the other hand Earliest-Deadline-First (EDF), presented in [LL73], is a dynamic priority scheduling algorithm. EDF assigns the highest priority to the task whose current request has the nearest deadline. EDF is proved to be optimal for tasksets with implicit deadlines; that is, it reaches a processor utilization factor of 1.

Scheduling on multi-processors systems attempts to solve two problems [DB11]: *allocation problem* (decide to which processor allocating a task) and *priority problem* (decide in which order the jobs of all tasks should execute). In the last years two main approaches to multi-processor scheduling have been investigated: *partitioning* and *global scheduling* [CFH+04]. With partitioned scheduling each task is permanently assigned to a processor and then each processor is scheduled independently. On the other hand, global scheduling enqueues all the tasks in a priority queue from which highest priority tasks are then selected for the execution. Unfortunately, optimal solutions found for the scheduling problem on single-processors like EDF result in non-optimal schedules with very low processor utilization [DL78]. Partitioning approaches reduce the problem of multi-processor scheduling to a set of single-processor ones.

1.4.2.1 Partitioning

A well known approach to the scheduling in systems where multiple applications share one processor is the partitioned one: static non-work-conserving schedulers, such as time division multiplexing, assign applications to the processor according to a pre-defined offline order. Since scheduling decisions do not rely on the run-time behaviour, such schedulers provide time isolation (time-composability between tasks or *partitioning*): the actual behavior of an application is independent of others.
It is also desirable to have different applications follow different computational models. In order to do so a two-level scheduling is required: *inter-application* and *intra-application*. Preemptive *intra-application* schedulers promise to reach better processor's utilization than non-preemptive approaches but at the cost of more overhead.
ARINC653 [Gro03] is a standard specification for time and space partitioning in safety-critical avionics real-time operating systems. According to ARINC, time is divided in a number of slots called partitions, each of which is assigned an application. In each partition a second-level scheduler can be used to allocate the CPU to the partition's tasks, see figure 1.3 for a sample of architecture for an ARINC653 RTOS [Inc08].
Just a few real-time operating systems implement partitioning with two-level scheduling, *inter-application* and *intra-application*. CompOSe [HEM+11], PikeOS [Pik], INTEGRITY [INT] and LynxOS-178 [Lyn] implement both partitioning and two-level scheduling.

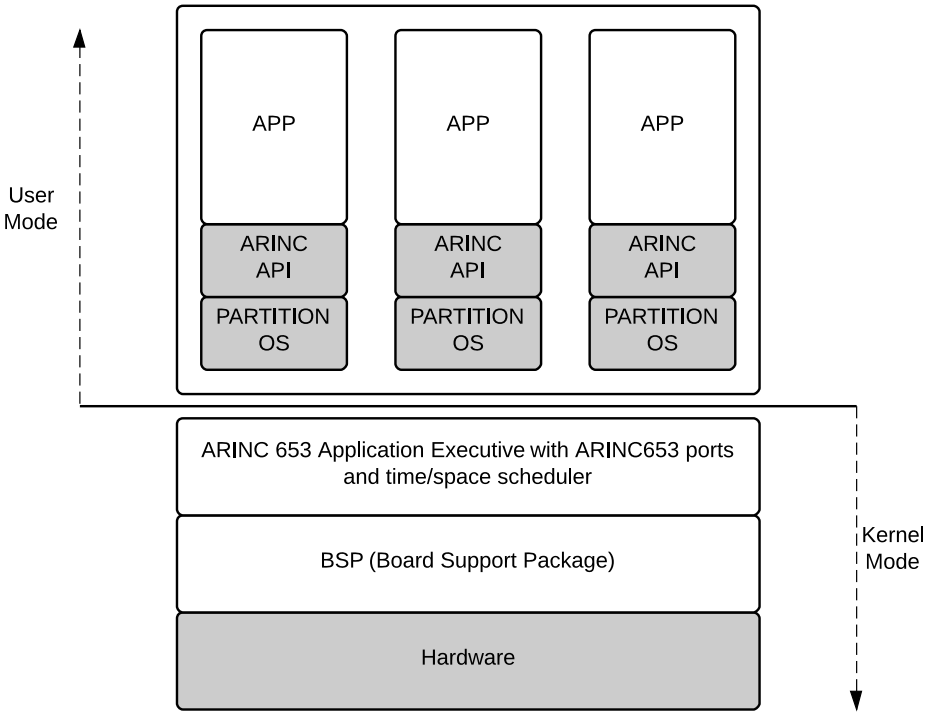


Figure 1.3: Architecture's structure of an ARINC653 system (from [Inc08])

1.4.3 PikeOS

PikeOS [Pik] is a commercial micro-kernel RTOS targeting safety-critical embedded real-time systems and supporting the ARINC653 standard. PikeOS is able to run different applications, whether real-time or not, in different virtual machines on a single underlying hardware. For real-time applications PikeOS is able to guarantee spacial and temporal constraints.

PikeOS executes applications with different timing requirements: hard real-time, soft real-time and non real-time. In order to support all these kinds of applications PikeOS implements both a priority-driven and time-driven scheduler. Such a scheduler allows to re-allocate computing times not used by hard real-time tasks. In figure 1.4 the difference between PikeOS scheduler and a normal RTOS scheduler is shown.

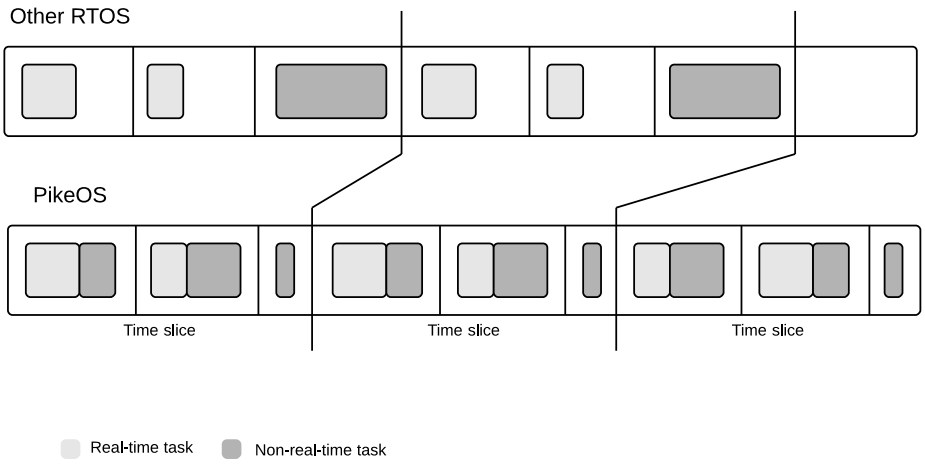


Figure 1.4: Comparison between PikeOS and a normal RTOS scheduler (from [Pik])

A time slice is assigned to each virtual machine, each virtual machine receives a fixed amount of time and is able to schedule real-time tasks itself. As previously stated, PikeOS scheduler is able to exploit the *slack time* (not used computing time) and allocate it to non real-time tasks. We can see in figure 1.4 how it reduces the time exclusively allocated to these tasks, with respect to a standard scheduler.

1.4.4 INTEGRITY

INTEGRITY [INT] is a partitioning RTOS aiming to provide reliability, security, and maximum performance to embedded systems. INTEGRITY uses memory protection to separate each application execution from other applications. Partitioning ensures that a task is granted enough resources to execute correctly and does not affect other application's and OS execution.
INTEGRITY exploits the Memory-Management Unit (MMU) hardware and therefore offers memory protection without sacrificing performances. INTEGRITY is implemented to react to interrupts as fast as possible and in order to do so it never masks or disables interrupts.

One of INTEGRITY's goals is a safe handling of memory: in order to always guarantee enough memory to the kernel, kernel's objects are never placed in ker-

nel's memory. RTOS services always use resources owned by the calling task. Moreover these services are executed on a dedicated kernel's stack in order to prevent stack overflow and to allow user's tasks to precisely specify their stack size.

1.4.5 LynxOS-178

LynxOS-178 [Lyn] is a commercial hard real-time operating system that supports both POSIX and ARINC653 standards. LynxOS-178 targets safety-critical real-time systems and guarantees time and space-partitioning relying on virtual machines, which lead system events happening in a partition not to interfere with events in another. Each RTOS partition behaves just like a single, independent RTOS. LynxOS-178's partitioning grants three types of exclusive access: time, space and resources.
Time is divided according to ARINC653 specification in fixed-size time slices, each of which is assigned to a partition.
Memory partition is accomplished by dividing it in blocks of different address spaces. Each partition is assigned one and only one block.
Partitioning of resources means that every resource is associated to one and only one partition at any time so that a fault can be handled in a single partition, without affecting the others. For example each partition uses a RAM-based file system which is private to the partition and never shared.

LynxOS-178 structure is depicted in figure 1.5. The *CPU support package* contains all processor family-specific routines.The *Board support package* contains all the routines needed for booting and controlling the hardware. The *Static device drivers* are components isolating specific hardware details from application code and are statically compiled with the kernel. *LynxOS-178 kernel* offers partitioning functionalities.

1.4.6 CompOS

CompOSe [HEM+11] is a composable, light weight real-time operating system targeting Multi-Processor Systems on Chip (MPSoC). Each processor executes an independent instance of CompOSe which has no idea of the existence of the other processors, therefore each scheduler performs local decisions. CompOSe scheduler's aim is to ensure that every task executes without any interference, in order to do so some requirements need to be fulfilled:

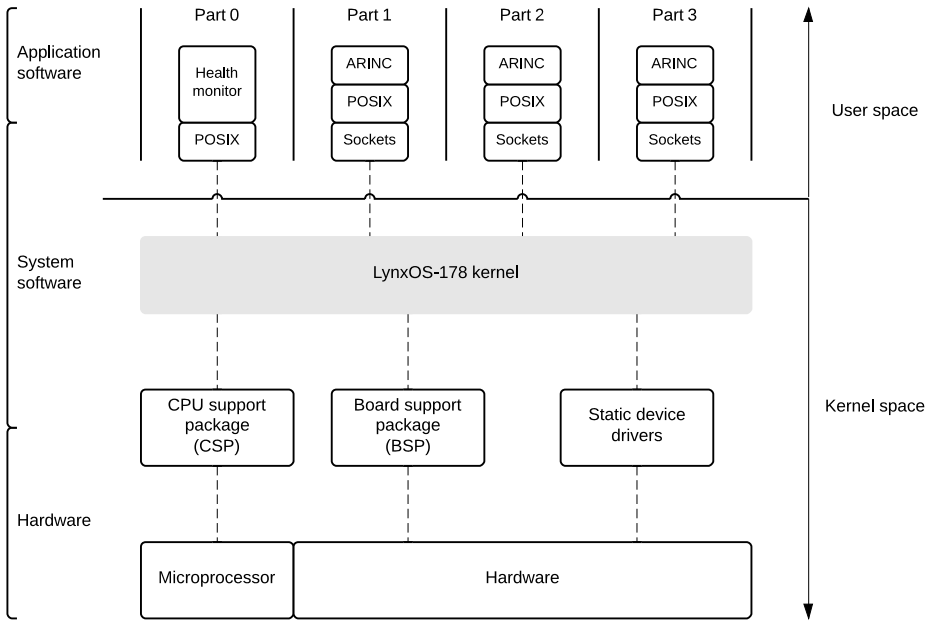


Figure 1.5: Architecture's structure of LynxOS-178 RTOS (from [Lyn])

- Preemption-based CPU sharing, so that system correctness does not rely on tasks well-behaviour
- Context-switch mechanism not interfering with user tasks; that is, running in constant time
- Scheduling between applications with adequate cache management

CompOSe implements a two-level scheduler in order to have different intra-application schedulers. Moreover a slack manager allows to exploit all unused capacity.

CompOSe uses C-HEAP [NKG+02] as a communication API. C-HEAP does not use atomic operations (therefore no need for locks and semaphores) or interrupts, all communication takes place on a buffer through explicit calls to *acquire* and *release*. A C-HEAP FIFO is a circular buffer placed in shared memory. FIFO read and write pointers are hold in producer and consumer local memory in order not to make a task suffer from remote access latency. If there is enough space in the task's local memory even FIFO's communication buffers are placed there

to avoid accessing remote memory. When communication buffers are placed in the local memory, before starting the execution of a task data has to be copied from remote locations to local buffers.

1.4.7 TiCOS

Time-Composable Operating System (TiCOS) [BMV12] is an open-source time-composable real-time operating system implementing a two-level partitioned scheduler and conforming to the ARINC653 standard. TiCOS is based on POK [DL11] a light weight ARINC653 operating system released under the BSD license. Details of TiCOS operating system can be found in Chapter 3.

1.5 The T-CREST Project

Real-time embedded systems are everywhere in our daily life. Examples can be found in the most different fields like space systems, avionics, automotive and consumer electronics. Those fields are constantly evolving and improving, so real-time embedded systems should improve too [DB11]. Real-time embedded systems evolution takes place on two different sides, one regards systems analyzability while the other regards system performances.

The modern technological progress causes software complexity to grow and along with this complexity growth even processing demand is increasing [DB11]. This increase of processing demand made hardware producers try to create more powerful processors. The first trial was miniaturization, following the Moore's law, which says that every two years the density of transistors on an integrated circuit doubles (that is: every two years processor speed doubles). But this was no longer possible due to problems of heat dissipation and high power consumption. The second and still used trial was putting more than one processor on a single chip: this gave birth to multi-processors.

Even for real-time embedded systems there is the need for more powerful processors and thus a big effort has been spent in the last years finding ways of creating time-predictable real-time systems on modern multi-processors. Time-predictability is a key feature of safety-critical systems (systems whose failure can result in the loss of life). In such systems the worst case execution time has to be known in order to assure a certain behaviour (response time) when

critical events happen. However, multi-processors vendors focus their attention on end-user needs and so they try to give best performances in the average case, leading to bad results in the worst case execution and making those processors hardly analyzable [DB11].
The aim of T-CREST is to create a time-predictable chip multi-processor, developing the processor, memories and the interconnect optimized for minimizing the WCET. Moreover, through time-predictable caches the system aims to fulfill the increased needs of processing power. Patmos [SSP+11] is the single-processor used to build T-CREST architecture. The final goal of T-CREST is creating an architecture capable of high performances but still easily analyzable.

1.6 Thesis Structure

The thesis is structured as follows:

1. First we present Patmos and TiCOS:
   - *Chapter 2: The Patmos processor*: a detailed description of the processor's instruction set and memories
   - *Chapter 3: The Operating System*: a description of the OS kernel's architecture and library
2. In order enable on the T-CREST architecture a computational model more complex than the direct mapping of threads to processors; that is: executing an operating system on each Patmos core, some extensions to the processor itself were needed:
   - *Chapter 4: Processor extensions*: presents a set of extensions performed to the Patmos processor and a set of useful possible extensions
3. Finally we present the extensions made to TiCOS to port it to Patmos:
   - *Chapter 5: TiCOS extensions*: details the main changes performed to TiCOS starting from the architectural-dependent layer up to the application library

Some details about Patmos simulator and ELF file format can be found in *Appendix A: Software Simulator of Patmos* and *Appendix B: ELF file structure.*

Chapter 2

The Patmos Processor

Patmos [SSP+11] is a time-predictable, reduced instruction set (RISC) processor targeting real-time systems aiming to reduce the complexity of WCET analysis. The Patmos pipeline is made of 5 stages:

- `FE`: instruction fetch
- `DEC`: instruction decode
- `EX`: execute
- `MEM`: memory access
- `WB`: register write back

In the following, Patmos is described. The aim is to clearly define the architecture, highlighting the main points in its technical report [SBH+] which influence the development of an operating system.

2.1 Memory

Patmos uses several local memories in order to reduce memory operations latency. Four types of local memories are used:

- Method cache
- Stack cache
- Data cache
- Scratchpad memory

Memory access instructions which deal only with local memory (that is: scratchpad or stack cache) are guaranteed to never stall the pipeline while other memory instructions (accessing data cache or global memory) define opcodes that either stall the pipeline or perform decoupled operations.

2.1.1 Method Cache

The Patmos processor implements the idea of a method cache [Sch04]. With today's performance requirements caches cannot be avoided due to their important role in filling the gap between processors speed and memory access delays, however caches focus on the average case performance, making the system hardly analyzable from a WCET point of view.
A standard instruction cache makes each instruction's WCET potentially suffer from the penalties of a cache miss and subsequent refill. The method cache is structured in order to free all instructions from cache miss penalties, these penalties are relegated to the function calls and returns.
The basic idea is that functions do not usually have big sizes and that an instruction cache could be divided in blocks that can possibly hold an entire function. The whole replacement mechanism is so performed on blocks of instructions not on single instructions: the instructions of a functions are in this case guaranteed to be cache hits while eventual cache misses can happen on function calls (called function miss) and function returns (caller function miss).
No way for loading single instructions is provided: cache can only be manipulated through blocks. Since the size of a block is fixed and functions bigger than this size can exist functions are allowed to be split into several blocks.
Some limitations caused by the method cache structure do exist:

- The size of the code sequence that can be loaded is bounded by the size of the method cache. However, as said, functions can be split in more segments
- Since `call` instruction does not carry any information regarding the function's size this information has to be saved in memory just before the first function's instruction
- Due to block-oriented manipulation of the cache more code than the function's one could be loaded into the cache

Just like other caches, different replacement policies can be implemented for the method cache: both `FIFO` and `LRU` are being investigated in the Patmos development.

2.1.1.1 FIFO

When working with the `FIFO` policy the method cache allocates to each segment of code a number of adjacent blocks. The first block assigned to the segment is labeled with the base address of the code segment it is holding, the other blocks labels are empty. The method cache is extended with a pointer, pointing to the location where a new loaded block will be placed and updated according to the `FIFO` semantics.

2.1.1.2 LRU

[SBH+] suggest that LRU replacement can be implemented in Patmos. This implementation needs more memory to work: a segment of code is no longer placed in adjacent blocks but its blocks can be split in the whole cache according to LRU semantics. Each block must be therefore extended with its real memory address and it LRU timestamp.
When an instruction has to be fetched all the cache blocks have to be looked for, if one of the blocks address contains the searched instruction then it is not loaded (already in the cache). If the instruction is not in the cache then its code segment has to be loaded.
A method cache may not have enough space to hold a new code segment, in this case some older segment has to be removed from the cache: note that the replacement mechanism removes all the blocks of a segment, not single blocks (for doing this a data structure is needed to keep track of which blocks make up a segment.

LRU implementation clearly need more hardware than the FIFO one and may also result in an additional pipeline stage in order to perform the lookup needed at each instruction fetch.

2.1.2 Data Cache

The data cache, as a normal cache, is used to speed up memory accesses. Patmos data cache uses a write-through policy, every store instruction writes changes to the data cache and to the main memory. Write-through policy leads to a higher number of memory accesses but results in a more analyzable system from a WCET perspective (less cache states to be taken into account).
Different opcodes for memory access instructions are provided by the Patmos instruction set allowing not only access different types of value (8, 16 and 32 bits) but also to perform blocking memory operations (stalling the pipeline) or decoupled ones.
Operations accessing directly the global memory are provided in order to bypass the data cache.

2.1.3 Stack Cache

The stack cache presented in [ABS13] is a dedicated cache meant to hold stack allocated values. The idea of having more than a cache holding different types of values (data cache, stack cache and instruction cache) is intended to help simplify the WCET analysis.
The stack area holds return address, callee saved registers, and function local variables. This data has a high access frequency and so it will greatly benefit of caching. Moreover stack data is always function local and therefore some optimizations can be done: due to its locality stack cache data does not need to be consistent with the main memory and when a function returns this data can simply be discarded with no need to write back the content. The stack cache has a fixed size so when some of its content has to be replaced it is written back to memory.
The stack cache is managed by the compiler: when a function is called a stack frame of the required size is reserved on the top of the stack and when a function returns this space is freed. When a function returns to a caller the frame of the caller must be ensured to be in the stack cache (due to the function call chain it may have been split to the main memory). Doing so each memory access in the body of the function is guaranteed to be a hit while a stack cache miss can only happen when reserving or ensuring a stack frame.

In order to manipulate the stack as just described the Patmos ISA defines the following instructions:

- `reserve - sres`: reserves space in the stack cache and may spill data to the main memory if there is no enough free space
- `ensure - sens`: when returning from a function call `sens` ensures that the stack frame of the calling function is available. May need to fill the cache with previously spilled data
- `free - sfree`: frees the reserved space on the stack

These instructions specify as an immediate parameter the size of the frame to be reserved/ensure so that the interaction with stack in the function calls tree can be easily analyzed (easy WCET calculation).

In the Patmos implementation the stack cache is a data structure placed in the local memory holding values from the actual global memory stack. Consistency with the global memory is managed through 2 pointers:

- `st`: pointer to the top of the stack
- `ss`: pointer to the last element of the stack cache spilled to main memory

The code in listing 2.1 represents a simple example used to show the stack cache behaviour. Assuming each function reserves (and ensures after each function call) 16B and the whole stack cache size is 32B, the evolution of stack and stack cache will be the one in figures 2.1, 2.2 and 2.3.

Listing 2.1: Code snippet to show the stack cache behaviour

```
1
2  void C() {
3    // ...
4  }
5
6  void B() {
7    // ...
8    C();
9    // ...
10 }
11
12
13 void A() {
```

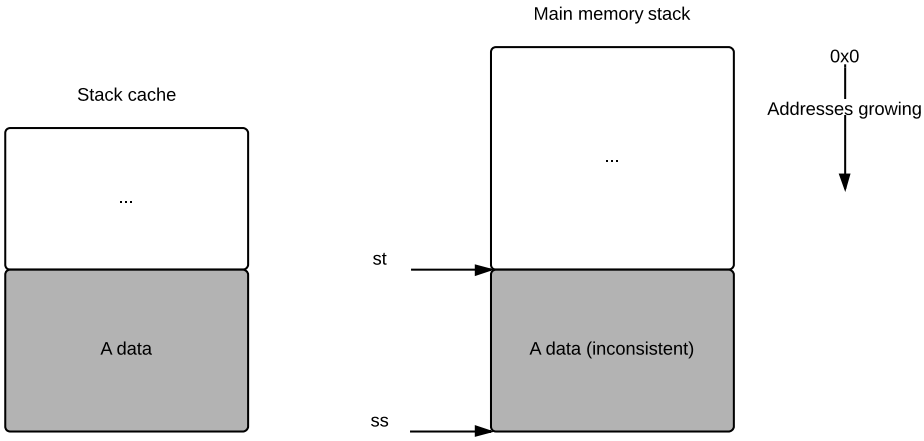


Figure 2.1: Stack cache status after the call A(). Space for A is reserved on the stack and the stack cache. The main memory is not consistent with the cache

```
14    // ...
15    B();
16    // ...
17  }
18
19  A();
```

The organization of the stack cache has some drawbacks:

- At any moment the program cannot access more stack data than the stack cache size. To solve this problem the stack frame can be split or the shadow stack can be used, an extra stack placed in main memory
- When a pointer to data placed on the stack cache is passed to another function it has to be ensured that the pointed data will not be split to main memory as long as it is needed. Otherwise this kind of aliased data can be placed on the shadow stack
- Reserve and ensure instructions only accept constant size immediate arguments, therefore dynamic size data has to be placed on the shadow stack

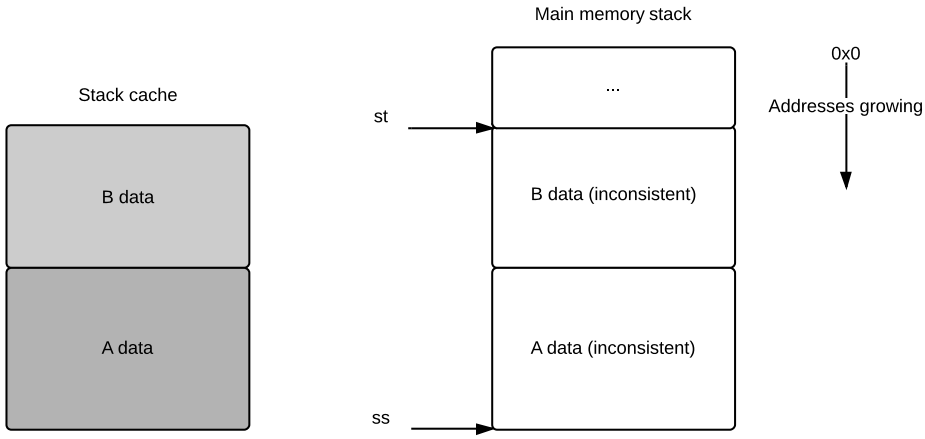


Figure 2.2: Stack cache status after the call B(). Space for B is reserved on the stack and the stack cache. The stack cache becomes full, main memory is still not consistent with the cache

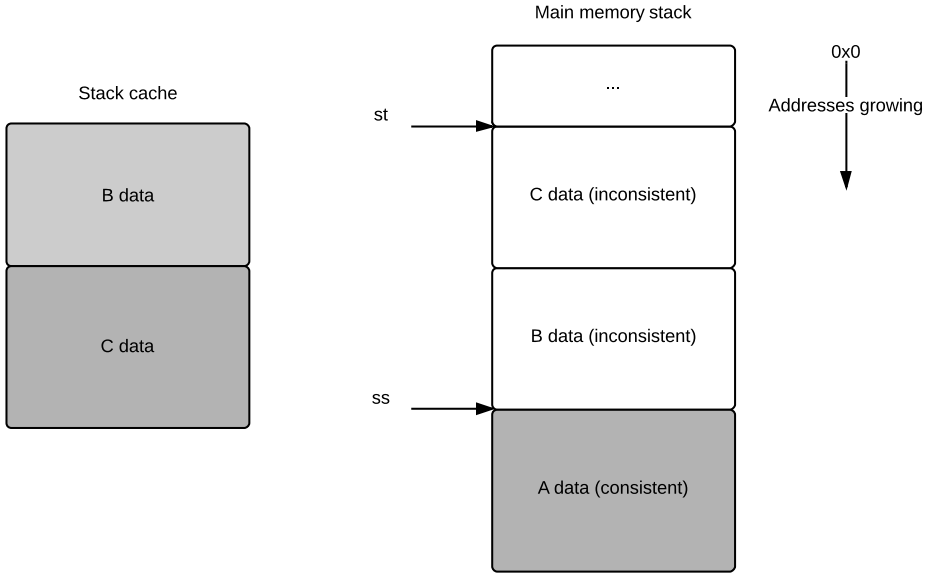


Figure 2.3: Stack cache status after the call C(). The data for A is spilled to main memory and replaced with C data

2.1.4 Data Scratchpad

Patmos provides a local memory called data scratchpad. The Patmos ISA has instructions that allow accessing this local memory. These instructions do not stall the pipeline.
The address space of the local memory is the same as that of global one but the used range is different. Instructions to access the data scratchpad are detailed in Section 2.3.8.

2.2 Registers

Patmos provides three distinct register files:

- `R`: 32 general-purpose registers (32 bit), shown in figure 2.5
- `S`: 16 special-purpose registers (32 bit), shown in figure 2.6
- `P`: 8 predicate registers (1 bit), by convention `p0` is set to true (`1`), shown in figure 2.4

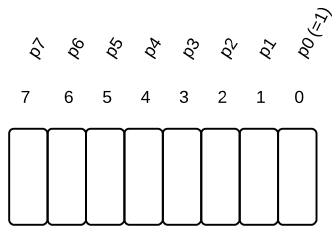


Figure 2.4: Predicate registers (`P`)

`r0` is always set to `0` and read-only. The other general-purpose registers are used, according to a compiler convention, as follows:

- `r1` and `r2`: contain the result of a function (up to 64 bits)
- `r3-r8`: contain the arguments of a function
- `r27`: temporary register
- `r28`: frame pointer

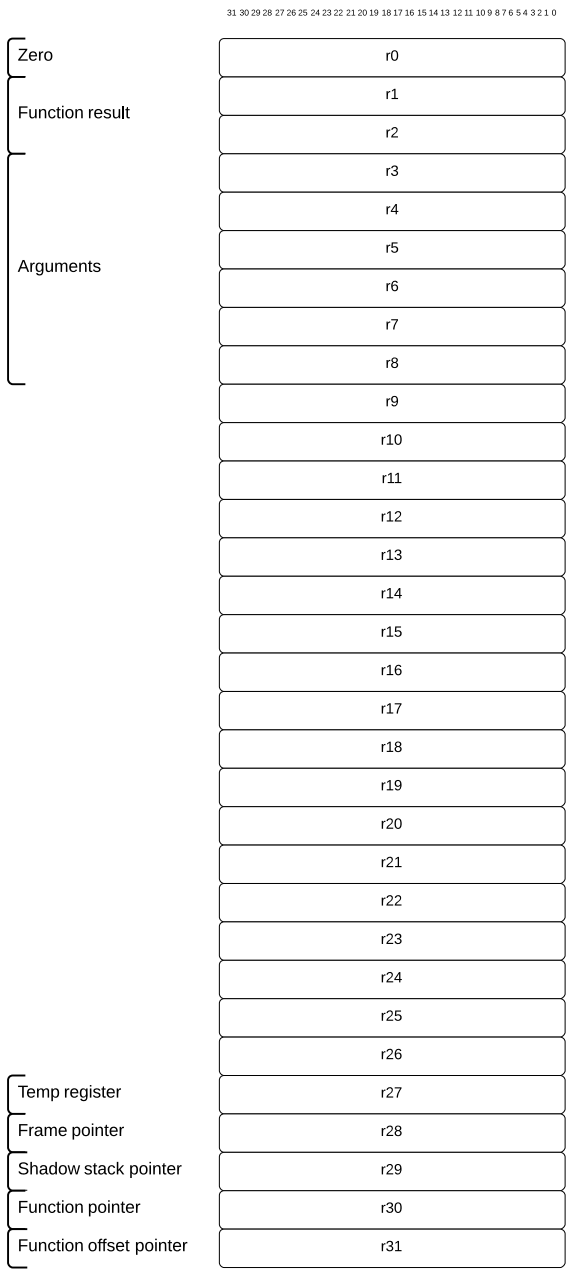


Figure 2.5: General-purpose registers (`R`)

- `r29`: shadow stack pointer
- `r30`: address of the return function (function base)
- `r31`: offset of the return instruction in the return function (function offset)

Special-purpose registers are dedicated to hold special values:

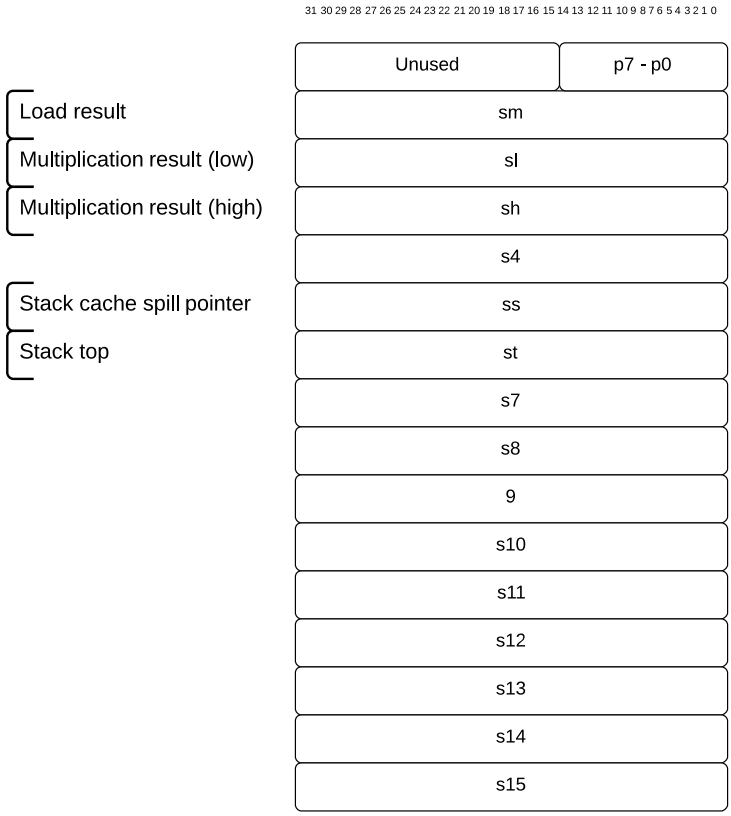


Figure 2.6: Special-purpose registers (`S`)

- `s0`: lower 8 bits are used to save and restore `P`, the other bits are reserved and not yet used
- `s1` (also named `sm`): contains the result of a decoupled load operation. The contained value can be signed or unsigned depending on the load's type

- `s2` (also named `sl`): lower 32-bits of the result of a multiplication
- `s3` (also named `sh`): upper 32-bits of the result of a multiplication
- `s5` (also named `ss`): stack cache spill pointer, point to the top element of the stack that is saved in the main memory
- `s6` (also named `st`): point to the top of the stack

2.3 Patmos ISA (Instruction Set Architecture)

Patmos allows bundles made of one or two instructions. Since instructions are 32 bits long bundles can be 32 or 64 bits. The most significant bit allows to distinguish between short and long bundles:

- 0: 32 bits long bundle
- 1: 64 bits long bundle

Several instruction's formats have been defined, each instruction respects a format; a single format can be used for several operations (with assigned opcodes). In the following all the Patmos instructions and operations are described.

2.3.1 Binary Arithmetic

In this section opcodes for binary arithmetic with registers (`ALUr`), immediate operands (`ALUi`) and long immediate operands (`ALUl`) are going to be examined. In table 2.1 opcodes semantics are described, `Op2` stands for immediate operand for `ALUi`, long immediate operand for `ALUl` and register operand for `ALUr`.

2.3.2 Multiply

Multiplications follow the `ALUm` format. Two opcodes are defined for this formats defining multiplication of normal integers and multiplication of unsigned integers, as shown in table 2.2. Multiplication are carried on in parallel with the normal pipeline and always terminate in constant number of cycles.

Opcode	Semantics
add	$Rd = Rs1 + Op2$
sub	$Rd = Rs1 - Op2$
xor	$Rd = Op2\hat{}Rs1$
sl	$Rd = Rs1 << Op2[0:4]$
sr	$Rd = Rs1 >>> Op2[0:4]$
sra	$Rd = Rs1 >> Op2[0:4]$
or	$Rd = Rs1 \mid Op2$
and	$Rd = Rs1 \& Op2$
nor	$Rd = \ (Rs1 \mid Op2)$
shadd	$Rd = (Rs1 << 1) + Op2$
shadd2	$Rd = (Rs1 << 2) + Op2$

Table 2.1: ISA: binary arithmetic

Opcode	Semantics
mul	$sl = Rs1 * Rs2$ $sh = (Rs1 * Rs2) >>> 32$
mulu	$sl = (uint32\_t)Rs1 * (uint32\_t)Rs2$ $sh = ((uint32\_t)Rs1 * (uint32\_t)Rs2) >>> 32$

Table 2.2: ISA: multiplication

2.3.3 Compare

Compare instruction follows the `ALUc` format. Operations perform comparisons between integers and unsigned integers putting the result in a predicate register.

2.3.4 Predicate

Predicate instruction follow the `ALUp`format and offer basic binary operations on predicate registers storing the result in a predicate register, opcodes and semantics are shown in table 2.4.

Opcode	Semantics
cmpeq	$Pd = Rs1 == Rs2$
cmpneq	$Pd = Rs1! = Rs2$
cmplt	$Pd = Rs1 < Rs2$
cmple	$Pd = Rs1 <= Rs2$
cmpult	$Pd = Rs1 < Rs2, unsigned$
cmpule	$Pd = Rs1 <= Rs2, unsigned$

Table 2.3: ISA: compare opcodes

Opcode	Semantics
por	$Pd = Ps1 \mid Ps2$
pand	$Pd = Ps1 \& Ps2$
pxor	$Pd = Ps1 \text{ ^ } Ps2$

Table 2.4: ISA: predicate manipulation opcodes

2.3.5 NOP

A single cycle no-operation instruction.

2.3.6 Wait

The `wait` instruction format is called `SPCw`. These instructions wait for a multiplication or a memory access to end stalling the pipeline. One single opcode is defined (`wait.mem`), it makes the pipeline stall until a memory operation ends.

Opcode	Semantics
wait.mem	Wait the end of a memory access

2.3.7 Move To/From Special

`SPCt` format is defined for its only opcode `mts` which moves the content of a general purpose register to a special purpose register. `SPCf` format is specified

for the opcode `mfs` which moves the content of a special purpose register to a general purpose register. Semantics are shown in table 2.5.

Opcode	Semantics
mts	$Sd = Rs$
mfs	$Rd = Ss$

Table 2.5: ISA: move to/from special opcodes

2.3.8 Load/Store Typed

Operations to access to the stack cache (`sc`), to the local scratchpad memory (`lm`), to the data cache (`dc`) and to the global shared memory (`gm`) are available. Load operations, in table 2.6, follow the `LDT` format while store operations, in table 2.7, follow the `STT` format. Load and store operations to the stack cache and the local memory do not stall the pipeline while the others do. However, when accessing the local memory, if the address maps some I/O device the operation may stall the pipeline.
Decoupled loads are also implemented which do not stall the pipeline and allow the execution to continue. These instructions put the loaded value into the special register `sm`. A wait instruction can be used to explicitly stall the pipeline. When a decoupled is tried to be executed while a previous one is still being processed the pipeline is automatically stalled so that the result of the first load will be available for at least one processor's cycle. A *use-delay* must be respected before accessing the destination register of a load operation.
Stores to the data cache use the *write-through* strategy with no *write allocate*, that is: when writing to data not present in the cache this data is fetched into the cache, if the data is present it is updated.

Loads to the stack cache are relative to the `st` pointer and can be performed on both slots of an instruction bundle. The other loads can only be issued on the first slot.

2.3.9 Stack control

Stack control operation all refer to the `STC` format and allow to manipulate the stack cache. `STC` instruction immediate argument is always interpreted in word size. Opcodes referring to the operations detailed in Section 2.1.3 are shown in table 2.8.

Opcode	Semantics
lws	$Rd = sc[Ra + Imm << 2]_{32}$
lwl	$Rd = lm[Ra + Imm << 2]_{32}$
lwc	$Rd = dc[Ra + Imm << 2]_{32}$
lwm	$Rd = gm[Ra + Imm << 2]_{32}$
lhs	$Rd = (int32\_t)sc[Ra + Imm << 1]_{16}$
lhl	$Rd = (int32\_t)lm[Ra + Imm << 1]_{16}$
lhc	$Rd = (int32\_t)dc[Ra + Imm << 1]_{16}$
lhm	$Rd = (int32\_t)gm[Ra + Imm << 1]_{16}$
lbs	$Rd = (int32\_t)sc[Ra + Imm]_8$
lbl	$Rd = (int32\_t)lm[Ra + Imm]_8$
lbc	$Rd = (int32\_t)dc[Ra + Imm]_8$
lbm	$Rd = (int32\_t)gm[Ra + Imm]_8$
lhus	$Rd = (uint32\_t)sc[Ra + Imm << 1]_{16}$
lhul	$Rd = (uint32\_t)lm[Ra + Imm << 1]_{16}$
lhuc	$Rd = (uint32\_t)dc[Ra + Imm << 1]_{16}$
lhum	$Rd = (uint32\_t)gm[Ra + Imm << 1]_{16}$
lbus	$Rd = (uint32\_t)sc[Ra + Imm]_8$
lbul	$Rd = (uint32\_t)lm[Ra + Imm]_8$
lbuc	$Rd = (uint32\_t)dc[Ra + Imm]_8$
lbum	$Rd = (uint32\_t)gm[Ra + Imm]_8$
dlwc	$sm = sc[Ra + Imm << 2]_{32}$
dlwm	$sm = gm[Ra + Imm << 2]_{32}$
dlhc	$sm = (int32\_t)dc[Ra + Imm << 1]_{16}$
dlhm	$sm = (int32\_t)gm[Ra + Imm << 1]_{16}$
dlhuc	$sm = (uint32\_t)sc[Ra + Imm << 1]_{16}$
dlhum	$sm = (uint32\_t)gm[Ra + Imm << 1]_{16}$
dlbuc	$sm = (uint32\_t)dc[Ra + Imm]_8$
dlbum	$sm = (uint32\_t)gm[Ra + Imm]_8$

Table 2.6: ISA: load typed operations

The `sres` and `sens` operations are blocking operations.

2.3.10 Call and Branch

These instructions apply to the `CFLb` format and supports different opcodes allowing to perform function calls (`call`), branches (`brcf`) and local branches (`br`) within the method cache.
A `call` instruction performs a function call and therefore stores the current

Opcode	Semantics
sws	$sc[Ra + Imm << 2]_{32} = Rs$
swl	$lm[Ra + Imm << 2]_{32} = Rs$
swc	$dc[Ra + Imm << 2]_{32} = Rs$
swm	$gm[Ra + Imm << 2]_{32} = Rs$
shs	$sc[Ra + Imm << 1]_{16} = Rs[15:0]$
shl	$lm[Ra + Imm << 1]_{16} = Rs[15:0]$
shc	$dc[Ra + Imm << 1]_{16} = Rs[15:0]$
shm	$gm[Ra + Imm << 1]_{16} = Rs[15:0]$
sbs	$sc[Ra + Imm]_8 = Rs[7:0]$
sbl	$lm[Ra + Imm]_8 = Rs[7:0]$
sbc	$dc[Ra + Imm]_8 = Rs[7:0]$
sbm	$gm[Ra + Imm]_8 = Rs[7:0]$

Table 2.7: ISA: store typed operations

Opcode	Semantics
sres	*Reserves space on the stack (eventually spilling other frames to main memory)*
sens	*Ensures that a stack frame is entirely loaded otherwise refills the stack cache*
sfree	*Frees space on the stack frame (no spill/fill)*

Table 2.8: ISA: stack control operations

function offset into register `r31` (function address has to be saved manually by the programmer/compiler) in order to return from the call itself.
Since both `call` and `brcf` opcodes perform non local jumps they can result in a method cache miss and a subsequent refill. On the contrary a `br` is always guaranteed to be a cache hit.
`call` is an absolute operation so the immediate operand is interpreted as an *unsigned int* while `br` and `brcf` are PC-relative and therefore the operand is considered to be *signed*. In both cases the operand is assumed to be word size.

Call and branch instructions are executed in the `EX` stage of the pipeline, the instructions fetched during the decode and the execution step of a branch are

call	*Function call, absolute, with cache fill*
br	*Local branch, PC relative, always hit*
brcf	*Local branch, PC relative, with cache fill*

Table 2.9: ISA: immediate Call and Branch

still executed, this results in a delay of 2 instructions which has to be taken into account be the compiler/programmer (`branch-delay slots`).
All call and branch instructions can only be placed in the first position of the bundle.

2.3.11 Call and Branch Indirect

This instruction is just like the call and branch one but takes a register operand rather than an immediate operand. A new format, called `CFLi` is therefore defined. As for the call and branch the three opcodes `call`, `brcf` and `br` are provided. Everything that was said in Section 2.3.10 regarding method cache interaction, parameter interpretation, and branch delay is also true for call and branch indirect.

call	*Function call, absolute, with cache fill*
br	*Local branch, PC relative, always hit*
brcf	*Local branch, PC relative, with cache fill*

Table 2.10: ISA: register Call and Branch

2.3.12 Return

Transfer control back to the calling function or return from an interrupt. `ret` may cause a cache miss and subsequent cache refill to load the target code.

Chapter 3

The Operating System

TiCOS [BMV12] is a time-composable real-time operating system developed within the framework of the Probabilistically Analysable Real-Time Systems (PROARTIS) project (`www.proartis-project.eu`) supporting the ARINC653 software specification and originally targeting the PPC architecture. TiCOS is based on POK [DL11], a light weight operating system implementing the ARINC653 standard and distributed under the BSD license.

The high level architecture of TiCOS is shown in figure 3.1. It is made of two basic layers: the kernel layer and the application layer. The kernel layer is responsible for implementing the OS services while the application layer is composed by the user application, a core library layer, the ARINC library layer and the middleware library layer. The ARINC library offers a simple set of ARINC services. Those services are implemented (just like the core library functionalities) through the system call mechanism to the middleware layer. That mechanism makes the systems start executing in supervisor mode and call the proper kernel layer services.

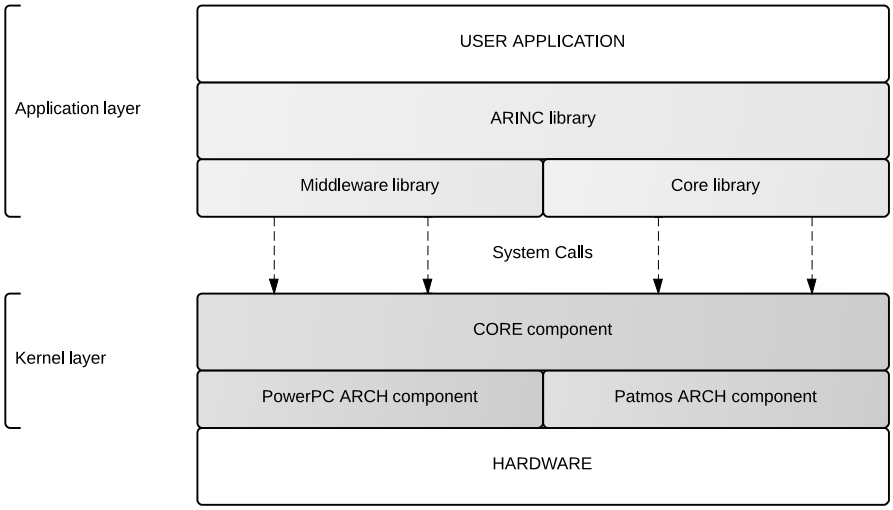


Figure 3.1: High level structure of the OS (from [BMV12])

3.1 The Kernel Layer

This layer is made of two fundamental components: an architectural dependent part (called *arch component*) and an architectural independent part (called *core component*), both parts implement kernel services but while the first one accesses directly architectural functionalities (to implement for example: timer, context switch, memory management, etc.) the *core component* does not need to directly use the underneath architecture. This kind of structure makes it possible to target an architecture different from PPC focusing only on changing the *arch component* and eventually adjusting the *core component*. Adjustments to the core component may be needed, as in the case of Patmos porting, for operating systems not supporting memory virtualization or for changing the loading mechanism of applications.

The kernel layer as a whole provides the following services:

- Partitions support. Partition is the name used by ARINC to indicate an application made by one or more threads. TiCOS wants to provide time-composability and for doing so it implements time and space isolation among partitions. Each partition is granted its own space and no shared memory is allowed
- Scheduling. A two-level scheduling algorithm is implemented. First a

partition is selected for execution according to a cyclic strategy then a selected partition's thread is chosen according to a fixed priority policy

- Lock objects management. Lock objects are used by the middleware layer to implement events and thus providing synchronization for buffers and blackboards (ARINC communication mechanisms). TiCOS works in a run-to-completion semantics which assures there's no need to worry about mutual exclusion; therefore locks are not used for that purpose. A lock object memorizes threads which are waiting for it to be free, at the moment priority policies are not used to select the next thread to be unlocked
- Ports support. Ports are the ARINC way of communicating between partitions, since partitions do not share memory the kernel is responsible for copying messages from source ports to destination ports

3.2 ARINC653 Entities

The ARINC specification defines several entities; some of these entities represent active components of a real-time systems, such as partitions and processes, while others represent communication mechanisms such as events, semaphores, blackboards, buffers and ports.

3.2.1 Partitions and Processes

An application is usually made of an ARINC partition (the equivalent of a POSIX process) and each partition is made of threads. Each partition has a main thread which runs only once and creates all other partition's threads. Each partition is allocated a fixed time slot and is not allowed to share memory with other partitions. ARINC processes are the way the ARINC653 specification names tasks or POSIX threads. Processes have defined priorities and are allowed to share memory. In TiCOS the scheduling of processes inside a partition is made by a fixed-priority constant time scheduler, called O(1) scheduler and detailed in Section 3.4.2.

3.2.2 Events

ARINC events can assume one between two states: *set* ("up") and *reset* ("down"). When an event is in *set* state it allows all the waiting processes to continue.

When an event is in *reset* state all processes waiting for the event are blocked [TBV10]. In TiCOS ARINC events are implemented through kernel locks; each middleware event is associated to one and only one kernel lock (and has the same identifier).

3.2.3 Semaphores

The aim of ARINC semaphores, just like normal semaphores, is to provide mutual exclusion in accessing a shared resource. Shared resources can only exist between processes of the same partition, however, as said, TiCOS assumes run-to-completion semantic between the processes of the same partition, this means that there is no need to worry about mutual exclusion. ARINC semaphores are implemented as nothing more than stubs since a semaphore lock is always granted by construction.

3.2.4 Blackboards

Blackboards are one of the mechanisms used to make processes of a partition communicate. A blackboard holds only one message at once, so if a process displays a new message in a non-empty blackboard the previously shown message gets lost. Events are used to allow more processes use the same blackboard: an event is associated to each blackboard, when a process tries to read from an empty blackboard the event is set to "down" and the process is suspended. When a process displays a message the event is set to "up" and all processes waiting for that event become runnable again.

3.2.5 Buffers

Buffers are another mechanism used to make processes of the same partition communicate. More than a message can be stored in a non-empty buffer, processes are not allowed to write to a full buffer and when a message is read it is removed from the buffer. When a process tries to receive a message from an empty buffer the associated event is set to "down". When a message is sent to an empty buffer the event is set to "up". Moreover, an associated event is used to handle the buffer's fullness: the event is set to "down" when a process tries to write to a full buffer and when a message is read from a full buffer the event is set to "up", making runnable all the waiting processes.

3.2.6 Sampling and Queueing Ports

Ports are used to make different partitions communicate. A queueing port is a communication channel which allows to enqueue tokens and read in the exact order they have been inserted and is the inter-partition equivalent to a buffer [TBV10]. A sampling port allows to read only the last inserted value, so writing to a sampling port means updating its content. Sampling ports are the equivalent to blackboards with the difference that writing and reading from a sampling port is never blocking [TBV10].

3.3 The Library Layer

The operating system library layer allows the user application to access TiCOS services through user-level functions. These functions are implemented through system calls to kernel services. The library is structured as shown in figure 3.2.

3.3.1 Core Library

The contained functions offer the operating system's core functionalities. Some of these functions are used to implement the ARINC library. Application error codes are also defined. These functions are not part of the ARINC653 specification, using them in the user code results in a non-portable application. The functions in the core and middleware library are the only part of the POK operating system ([DL11]) kept unmodified, this is the reason why their names start with the `pok_` prefix.

- **Event**: is implemented through a kernel lock
  - `pok_event_create`: performs a system call to create a kernel lock
  - `pok_event_wait`: performs a system call to wait for the event associated to the lock, with a specified timeout
  - `pok_event_broadcast`: performs a system call to wake up all the threads waiting for the event associated to the lock
  - `pok_event_signal`: performs a system call to notify the first thread waiting for the event
  - `pok_event_lock`: performs a system call to lock the kernel lock corresponding to the event (event set to "down")

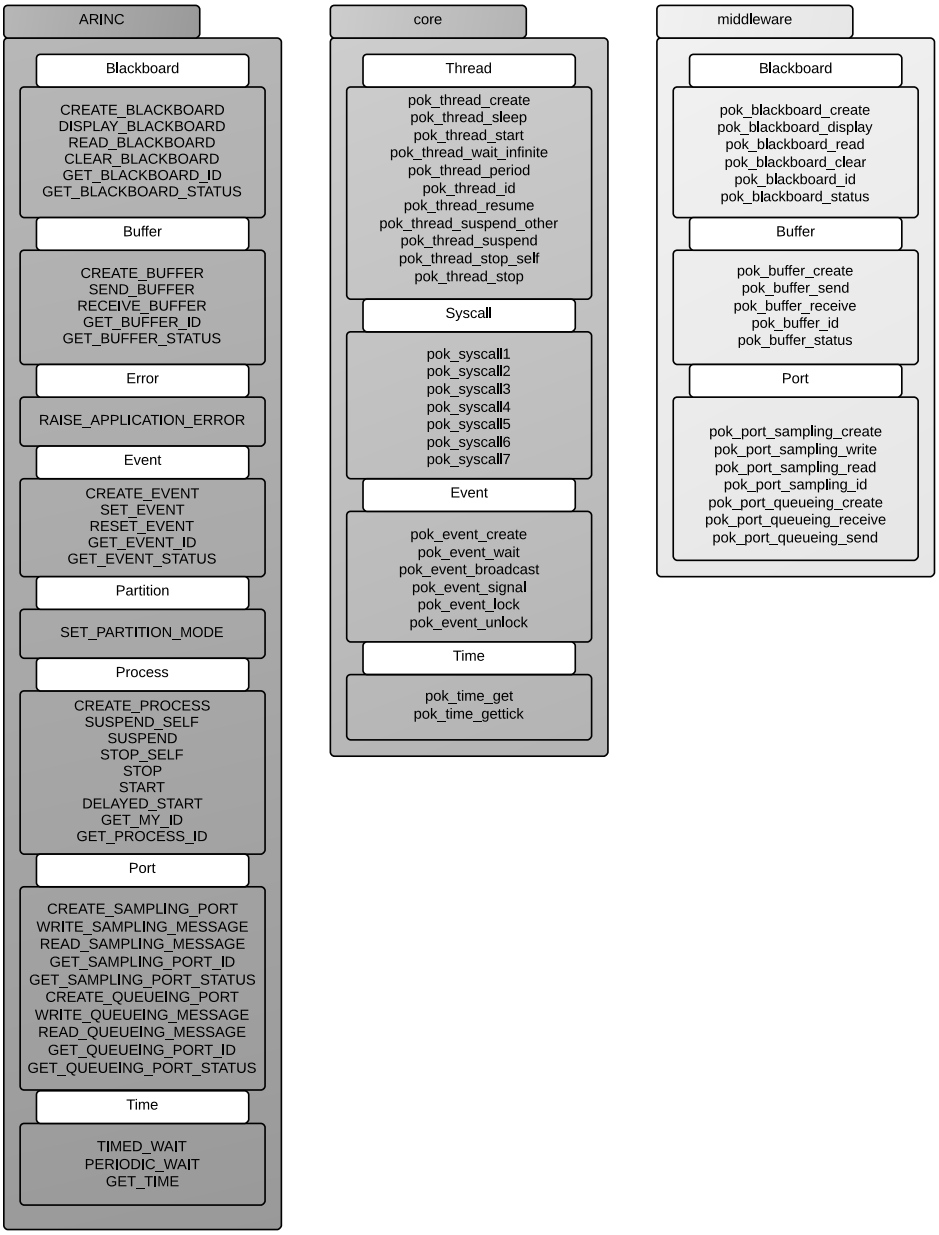


Figure 3.2: Structure of the OS library for user applications

  - `pok_event_unlock`: performs a system call to unlock the kernel lock corresponding to the event (event set to "up")

- **Syscall**: An identifier for each system call is defined as depicted in table 3.1. The following functions realize the system calls for different number of parameters.

System call	ID
CONSWRITE	10
GETTICK	20
GET_TIME	21
THREAD_CREATE	50
THREAD_SUSPEND	52
THREAD_STOP	55
THREAD_PERIOD	56
THREAD_STOPSELF	57
THREAD_ID	58
THREAD_SUSPEND_OTHER	67
THREAD_RESUME	68
THREAD_START	69
THREAD_DELAYED_START	70
MIDDLEWARE_SAMPLING_ID	101
MIDDLEWARE_SAMPLING_READ	102
MIDDLEWARE_SAMPLING_WRITE	104
MIDDLEWARE_SAMPLING_CREATE	105
MIDDLEWARE_QUEUEING_CREATE	110
MIDDLEWARE_QUEUEING_SEND	111
MIDDLEWARE_QUEUEING_RECEIVE	112
LOCKOBJ_CREATE	201
LOCKOBJ_OPERATION	202
ERROR_HANDLER_CREATE	301
ERROR_HANDLER_SET_READY	302
ERROR_RAISE_APPLICATION_ERROR	303
ERROR_GET	304
PARTITION_SET_MODE	404

Table 3.1: System call IDs

  - `pok_syscall1`
  - `pok_syscall2`
  - `pok_syscall3`
  - `pok_syscall4`
  - `pok_syscall5`

  - `pok_syscall6`
  - `pok_syscall7`

- **Thread**:
  - `pok_thread_create`: performs the `THREAD_CREATE` system call which creates a thread with the specified identifier
  - `pok_thread_sleep`: performs the `THREAD_SLEEP` system call which makes the thread sleep for the specified amount of time
  - `pok_thread_start`: performs the `THREAD_START` system call which starts the specified thread
  - `pok_thread_wait_infinite`: alternative name for `pok_thread_suspend`
  - `pok_thread_period`: performs the `THREAD_PERIOD` system call which ends the current thread's period
  - `pok_thread_id`: performs the `THREAD_ID` system call which sets a value at a given location to the current thread identifier
  - `pok_thread_resume`: performs the `THREAD_RESUME` system call which resumes a thread given an id
  - `pok_thread_suspend_other`: performs the `THREAD_SUSPEND_OTHER` system call which suspends the specified thread
  - `pok_thread_suspend`: performs the `THREAD_SUSPEND` system call which suspends the current thread
  - `pok_thread_stop_self`: performs the `THREAD_STOP_SELF` system call which stops the current thread
  - `pok_thread_stop`: performs the `THREAD_STOP` system call which stops the specified thread

- **Time**:
  - `pok_time_get`: performs the `GET_TIME` system call which returns a 64 bit value containing the current time
  - `pok_time_gettick`: performs the `GETTICK` system call which returns the number of ticks since the system started

3.3.2 Middleware Library

The functions contained in this layer implement most of the functionalities offered by the ARINC library but are not part of the ARINC653 specification. These functionalities should be used through ARINC library functions, accessing them directly results in a non-portable application.

- **Blackboard**:
  - `pok_blackboard_create`: creates a blackboard data structure and associates an event to it
  - `pok_blackboard_read`: reads a message from the blackboard, if the blackboard is empty the associated event is locked and the caller waits the specified timeout. When a message arrives or the timeout expires the event is unlocked
  - `pok_blackboard_display`: displays a message in the blackboard and unlocks the associated event
  - `pok_blackboard_clear`: empties the blackboard
  - `pok_blackboard_id`: returns blackboard's identifier
  - `pok_blackboard_status`: returns blackboard's status
- **Buffer**:
  - `pok_buffer_create`: creates a buffer data structure and the associated events
  - `pok_buffer_receive`: reads a message from the buffer, if the buffer is empty the associated event is locked and the caller waits the specified timeout. If the buffer becomes empty the event is locked, if it becomes non-full the event is unlocked
  - `pok_buffer_send`: tries to send a message to the buffer, if it is full the associated event is locked and the caller waits for it to become unlocked. If the buffer becomes non-empty the corresponding event is unlocked
  - `pok_buffer_status`: returns the status of the buffer
  - `pok_buffer_id`: returns the identifier of the buffer
- **Port**:
  - `pok_port_queueing_create`: creates a queueing port
  - `pok_port_queueing_receive`: receives data present in the queue
  - `pok_port_queueing_send`: sends data to the port if it is not full
  - `pok_port_sampling_create`: creates a sampling port
  - `pok_port_sampling_write`: writes some content to a sampling port, updating the previously stored data
  - `pok_port_sampling_read`: reads the content of the sampling port which according to the sampling semantics is the last written content
  - `pok_port_sampling_id`: gets the sampling port identifier

3.3.3 ARINC Library

The functions contained in this part of the library deal with the creation and management of the data structures presented in Section 3.2. Blackboards, buffers and ports functions are implemented calling the ones in the middleware library presented in Section 3.3.2.

- **Error**:
  - `RAISE_APPLICATION_ERROR`: raises an application error through a system call
- **Event**:
  - `CREATE_EVENT`: creates an event using the core library, the ARINC event is created locked
  - `SET_EVENT`: calls the `pok_event_signal` function of the core library
  - `RESET_EVENT`: calls the `pok_event_lock` of the core library and sets the event to "down"
  - `GET_EVENT_ID`: returns the identifier of the event
  - `GET_EVENT_STATUS`: returns the status of the event
- **Partition**:
  - `SET_PARTITION_MODE`: calls the core library and sets the partition operating mode
- **Process**: As stated before an ARINC process corresponds to a POSIX thread. Processes are implemented through kernel threads
  - `CREATE_PROCESS`: creates a thread using the core library
  - `SUSPEND_SELF`: calls the core library and suspends the calling thread
  - `SUSPEND`: calls the core library and suspends a thread given an identifier
  - `STOP_SELF`: calls the core library and stops the calling thread
  - `STOP`: calls the core library and stops a thread given an identifier
  - `START`: calls the core library and starts a thread given an identifier
  - `DELAYED_START`: calls the core library and starts a thread given an identifier after a given delay
  - `GET_MY_ID`: returns the identifier of the calling process
  - `GET_PROCESS_ID`: returns a process identifier given its name

- **Time**:
  - `TIMED_WAIT`: calls the function `pok_thread_sleep` of the core library
  - `PERIODIC_WAIT`: calls the function `pok_thread_period` of the core library
  - `GET_TIME`: calls the function `pok_time_get` of the core library

3.4 TiCOS Time-composability

One of the goals of the development of TiCOS was obtaining time-composability. The ARINC 653 standard focuses on the creation of partitioned systems in which each application is time and space isolated from the others.
In order to assure time-composability at the kernel level two main requirements have to be guaranteed:

- **Zero-disturbance**: when using hardware which exposes history-dependent behaviour the execution of the operating system services must not influence the application timing behaviour. Two techniques were inspected in the development of the operating system [BMV12]: 1) using techniques similar to cache partitioning [Mue95]: reserving a cache portion for the OS services makes the OS still benefit from performance enhancements without affecting user code timing behaviour; 2) preventing the OS from using all the history dependent hardware. Moreover: time-triggered OS services (activated by a timer expire) may disturb the execution of application software. To mitigate this interference a deferred preemption mechanism is adopted by the OS in order to identify a limited number of preemption points assuring uninterrupted execution times.

- **Steady timing behaviour**: Different factors may influence the OS jittery time behaviour: 1) hardware state; 2) software state; 3) input data. Hardware state influence is taken into account along with the zero-disturbance treatment. Software state is solved by developing system services with a constant-time behaviour, such as a *O(1)* scheduler [BMV12]. The execution time of some services, such as ARINC communication services, may depend on the size of manipulated data. This type of influence is hard to be completely removed, the solution adopted is shown in Section 3.4.3.

3.4.1 Time Management

The time management approach used by TiCOS minimizes the interference of OS routines used to manage the time with the user applications. Instead of using a tick-counter-based approach which forces certain routines of the OS to be periodically executed, and thus interfere with applications, an interval timer mechanism is used. Originally implemented through the PPC `DEC` register the interval timer approach ensures to incur in less interference: an application is interrupted at fixed points. In the ARINC standard, where a system is divided into partitions each of which is assigned a time slot, the interval timer is set to expire only at partition switches so that no application is interrupted. During each time slot partition's processes (ARINC equivalent for task) share the processor, preemption is enabled only at the end of each job, implementing *run-to-completion* behaviour.

3.4.2 Scheduling

TiCOS has a constant-time (O(1)) fixed-priority scheduler inspired by the O(1) Linux scheduler adopted in kernel version 2.5 [Mol]. The selection of the highest priority task is implemented through bitwise operations and in order to avoid the use of FIFO queues (non-constant-time insertion) for each priority level each task is required to have a different level of priority. 255 levels of priority are defined, since ARINC applications have to support up to 128 processes a different priority level can be assigned to each of them.
A process can be set to be activated at each time in a scheduling slot but since a *run-to-completion* semantics is required the activation time is delayed until the end of the current process execution (see figure 3.3).

3.4.3 IO Communication

The communication between partitions is implemented through ARINC services: sampling and queueing ports. These two structures enable a message-oriented communication between applications. Ports can be a treat for time-composability mostly because of the variability of the amount of data that can be transferred. A possible solution could be forcing to transfer always as much data as the maximum size of a port, resulting in a constant time behaviour, but this would lead to a considerable performance loss.
To partially solve the problem the data-dependent part of each IO operation is delayed and placed where it will have less interference with the user appli-

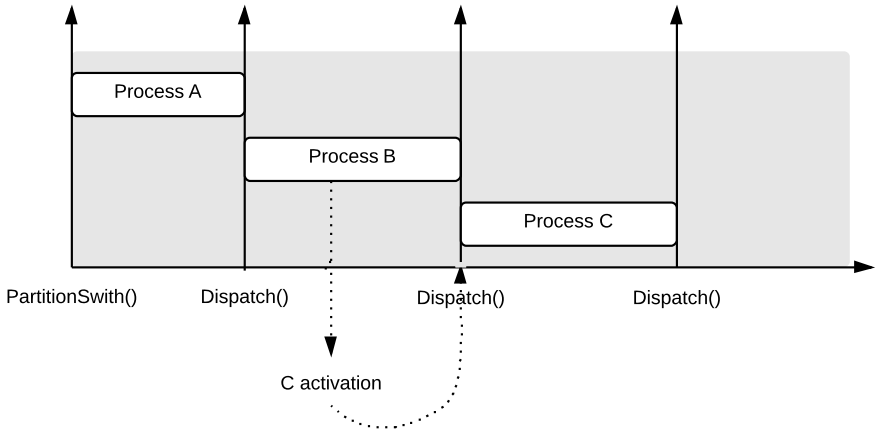


Figure 3.3: Run-to-completion within a time slot (from [BMV12])

cations. Each write to a port is placed in the slack time at the end of the execution of a partition before the end of the execution slot. Likewise data is read at the beginning of the execution slot, as shown in figure 3.4. In this way the correctness of the communication is guaranteed since we are dealing with inter-partition communication.

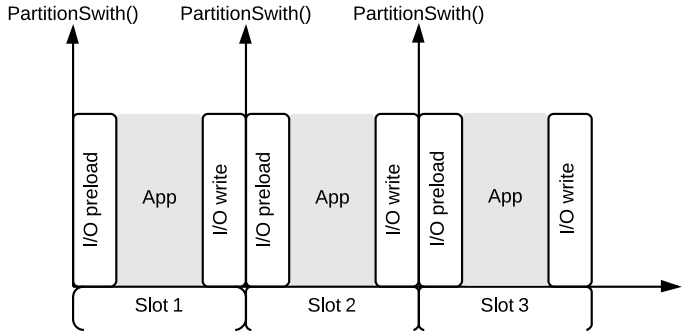


Figure 3.4: Inter-partition port communication (from [BMV12])

3.5 Build Chain

The compilation of an architecture realized with TiCOS requires two phases: 1) kernel and partition compilation; 2) integration of kernel and partitions into

a single executable. Kernel and each partition are compiled in different ELF files and then assembled in a single bootable ELF. Each partition contains a main function responsible for creating all the partition's threads.

3.5.1 Kernel Compilation

The kernel's compilation requires a configuration file, `deployment.h`, containing pre-compilation directives to configure the kernel's executable. `deployment.h` is included in every compilation unit. The build process creates the file `kernel.lo` as shown in figure 3.5.

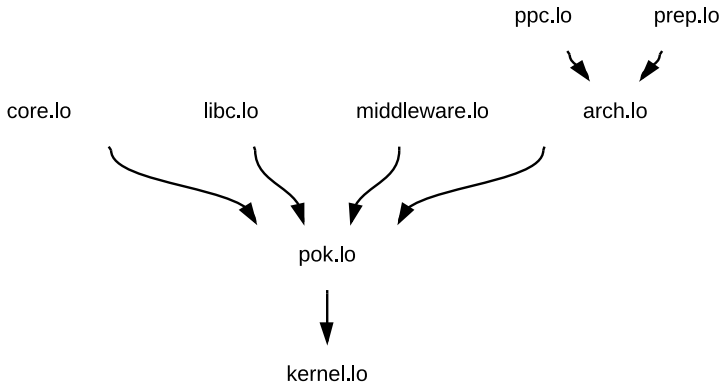


Figure 3.5: Visual representation of the compilation units concurring to form the kernel link object

A linker script (`kernel.lds`) is used to create the kernel's executable.

3.5.2 Partitions Compilation

Each partition willing to execute in the system is compiled separately into an ELF file. These executables contain the user code and the TiCOS library. First of all the TiCOS library is compiled using the configuration file `deployment.h` into a static library, secondly the partition ELF is built compiling with a linker script (`partition.lds`) the user code with the shared library just created, as depicted in figure 3.6.

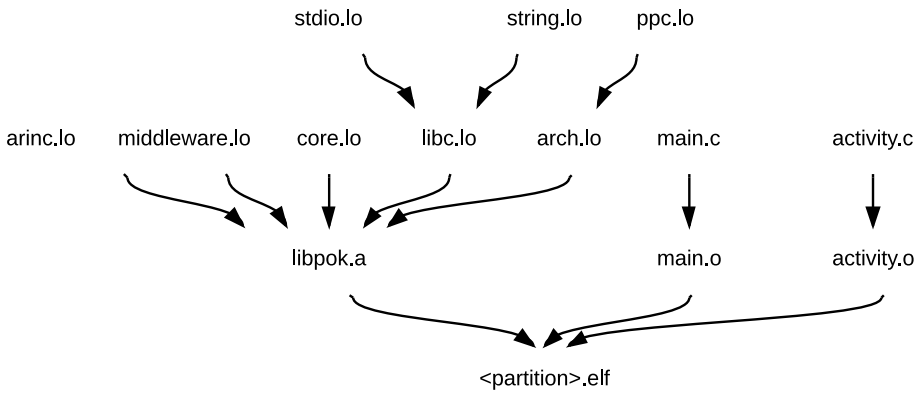


Figure 3.6: Visual representation of the compilation units concurring to form the partition ELF

3.5.3 Integration of Kernel and Partitions

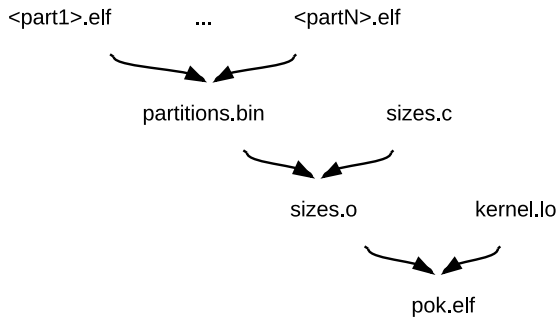


Figure 3.7: Visual representation of the integration process of kernel and partitions

To integrate kernel and partitions each partition executable is padded in order to have it aligned then all partition's ELF are assembled in the same binary archive `partitions.bin`. After this a `sizes.c` file is created, containing the sizes of each partition, and compiled obtaining `sizes.o`. `partitions.bin` is then added to `size.o` and finally compiled with `kernel.lo`. The process is described in figure 3.7.

Chapter 4

Processor Extensions

Coarsely, an RTOS offers services of three kinds: CPU allocation, memory allocation, and intra-task communication. In order to offer these functionalities, which allow to furnish a computational model more complex than the fixed allocation of a task to a processor, the underlying hardware should implement mechanisms such as interrupts and memory protection. This is the reason why the Patmos processor described in Chapter 2 has been extended with:

- Interrupts
- Stack cache manipulation instructions

Other extensions, e.g., for memory protection, have been discussed and investigated. However, since their implementation was not mandatory for the OS to work they have not been realized yet:

- Address range checks
- Supervisor mode and cache invalidation

4.1 Interrupts

Interrupts are used to stop the current flow of control jumping to a dedicated Interrupt Service Routine (ISR) in order to react to external events [DTU12b]. Interrupts are widely adopted by modern architectures for several reasons; interrupts are used, for instance, to implement preemptive scheduling mechanisms, start periodic operations, or react to I/O events.
When a context switch happens, processor's state must be stored to memory in order to restore it at a later point.

Different solutions for time and interrupt management have been developed. PPC 750 model [Fre12] offers a long-period timer called `TB - Time Base`. The `TB` is a 64-bit structure that consists of two 32-bit registers: time base upper (`TBU`) and time base lower (`TBL`). PPC 750 offers also a decrementer register `DEC`. The `DEC` register is a 32-bit decrementing register used to fire a *decrementer interrupt* at a programmable delay. `DEC` has the same update frequency as the time base. When a decrementer interrupt is fired, instruction fetching resumes at a specified position in main memory where an `ISR` (Interrupt Service Routine) has been previously set.
A more complete solution is AM17x/AM18x ARM Microprocessor which offers a Real-Time Clock (RTC) [Ins11]. Several RTC registers are mapped into memory: `SECOND`, `MINUTE`, `HOUR`, `DAY`, `MONTH`, `YEAR`, some memory-mapped registers can be used to set up interrupt intervals: `ALARMSECOND`, `ALARMMINUTE`, `ALARMHOUR`, `ALARMDAY`, `ALARMMONTH`, `ALARMYEAR`.

The Patmos RTC offers different registers mapped to the local memory, which allow to read/write different values.

Address	I/O Device
0xf0000300	Clock cycles (lower 32 bits)
0xf0000304	Clock cycles (upper 32 bits)
0xf0000308	Time in microseconds (lower 32 bits)
0xf000030C	Time in microseconds (upper 32 bits)
0xf0000310	Interrupt interval
0xf0000314	ISR address

Table 4.1: Memory mapped RTC registers

The clock cycles counter is a 64 bits value updated every CPU cycle. The time in microseconds (64 bits) value is incremented every microsecond. The interrupt interval value is decremented every CPU cycle and when it reaches 0 an interrupt

is fired and it is set back to its maximum value (both upper and lower registers to `0xffffffff`). The ISR address value can be used to set the address of the interrupt service routine which has to be called after an interval interrupt.

4.1.1 Simulator Implementation

In order to implement the interrupts mechanism (the memory mapped RTC and the ISR calls) the Patmos simulator (whose architecture is described in Appendix A) has been extended. A memory mapped resource has been added mapping the register of the RTC as shown in table 4.1. Given the architecture explained in appendix A a new class called `mm_rtc` has been created extending the base class `mapped_device`, this class makes available the addresses in table 4.1 and allows to access the RTC registers. In order to access RTC registers the `mm_rtc` class contains a field referencing an object of another class which actually implements the RTC functionalities (`rtc_t` class).

The `rtc_t` class holds a value for each register: clock cycles, time in microseconds, interrupt interval and ISR address and publishes methods to read and write those values. Moreover the `rtc_t` class has a method to notify the RTC that a cycle has passed, this method is called `tick`.

A single RTC object is created as the simulation starts and is contained in the class `simulator_t`, at every simulator cycle the `tick` method is called on the `rtc_t` object. Since the software simulator cycle can not execute with a guaranteed frequency the `tick` method increments by one both the clock cycle counter and the time in microseconds register; this method is also responsible for decrementing the interrupt interval value. The extended architecture is shown in figure 4.1.

The class `interrupt_handler_t` realizes the connection between the components firing interrupts and the simulation core, responsible of actually handling those interrupts. When an interrupt is fired it is appended to a collection of interrupt describers in the class `interrupt_handler_t`. At every cycle the simulator checks for interrupts pending in the `interrupt_handler_t` object, the handling code is shown in listing 4.1.

To handle the interrupts a new branch instruction has been defined, represented by the class `i_intr_t`, and it is inserted in the pipeline when a pending interrupt is detected. An `i_intr_t` instruction can not be decoded so it can not be

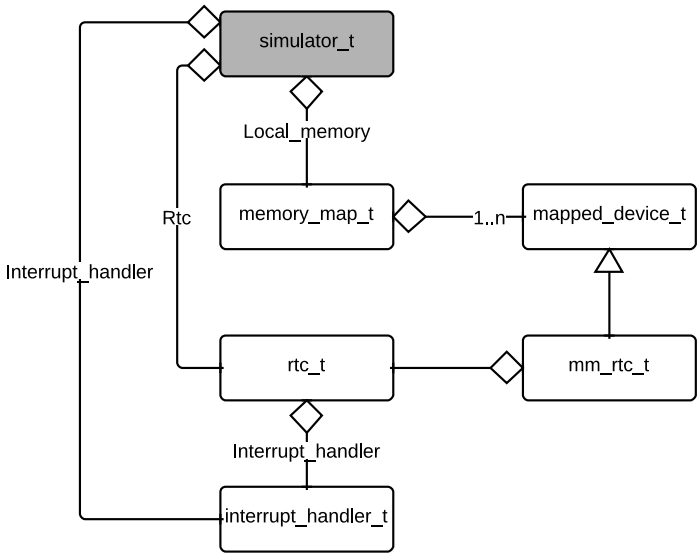


Figure 4.1: Class diagram for the RTC's simulator extension

used by the programmer. However, it can be executed by the simulator. An `i_intr_t` instruction is just like a branch instruction which jumps to the ISR address. After this instruction is added a special register (`s9`) is set to the return address (the `PC`). `interrupt_handling` counter is set in order to remember that an interrupt instruction has been added and since it behaves like a branch bubble instructions must be added in the next cycles.

Particular attention must be placed into handling interrupts fired during the execution of branch instructions. As shown in Section 2.3.10 branch instructions must be followed by `branch-delay slots`, during these slots the branch takes actually place. The execution of an interrupt instruction should be placed after those slots, but in case of loop it has to be placed before the branch instruction is executed again. Those are the reasons why if a branch instruction is decoded a counter is set in order to make the simulator aware that the next instructions have not to be interrupted.

Listing 4.1: Interrupt handling code added to the simulator main cycle

```
1 if (Interrupt_handler.interrupt_pending() &&
2   branch_counter == 0 &&
3   interrupt_handling == 0)
4 {
5
```

```
6     interrupt_t &interrupt = Interrupt_handler.get_interrupt();
7
8     Pipeline[0][0] = instruction_data_t::mk_CFLb(*intr, p0, interrupt
          .Address);
9     Pipeline[0][1] = instruction_data_t();
10
11    // Handling interrupt, next CPU cycle no new instructions have to
          be decoded
12      interrupt_handling = 3;
13
14      // Store return from interrupt address
15      SPR.set(s9, PC);
16  }
17  else
18  {
19    if (interrupt_handling > 0)
20    {
21      // Putting more empty instructions
22      Pipeline[0][0] = instruction_data_t();
23      Pipeline[0][1] = instruction_data_t();
24      interrupt_handling--;
25    } else {
26      // fetch the instruction word from the method cache.
27      Method_cache.fetch(PC, iw);
28      iw_size = Decoder.decode(iw,  Pipeline[0]);
29
30      // provide next program counter value
31      if(Pipeline[0][0].I->is_flow_control())
32        branch_counter = 2;
33      else if (branch_counter)
34        branch_counter--;
35
36      nPC = PC + iw_size*4;
37    }
38  }
```

4.2 Stack Cache Manipulation

As noted in Section 2.1.3 the stack cache is handled using three simple instructions:

- `sres`
- `sens`
- `sfree`

Any of those instructions takes an immediate parameter. In order to support context switching more complex ways of manipulating the stack cache are needed. Section 2.1.3 shows how the type of data a stack cache contains allows not to implement a write-back policy; this may lead to inconsistencies: the stack cache can contain data not persisted in the actual stack. When a context switch happens there is the need not only to save registers state but the state of the stack cache needs to be saved for the current executing thread and not only registers state of the next executing thread but even its stack cache has to be restored. Figure 4.2 shows how the stack cache can have a more updated state than the actual stack and the values of the pointers `ss` (stack spill address) and `st` (stack pointer).

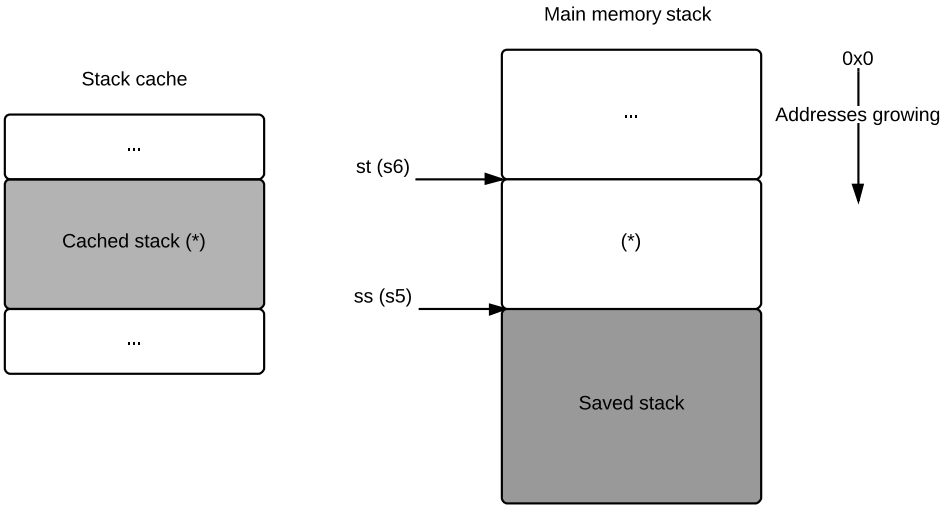


Figure 4.2: Possible state of the stack cache and the actual stack during the execution

When a context switch occurs the stack cache state may be the one depicted in figure 4.2. In this case stack cache content has to be saved to the main memory since it has not been saved yet and the new executing thread may be changing the stack cache itself. None of the three previously defined instructions allows writing back a specific amount of stack cache, so a new instruction, `sspill` has been introduced.

`sspill` instruction takes an immediate or a register parameter and saves the specified amount of stack cache in main memory downward, starting from address `ss`. This means that, starting from the configuration in figure 4.2 and supposing to execute the instructions in listing 4.2, the stack state will be just

like the one in figure 4.3.

Listing 4.2: Code for spilling to main memory the cached stack

```
1 // Computing the size of the cached stack
2 mfs    r5  = ss
3 mfs    r6  = st
4 sub    r27 = r5, r6
5 // Spilling the computed size to main memory
6 sspill  r27
```

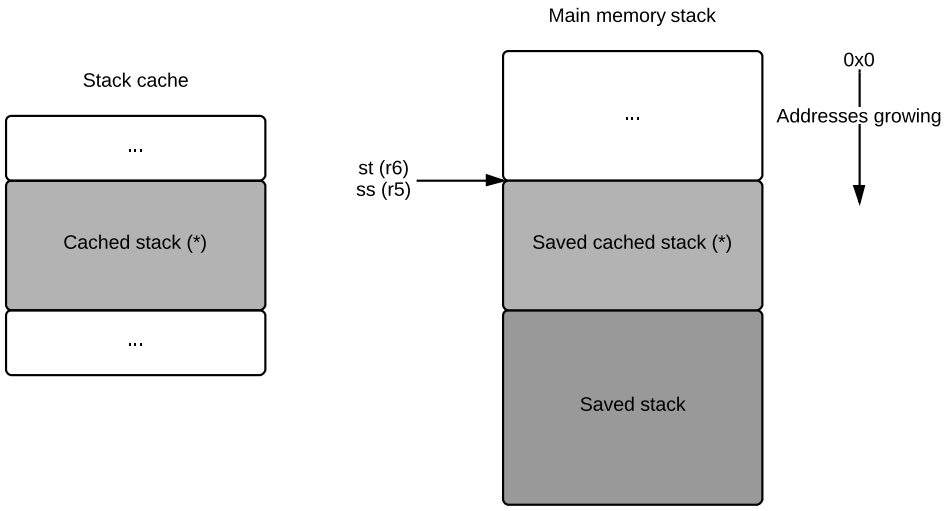
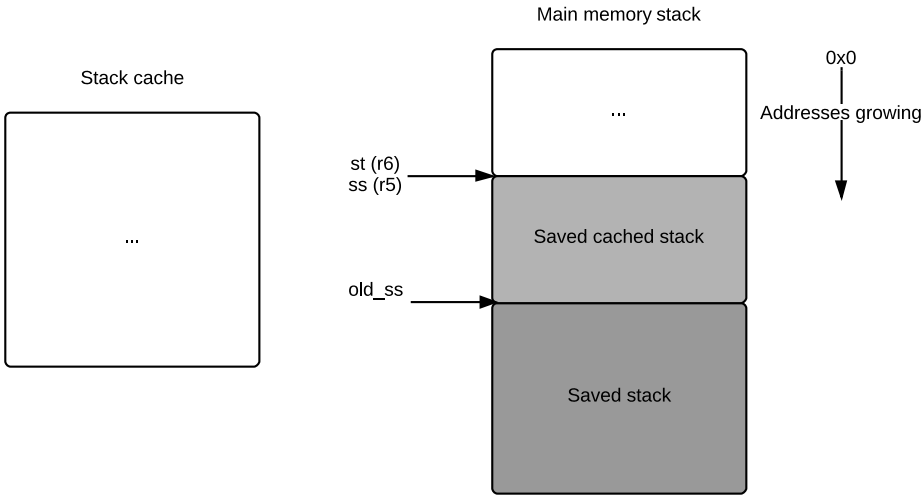


Figure 4.3: Stack cache and memory state after executing instructions 4.2

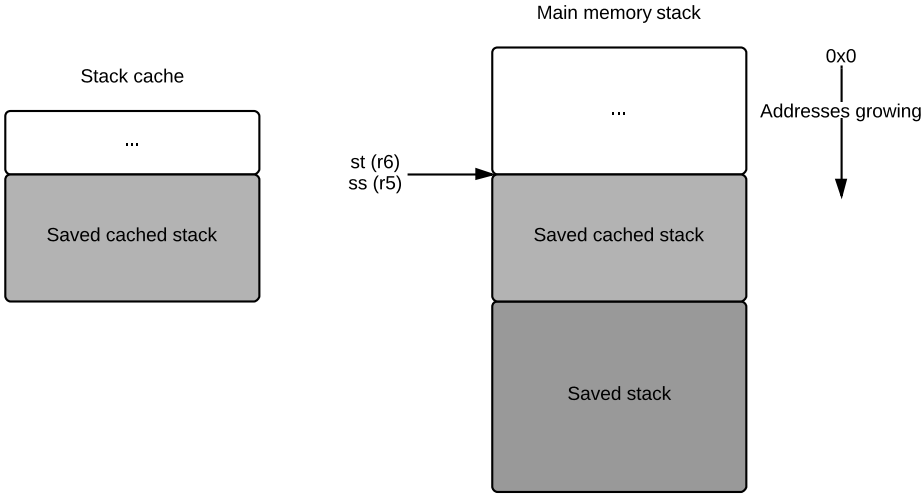
On a context switch the current executing thread must be saved but the re-entering thread's context must also be restored and this means restoring the stack cache state too. Even in this situation none of the three previously defined instructions can be helpful. What is needed is an ensure instruction capable of checking and eventually loading a dynamic amount of stack. A new `sens` instruction, accepting a register parameter has been defined. Its behaviour is just like the immediate ensure and is shown in figure 4.4.

Immediately after saving the current executing thread's stack cache the elected thread's stack cache state needs to be restored. To restore the stack cache the `sens` instruction is used in its register version. This instruction takes a register as a parameter and ensures that the specified amount from the top of the stack is contained in the stack cache. To restore the previous state the size of the stack

previously contained in the cache has to be known, this value can be computed as `old_ss - st`.



(a) State of a newly elected thread's stack and stack cache before `sens`



(b) State of the stack cache after `sens`

Figure 4.4: Changes to the stack cache after the execution of `sens rX` where `rX` contains the value `old_ss - st`

4.3 Memory Protection

A composable operating system has to offer timing and spatial separation between processes. Moreover, in modern real-time systems it is common to have processes of different integrity levels executing on the same processor, this makes the need of memory protection arise. In order to have a time-composable system and to guarantee memory protection between different integrity levels spatial separation is necessary.
Several ways of providing spatial separation exist, most of them rely on memory virtualization and the usage of a Memory Management Unit (MMU). Modern processors implement a MMU which is responsible for translating virtual to physical addresses in order to make processes access their own memory and avoid forbidden accesses. Commercial MMUs are often tuned on the average case, they make use of a software managed address cache called Translation Look-aside Buffer (TLB). A TLB is a fixed size software managed set of PTE (Page Table Entries), it is searched through a virtual address and allows to translate it to a physical address.

TLB lookups, just like normal caches lookups, can result in a hit or a miss. A successful lookup (hit) has a bounded WCET. On the other hand the miss WCET is really hard to compute since a complex pagetable data structure stored in memory has to be consulted.

Different solutions are adopted by modern real-time operating systems in the field of memory separation and protection, always trying to avoid the complexity of virtual address translation. [BA01] argues that two main approaches can be carried on:

1. Using the MMU translation system as it is
2. Disabling all the memory translation features and running all the processes in the same address space

TiCOS natively targets the PowerPC architecture and makes use of virtual address translation. In the case of Patmos no virtual memory support and address translation was available. Because of the lack of these functionalities all processes run in the same address space.

Often real-time operating system exploit only the permission functionalities of a MMU, avoiding the virtual-to-physical address translation, to provide mem-

ory protection ([SMB05] and [Car04]). Using only some MMU's functionalities, as shown in [BA01], can guarantee spatial separation maintaining the system analyzable. Processors providing a software programmable TLB (that is: with explicit PTEs load, such as PowerPC) allow to translate a certain range of virtual addresses without incurring in a TLB miss. If the virtual addresses of all pagetables are limited to this range the worst case execution time for a translation to physical address is limited by the time needed to search the TLB.
In the future Patmos processor may be extended with address range checks in order to have spatial separation between processes.

4.4 Explicit Supervisor Mode and Cache Invalidation

Modern processors offer a way to distinguish between at least two execution modes: user and supervisor mode. Supervisor mode allows to execute all instruction, even privileged ones, access a different address space and manipulate memory management hardware. The existence of these modes helps protecting the operating system data structures from corruption attempts made by user code. When supervisor mode is not active the CPU will not allow to access privileged memory areas or execute privileged instructions.

The switch between execution modes may be expensive in terms of computing time: when switching from supervisor to user mode it is recommended to flush and invalidate the caches in order to make sure the user code can not access something it is not meant to. Switching to supervisor mode is usually done when executing context switches and system calls.

Different processors provide different ways of switching to supervisor mode.
PowerPC processors [Fre05] have two levels of privilege: supervisor mode (used only by the operating system) and user mode (used by the operating system and application code). PowerPC architecture distinguishes between these two modes through the Machine State Register (MSR), a special purpose register holding processor's configuration setup: the PR bit of the MSR indicates the privilege level as follows:

- MSR[PR]=1: user mode
- MSR[PR]=0: supervisor mode

MSR[PR] is set to 0 (supervisor) every time an interrupt happens and an ISR is executed. The user code can explicitly switch to supervisor mode through the sc instruction which causes a System Call exception and the execution of the system call ISR.
An ARM processor [ARM12] has 7 different privilege levels:

- SVC (supervisor): entered when a system call instruction (SVC) is executed
- FIQ: high priority interrupts mode
- IRQ: normal priority interrupts mode
- Abort: entered when memory protection is violated
- Undef: entered when undefined instructions are used
- System: just like SVC mode but working with user mode registers
- User

The privilege level is stored in a field in the Current Program Status Register (CPSR), a special register holding both ALU's and processor's status. Modification to the priority level can be done by directly manipulating the CPSR. In order to access the OS services user code can explicitly switch to supervisor mode through the system call instruction (SVC).

As a future possible extension, the Patmos processor might be added a syscall instruction dedicated to switching to supervisor mode through an interrupt and defining a special register holding the machine state, including at least one bit to distinguish between user and supervisor mode.

Another possible extension may be the previously mentioned cache flushing and invalidation. Invalidating the caches when a context switch happens is a common practice in avionics when a partition switch happens since partitions do not have shared memory. Moreover cache invalidation can be useful to avoid interference between the current executing thread and the next elected one when preemption is enabled.

Patmos provides three kinds of caches: stack, method, and data cache. The stack cache has enough instruction to be handled precisely by the operating system, storing and restoring its content without any need for invalidation.

Chapter 5

TiCOS Extensions

The TiCOS operating system has been extended to support the Patmos architecture. The porting of the operating system has been carried on following an incremental development technique. Following the structure presented in Chapter 3 the incremental development followed the operating system layers order:

1. **kernel**: the kernel layer, divided itself in two layers, examined in the appearing order: **arch** and **core**
2. **ARINC API** and **library**

The incremental development technique allowed to start from the most architectural-dependent layer, understanding and modifying small functions which form together the core functionalities of the operating system. Moving to the next layer (the **core** one) all the underneath dependencies were already ported to the Patmos architecture and this let the development be more focused on core services of the operating system.

A common risk when porting an existing operating system to a different architecture is concentrating too much on the architectural-dependent code. This

attitude may lead to miss the perspective on the system as a whole. When working with notably different architectures, like Patmos, the porting is not only an assembly-translation but the core functionalities of an OS have to be inspected paying attention on identifying unnecessary operations and possible optimizations.

In the following the operating system's changes are analyzed as incrementally as possible. Some functionalities that are inherently cross layer (like context switching) are however explained in a dedicated section in order to achieve more clarity.

5.1 Architectural Changes

As shown is Section 3.5 the architectural layer of the operating system is compiled in a single link object (`arch.lo`). Since this layer is mainly made of assembly files not much code can be reused and for this reason a new directory named `arch/patmos` has been created containing all the Patmos-dependent code which is then compiled in `arch.lo`.

5.1.1 Clock

In Section 4.1 the new memory mapped real-time clock is presented. The RTC's registers are mapped to the local memory and therefore have to be accessed through local memory instructions.
A header file has been created defining a number of pre-compilation directives allowing to read and write RTC memory mapped registers, see listing 5.1 for details.

Listing 5.1: rtc.h contains directives to read and write memory mapped RTC registers

```
1 #define _IODEV __attribute__((address_space(1)))
2
3 typedef _IODEV unsigned int volatile * const _iodev_ptr_t;
4
5 #define __PATMOS_RTC_CYCLE_LOW_ADDR (0xF0000300)
6
7 #define __PATMOS_RTC_CYCLE_UP_ADDR (0xF0000304)
8
9 #define __PATMOS_RTC_TIME_LOW_ADDR (0xF0000308)
10
```

```
11 #define __PATMOS_RTC_TIME_UP_ADDR (0xF000030C)
12
13 #define __PATMOS_RTC_INTERVAL_ADDR (0xF0000310)
14
15 #define __PATMOS_RTC_ISR_ADDR (0xF0000314)
16
17 #define __PATMOS_RTC_RD_CYCLE_LOW(res) res = *((_iodev_ptr_t)
       __PATMOS_RTC_CYCLE_LOW_ADDR);
18
19 #define __PATMOS_RTC_RD_CYCLE_UP(res) res = *((_iodev_ptr_t)
       __PATMOS_RTC_CYCLE_UP_ADDR);
20
21 #define __PATMOS_RTC_RD_TIME_LOW(res) res = *((_iodev_ptr_t)
       __PATMOS_RTC_TIME_LOW_ADDR);
22
23 #define __PATMOS_RTC_RD_TIME_UP(res) res = *((_iodev_ptr_t)
       __PATMOS_RTC_TIME_UP_ADDR);
24
25 #define __PATMOS_RTC_WR_CYCLE_LOW(val) *((_iodev_ptr_t)
       __PATMOS_RTC_CYCLE_LOW_ADDR) = val;
26
27 #define __PATMOS_RTC_WR_CYCLE_UP(val) *((_iodev_ptr_t)
       __PATMOS_RTC_CYCLE_UP_ADDR) = val;
28
29 #define __PATMOS_RTC_WR_INTERVAL(interval) *((_iodev_ptr_t)
       __PATMOS_RTC_INTERVAL_ADDR) = interval;
30
31 #define __PATMOS_RTC_WR_ISR(address) *((_iodev_ptr_t)
       __PATMOS_RTC_ISR_ADDR) = address;
```

Some of the directives define addresses to the local memory:

- `_IODEV`: redefines `__attribute__((address_space(1)))` which allows to access different address spaces, in this case 1 indicates the local memory
- `_iodev_ptr_t`: type for a constant pointer to the local memory, allows to access memory mapped registers
- `__PATMOS_RTC_CYCLE_LOW_ADDR`: address for the lower 32 bits of the cycle counter
- `__PATMOS_RTC_CYCLE_UP_ADDR`: address for the upper 32 bits of the cycle counter
- `__PATMOS_RTC_TIME_LOW_ADDR`: address for the lower 32 bits of the time in microseconds register
- `__PATMOS_RTC_TIME_UP_ADDR`: address for the upper 32 bits of the time in microseconds register
- `__PATMOS_RTC_INTERVAL_ADDR`: address for the interval register

- `__PATMOS_RTC_ISR_ADDR`: address for the memory mapped register of the ISR's address

Other directives enable reading memory-mapped registers:

- `__PATMOS_RTC_RD_CYCLE_UP` and `__PATMOS_RTC_RD_CYCLE_LOW`: read upper and lower 32 bits of the cycle counter
- `__PATMOS_RTC_RD_TIME_UP` and `__PATMOS_RTC_RD_TIME_LOW`: read upper and lower 32 bits of the time in microseconds

And some directives allow to write values to the memory-mapped registers:

- `__PATMOS_RTC_WR_CYCLE_UP` and `__PATMOS_RTC_WR_CYCLE_LOW`: write upper and lower 32 bits of the cycle counter
- `__PATMOS_RTC_WR_INTERVAL`: writes to the memory-mapped interval register, the value is supposed to be a clock cycles number
- `__PATMOS_RTC_WR_ISR`: writes to the memory-mapped register holding the interval ISR address

The above directives are used in the time management functions of the operating system's architectural layer. These functions realize basic operative system's services such as: reading the time base, writing the time base, and setting the interrupt interval.

5.1.2 Thread's Context

The PowerPC architecture provides three types of registers [IBM98]:

- *volatile*: functions may modify these registers. The content of volatile register has not to be preserved
- *non-volatile*: functions must preserve their value. If a function modifies a non-volatile register it has to restore the previous content before returning
- *dedicated*: these registers have to be used only for their specific purpose

The original operating system defined two data structures holding a thread's context: `volatile_context_t` and `context_t`. The structures holding the context were originally allocated on the top of the thread's stack. The union of the two data structures is a multiple of a quadword since, according to the PowerPC EABI [IBM98], a stack frame has to be quadword aligned.

The thread context in Patmos is much simpler for several reasons:

- In Patmos there's no particular need to have different data structures for *volatile* (*caller-saved*) and *non-volatile* (*callee-saved*) registers
- The thread context in Patmos is more compact: as presented in Section 2.2 there are only 32 general purpose registers (32 bit), 16 special purpose registers (32 bit) and 8 predicate registers (1 bit). No floating point registers have to be saved
- Patmos stack do not impose any particular alignment to stack frames; furthermore, in Patmos thread's context is not placed on the stack

That said, the data structure holding the context is presented in listing 5.2.

Listing 5.2: C structure holding Patmos context

```
1  typedef struct
2  {
3    uint32_t r1;   /*  0 */
4    uint32_t r2;
5    uint32_t r3;   /*  8 */
6    uint32_t r4;
7    uint32_t r5;   /*  16  */
8    uint32_t r6;
9    uint32_t r7;   /*  24  */
10   uint32_t r8;
11   uint32_t r9;   /*  32  */
12   uint32_t r10;
13   uint32_t r11;  /*  40  */
14   uint32_t r12;
15   uint32_t r13;  /*  48  */
16   uint32_t r14;
17   uint32_t r15;  /*  56  */
18   uint32_t r16;
19   uint32_t r17;  /*  64  */
20   uint32_t r18;
21   uint32_t r19;  /*  72  */
22   uint32_t r20;
23   uint32_t r21;  /*  80  */
24   uint32_t r22;
25   uint32_t r23;  /*  88  */
```

```
26   uint32_t r24;
27   uint32_t r25; /*  96  */
28   uint32_t r26;
29   uint32_t r27; /*  104 */
30   uint32_t r28;
31   uint32_t r29; /*  112 */
32   uint32_t r30;
33   uint32_t r31; /*  120 */
34
35   uint32_t s0;
36   uint32_t s1;  /*  128 */
37   uint32_t s2;
38   uint32_t s3;  /*  136 */
39   uint32_t s4;
40   uint32_t s5;  /*  144 */
41   uint32_t s6;
42   uint32_t s7;  /*  152 */
43   uint32_t s8;
44   uint32_t s9;  /*  160 */
45   uint32_t s10;
46   uint32_t s11; /*  168 */
47   uint32_t s12;
48   uint32_t s13; /*  176 */
49   uint32_t s14;
50   uint32_t s15; /*  184 */
51
52   uint32_t ssize; /*  188 */
53 } context_t;
```

The Patmos context contains general-purpose caller-saved (scratch) registers (`r1-r19`), general-purpose callee-saved registers (`r20-r31`) and special purpose registers (`s0-s15`). Since `r0` always contains 0 by convention there's no need to save in in the context. In the Patmos processor the context of a thread is not only made of registers but even the stack cache status has to be taken into account. This is the reason for the field `ssize` in the context data structure: it holds the amount of thread's stack stored in the cache and it is used to restore the stack cache status after a context switch, details in Section 5.4.

5.1.3 Memory Management

The original operating system made use of PowerPC memory virtualization functionalities. As stated in Section 4.3 Patmos has no virtual memory capabilities so a new memory allocation mechanism has to be thought. In 4.3 address range checks have been presented as a possible extension for the Patmos processor, keeping this in mind the memory could be divided in parts each of which could be assigned a different privilege level. According to this a thread's context is no longer stored on the top of the thread's stack but a dedicated OS structure

has been allocated in memory, holding all the user application threads contexts. Using the memory protection technique this array of contexts could be stored in the protected memory area with kernel level privilege.

As presented in Chapter 2, Patmos not only supports a normal and cached stack but even a shadow stack dedicated to hold aliased data. Therefore each thread must benefit from allocated memory for both caches. Before inspecting the memory layout created by the operating system it is worth to take a look to how the system is inited and the partitions are created. First of all the kernel is booted using a kernel stack (cached) and a kernel shadow stack. During the booting phase each partition is loaded into memory and for each partition a *main thread* is created, responsible for starting all the partition's processes. Each partition's *main thread* owns a cached stack and a shadow stack. Each ARINC process is mapped to a thread and so is granted space for both stacks. To sum things up the memory to be allocated is:

- A cached stack and a shadow stack for the kernel
- A cached stack and a shadow stack for each partition's (*main thread*)
- A cached stack and a shadow stack for each partition's process

The memory for the kernel's stack is placed in the `bss` section and two labels are defined in order to access this memory (see listing 5.3).

Listing 5.3: Assembly labels for the kernel's stacks

```
1   .section ".bss", "aw"
2 pok_shadow_stack:
3     .space 8 * 1024
4     .globl pok_shadow_stack_end
5 pok_shadow_stack_end:
6
7 pok_stack:
8     .space 8 * 1024
9     .globl pok_stack_end
10 pok_stack_end:
```

The two defined labels are used to initialize stack pointer's register (`st`), shadow stack pointer's register (`r29`) and the stack cache spill address (`ss`) in the entry point function of the operating system, indicated in listing 5.4.

Listing 5.4: Assembly code for the entry point of the operating system

```
1     .globl      _pok_reset
```

```
2       .type       _pok_reset,@function
3       .size       _pok_reset, .Ltmp8-_pok_reset
4       .fstart     _pok_reset, .Ltmp8-_pok_reset, 4
5   _pok_reset:
6       li      $r1             = pok_stack_end
7       li      $r2             = pok_shadow_stack_end
8       mov     $r29            = $r1
9       mts     $st             = $r2
10      mts     $ss             = $r2
11      and     $r0             = $r0, 0x0
12
13      brcf    _pok_clear_bss
14      nop
15      nop
16      nop
17  .Ltmp8:
```

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

1. For each partition:
   (a) An amount of memory enough to hold all its threads stacks is allocated (`_end` pointer is moved forward)
   (b) From this memory some space is reserved for the *main thread*'s stacks
2. For each partition's user thread:
   (a) Some space is reserved in the partition's memory for stack and shadow stack
   (b) Some memory is allocated, starting from `_end` pointer) to hold the thread's context

The size of the kernel stacks is set by default while the size of the user threads stacks can be configured through two directives, as shown in table 5.1.

Threads stacks are placed in partition's memory starting from its bottom, first the cached stack is placed and then the shadow stack. The memory layout created by the operating system follows the structure depicted in figure 5.1.

Directive	Value
USER_STACK_SIZE	Configurable by the user
USER_SHADOW_STACK_SIZE	Configurable by the user
KERNEL_STACK_SIZE	8192
KERNEL_SHADOW_STACK_SIZE	8192

Table 5.1: Stack sizes for user and kernel threads

5.2 Core Changes

Some of the changes made to the architectural layer are reflected to the core layer. The core layer of the kernel is compiled into a library object `core.lo`, as stated in 3.5. Since most of the code of this layer did not change, it made no sense to create different directories for each architecture, therefore to implement the following changes two pre-compilation directives are used: `POK_ARCH_PPC` (indicates that the kernel is being compiled for PowerPC architecture) and `POK_ARCH_PATMOS` (indicates that the kernel is being compiled for Patmos architecture).

5.2.1 Bootloader

Section 3.5 showed how the partitions are compiled, archived together and then put in the kernel's executable. During the kernel's boot phase the original version of the operating system used to search the executable for the partitions code, the research was made exploiting an array of partition's sizes.

The partitions loading mechanism has been modified so as to make the system more flexible and extendable: when booting the OS calls the function `partition_init` responsible for creating the data structures holding all partitions and for loading each partition into memory, the partitions are loaded from the `UART`.
The operating system expects a specific stream of data for each partition, the format of the partition's stream is shown in figure 5.2.

The loading of a partition, given a stream structured like in figure 5.2, is made through two auxiliary functions: `read_uint32` and `read_data` which respectively load from the `UART` an unsigned integer and a specified amount of binary data, these functions are shown in listing 5.5; both return the number of read bytes.

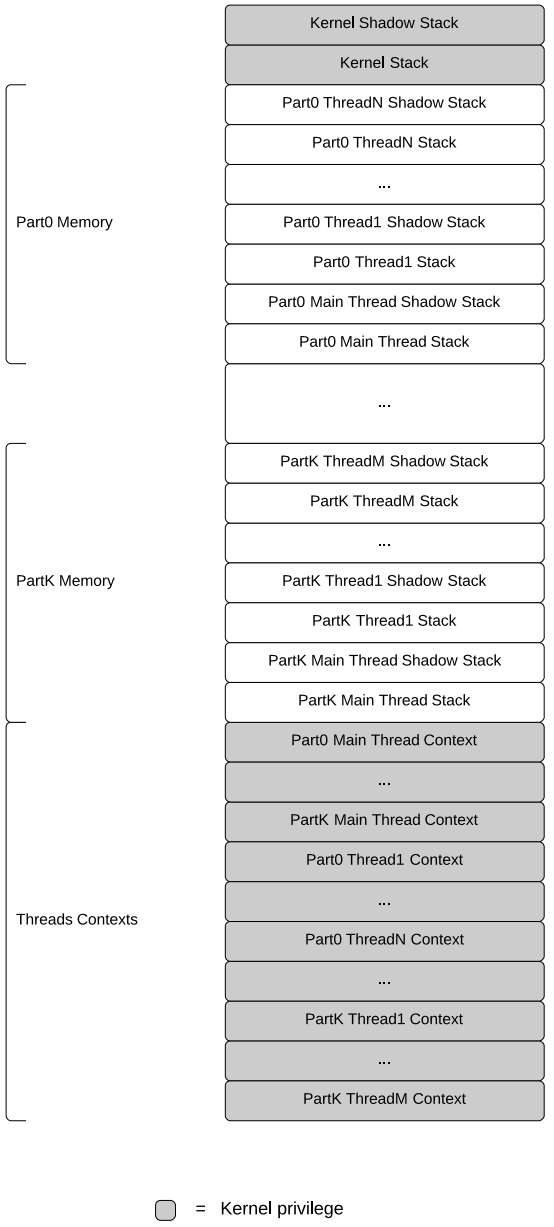


Figure 5.1: Memory layout created by the operating system

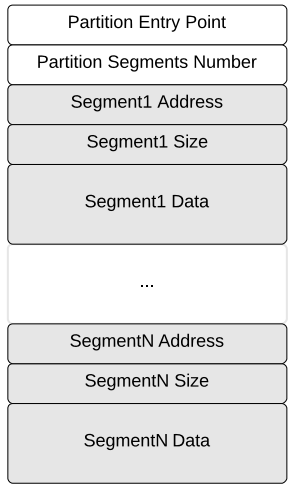


Figure 5.2: Format of the partition's stream expected by the operating system on the `UART`

Listing 5.5: Function to read an integer and a variable amount of data from the `UART`

```
1 static uint32_t read_uint32(uint32_t* ptr)
2 {
3   return uart_read((char*)(ptr), sizeof(uint32_t));
4 }
5
6 static uint32_t read_data(uint32_t size, char* ptr)
7 {
8   return uart_read(ptr, size);
9 }
```

A partition is loaded through the function `load_partition`. First of all the partition entry and the number of segments are read through the function `read_int`. Then for each segment address and size are read through `read_int`, after this an amount of data of the specified size is read using `read_data`. `load_partition`'s code can be seen in listing 5.6.

Listing 5.6: `load_partition` code

```
1 void pok_loader_load_partition (const uint32_t part_id, uint32_t *
      entry)
2 {
3   uint32_t part_entry;
4   uint32_t segments;
5
```

```
6   if(read_uint32(&part_entry) != sizeof(uint32_t))
7   {
8     pok_partition_error (part_id,
          POK_ERROR_KIND_PARTITION_CONFIGURATION);
9   }
10  if(read_uint32(&segments) != sizeof(uint32_t))
11  {
12    pok_partition_error (part_id,
          POK_ERROR_KIND_PARTITION_CONFIGURATION);
13  }
14
15  unsigned int segment = 0;
16  while (segment < segments)
17  {
18    uint32_t segment_address;
19    uint32_t segment_size;
20
21    if (read_uint32(&segment_address) != sizeof(uint32_t))
22    {
23      pok_partition_error (part_id,
            POK_ERROR_KIND_PARTITION_CONFIGURATION);
24    }
25    if (read_uint32(&segment_size) != sizeof(uint32_t))
26    {
27      pok_partition_error (part_id,
            POK_ERROR_KIND_PARTITION_CONFIGURATION);
28    }
29    if (read_data(segment_size, (char*)segment_address) !=
          segment_size)
30    {
31      pok_partition_error (part_id,
            POK_ERROR_KIND_PARTITION_CONFIGURATION);
32    }
33    segment++;
34  }
35  *entry = part_entry;
36 }
```

The partition entry point is then set as the entry for the partition's main thread and partition's segments are loaded into memory at the specified addresses.

5.2.1.1 elf2uart

`elf2uart` is a small program written in order to stream a Patmos executable to the simulator `UART`. The Patmos simulator can be configured to take a text file to be used as an input for `UART` reads, so `elf2uart` takes two files as parameters: the Patmos executable and the `UART` file.
`elf2uart` uses `libelf` to inspect the executable file and performs the following operations:

- Opens the executable file as a read file and the `UART` file as write file; lines 5 to 10 of listing 5.7
- Gets the ELF header; lines 21 to 23 of listing 5.7
- Gets the number of program's headers in the program's header table (`e_phnum` field); lines 25 to 27 of listing 5.7
- Gets the file entry point (`e_entry` field) from the ELF header and writes it to the `UART`; lines 29 to 32 of listing 5.7
- Computes the number of segments of type `PT_LOAD`. That is, segments to be loaded into memory; lines 36 to 48 of listing 5.7
- For each segment:
  - Segment address is written to the `UART`; lines 65 to 73 of listing 5.7
  - Segment size is written to the `UART`; lines 65 to 73 of listing 5.7
  - Segment data is written to the `UART`; lines 75 to 79 of listing 5.7

The function responsible for streaming a partition to the `UART` can be seen in listing 5.7.

Listing 5.7: Functions to stream an `ELF` file to the `UART` file

```
1  static void streamelf(const char* elf_filename, const char*
       uart_filename)
2  {
3    elf_version(EV_CURRENT);
4
5    int elf_file = open(elf_filename, O_RDONLY, 0);
6    assert(elf_file > 0);
7
8    FILE* uart_file;
9    uart_file = fopen(uart_filename, "w");
10   assert(uart_file);
11
12   Elf *elf = elf_begin(elf_file, ELF_C_READ, NULL);
13   assert(elf);
14
15   Elf_Kind ek = elf_kind(elf);
16   assert(ek == ELF_K_ELF);
17
18   int ec = gelf_getclass(elf);
19   assert(ec == ELFCLASS32);
20
21   GElf_Ehdr hdr;
22   GElf_Ehdr *tmphdr = gelf_getehdr(elf, &hdr);
23   assert(tmphdr);
24
```

```
25  size_t n, i;
26  int ntmp = elf_getphdrnum(elf, &n);
27  assert(ntmp == 0);
28
29  uint32_t entry = hdr.e_entry;
30
31  uint32_t big_entry = toBigEndian(entry);
32  fwrite(&big_entry, 4, 1, uart_file);
33
34  std::vector<GElf_Phdr> load_segments;
35
36  for(i = 0; i < n; i++)
37  {
38      GElf_Phdr phdr;
39      GElf_Phdr *phdrtmp = gelf_getphdr(elf, i, &phdr);
40      assert(phdrtmp);
41
42      if (phdr.p_type == PT_LOAD)
43      {
44      load_segments.push_back(phdr);
45    }
46  }
47
48  int segments_number = load_segments.size();
49
50  uint32_t big_segments_number = toBigEndian(segments_number);
51  fwrite(&big_segments_number, 4, 1, uart_file);
52
53  for(std::vector<GElf_Phdr>::iterator it = load_segments.begin();
        it != load_segments.end(); ++it)
54  {
55    GElf_Phdr phdr = *it;
56    assert(phdr.p_vaddr == phdr.p_paddr);
57    assert(phdr.p_filesz <= phdr.p_memsz);
58
59    char *buf = (char*)malloc(phdr.p_filesz);
60    assert(buf);
61
62    lseek(elf_file, phdr.p_offset, SEEK_SET);
63    read(elf_file, buf, phdr.p_filesz);
64
65    uint32_t start_offset = phdr.p_vaddr;
66    uint32_t size = phdr.p_filesz;
67    uint32_t total_size = phdr.p_memsz;
68
69    uint32_t big_start_offset = toBigEndian(start_offset);
70    uint32_t big_size = toBigEndian(size);
71
72    fwrite(&big_start_offset, sizeof start_offset, 1, uart_file);
73    fwrite(&big_size, sizeof size, 1, uart_file);
74
75    int j;
76    for(j=0; j < size; j++)
77    {
78      fwrite(&(buf[j]), 1, 1, uart_file);
```

```
79      }
80
81      free(buf);
82    }
83
84    elf_end(elf);
85    close(elf_file);
86    fclose(uart_file);
87  }
```

If `elf2uart` is executed on a *little-endian* architecture the read integers (sizes and addresses) are represented according to the *little-endian* format. Patmos processor manipulates *big-endian* data, so before streaming those integers to the `UART` they have to be converted to *big-endian* using the function in listing 5.8.

Listing 5.8: Functions converting *small-endian* integers to *big-endian* integers

```
1 static inline uint32_t toBigEndian(uint32_t value)
2 {
3   return ((value & 0xFF000000) >> 24) |
4                ((value & 0x00FF0000) >> 8) |
5                ((value & 0x0000FF00) << 8) |
6                ((value & 0xFF) << 24);
7 }
```

5.3 Library Changes

The ARINC and library layer allows the user code to call operating system services. This layer is not compiled with the kernel but guarantees access to kernel functionalities through the system call mechanisms. Since the services offered by the operating system to the user code did not change this layer remained almost the same apart from the way system calls are performed.

5.3.1 System Calls Implementation

A library layer allows to call kernel services from the user code. Clearly a way to jump safely to the kernel code must be provided. As stated in 4.4 architectures like PowerPC and ARM provide a special instruction able to perform a system call using the interrupts mechanism. In the original version of the operating system the dedicated PowerPC `sc` instruction was used to implement all the system call functions presented in Section 3.3.1, see listing 5.9.

Listing 5.9: PowerPC system call implementation

```
1       .globl pok_syscall2
2       .globl pok_syscall3
3       .globl pok_syscall4
4       .globl pok_syscall5
5       .globl pok_syscall6
6       .globl pok_syscall7
7  pok_syscall2:
8  pok_syscall3:
9  pok_syscall4:
10 pok_syscall5:
11 pok_syscall6:
12 pok_syscall7:
13      sc
14      brl
```

The Patmos processor does not provide a system call instruction. A normal `call` instruction can not be used since kernel symbols are not available when compiling user code. A special convention has to be adopted between kernel and library layers. Two solutions have been investigated:

- Placing each kernel service in a fixed and known location in memory, call it `serviceN_addr`, with each library's system call performing a immediate call to this address (`call serviceN_addr`)
- Placing in a dedicated and known memory location a system call service routine used by all system calls and able to dispatch to the proper kernel service

Clearly, the first proposed technique would have result in an unrealistic and non-extensible system. The second solution, on the other hand, is more realistic and resembles the way system calls are handled when using interrupts.

To implement the second solution the `system_call` function, shown in listing 5.10, is placed at location `0x900` in the `text` segment through the `.org 0x900 - 4` directive. As said in Section 2.1.1 each function's code is preceded by its size so, in order to place a method at location `0x900`, the `.org` directive has to consider the space dedicated to the function's size, this explains the given address of `0x900 - 4`.

Listing 5.10: Kernel `system_call` function

```
1    .globl    system_call
2    .type     system_call,@function
```

```
    .size      system_call, .Ltmp2-system_call
    .org     0x900 - 4
    .fstart    system_call, .Ltmp2-system_call, 4
system_call:
    sres   2
    sws    [1] = $r31
    sws    [0] = $r30

    li     $r30 = system_call
    call   pok_arch_sc_int
    nop
    nop
    nop

    sens   2
    lws    $r31 = [1]
    lws    $r30 = [0]
    sfree  2
    ret    $r30, $r31
    nop
    nop
    mov    $r1 = $r0
.Ltmp2:
```

`pok_arch_sc_int` is the function responsible for identifying the type of system call, retrieving all the parameters, and calling the proper function.

Through the linker script the kernel's `text` segment is placed at the address `0x1C0000`. When running the system the kernel's `system_call` function is going to be located at the address `0x1C0900`. That said, library's system calls are implemented via explicit calls to `0x1C0900`, as shown in listing 5.11.

Listing 5.11: Library's system call implementation

```
#define SYS_CALL_ADDR    0x1C0900
#define NOT_USED(x)    ((void)(x))
uint32_t (*syscall)(void) = (uint32_t (*)(void)) SYS_CALL_ADDR;
pok_ret_t pok_syscall1  (pok_syscall_id_t syscall_id,
             uint32_t arg1)
{
  NOT_USED(syscall_id);
  NOT_USED(arg1);
   return syscall();
}
pok_ret_t pok_syscall2  (pok_syscall_id_t syscall_id,
             uint32_t arg1,
             uint32_t arg2)
{
  NOT_USED(syscall_id);
  NOT_USED(arg1);
  NOT_USED(arg2);
```

```
18   return syscall();
19 }
20 pok_ret_t pok_syscall3 (pok_syscall_id_t syscall_id,
21              uint32_t arg1,
22              uint32_t arg2,
23              uint32_t arg3)
24 {
25   NOT_USED(syscall_id);
26   NOT_USED(arg1);
27   NOT_USED(arg2);
28   NOT_USED(arg3);
29   return syscall();
30 }
31 pok_ret_t pok_syscall4 (pok_syscall_id_t syscall_id,
32              uint32_t arg1,
33              uint32_t arg2,
34              uint32_t arg3,
35              uint32_t arg4)
36 {
37   NOT_USED(syscall_id);
38   NOT_USED(arg1);
39   NOT_USED(arg2);
40   NOT_USED(arg3);
41   NOT_USED(arg4);
42   return syscall();
43 }
44 pok_ret_t pok_syscall5 (pok_syscall_id_t syscall_id,
45              uint32_t arg1,
46              uint32_t arg2,
47              uint32_t arg3,
48              uint32_t arg4,
49              uint32_t arg5)
50 {
51   NOT_USED(syscall_id);
52   NOT_USED(arg1);
53   NOT_USED(arg2);
54   NOT_USED(arg3);
55   NOT_USED(arg4);
56   NOT_USED(arg5);
57   return syscall();
58 }
59 pok_ret_t pok_syscall6 (pok_syscall_id_t syscall_id,
60              uint32_t arg1,
61              uint32_t arg2,
62              uint32_t arg3,
63              uint32_t arg4,
64              uint32_t arg5,
65              uint32_t arg6)
66 {
67   NOT_USED(syscall_id);
68   NOT_USED(arg1);
69   NOT_USED(arg2);
70   NOT_USED(arg3);
71   NOT_USED(arg4);
72   NOT_USED(arg5);
```

```
73   NOT_USED(arg6);
74   return syscall();
75 }
76 pok_ret_t pok_syscall7 (pok_syscall_id_t  syscall_id,
77               uint32_t arg1,
78               uint32_t arg2,
79               uint32_t arg3,
80               uint32_t arg4,
81               uint32_t arg5,
82               uint32_t arg6,
83               uint32_t arg7)
84 {
85   NOT_USED(syscall_id);
86   NOT_USED(arg1);
87   NOT_USED(arg2);
88   NOT_USED(arg3);
89   NOT_USED(arg4);
90   NOT_USED(arg5);
91   NOT_USED(arg6);
92   NOT_USED(arg7);
93   return syscall();
94 }
```

5.4 Context Switch

Patmos context switching can be interrupt-driven: TiCOS configures the real-time clock interrupt interval to expire at specified points where scheduling decisions have to be taken and the current executing thread's context has to be switched, for example when a time slice ends and another partition has to execute.
A context switch can be also caused by the run-to-completion semantics: when a periodic thread ends its period or when a sporadic thread waits for an event to be "up".

5.4.1 Interrupt-driven Context Switching

When an interval interrupt is raised the control flow jumps to an interval interrupt service routine. The ISR defined by the operating system whose role is to perform the context switch is called `_interval_ISR` and is shown in listing 5.12.
The Patmos RTC was presented in Section 4.1: one of its memory mapped registers allows to set the address of the interval ISR. When the system boots the operating system's entry point (in listing 5.4) calls the booting function

`pok_boot` responsible for configuring the system before starting user's applications. `pok_boot` initializes the underlying hardware through the function `pok_arch_init` which uses the function `__PATMOS_RTC_WR_ISR` to set the address of the ISR.

The modified TiCOS maintains a pointer to the current executing thread's context, `current_context`, which is used to save the current thread's status. After performing scheduling decisions `current_context` points to the thread elected for execution (possibly the same), so it can be used to restore the elected thread's context.

Listing 5.12: Function `_interval_ISR`: the interval interrupt service routine

```
1     .globl    _interval_ISR
2     .type     _interval_ISR,@function
3     .size     _interval_ISR, .Ltmp3-_interval_ISR
4     .fstart   _interval_ISR, .Ltmp3-_interval_ISR, 4
5 _interval_ISR:
6     and   $r0           = $r0, 0x0
7
8     sub   $r29          = $r29, 4
9     swm   [$r29 + 0]    = $r1
10    li    $r1           = pok_current_context
11    lwc   $r1           = [$r1 + 0]
12
13    swm   [$r1 + 1]     = $r2
14    swm   [$r1 + 2]     = $r3
15    swm   [$r1 + 3]     = $r4
16    swm   [$r1 + 4]     = $r5
17    swm   [$r1 + 5]     = $r6
18    swm   [$r1 + 6]     = $r7
19    swm   [$r1 + 7]     = $r8
20    swm   [$r1 + 8]     = $r9
21    swm   [$r1 + 9]     = $r10
22    swm   [$r1 + 10]    = $r11
23    swm   [$r1 + 11]    = $r12
24    swm   [$r1 + 12]    = $r13
25    swm   [$r1 + 13]    = $r14
26    swm   [$r1 + 14]    = $r15
27    swm   [$r1 + 15]    = $r16
28    swm   [$r1 + 16]    = $r17
29    swm   [$r1 + 17]    = $r18
30    swm   [$r1 + 18]    = $r19
31    swm   [$r1 + 19]    = $r20
32    swm   [$r1 + 20]    = $r21
33    swm   [$r1 + 21]    = $r22
34    swm   [$r1 + 22]    = $r23
35    swm   [$r1 + 23]    = $r24
36    swm   [$r1 + 24]    = $r25
37    swm   [$r1 + 25]    = $r26
38    swm   [$r1 + 26]    = $r27
39    swm   [$r1 + 27]    = $r28
```

```
40
41      add     $r29            = $r29, 4
42      swm     [$r1 + 28]      = $r29
43      swm     [$r1 + 29]      = $r30
44      swm     [$r1 + 30]      = $r31
45
46      mfs     $r5             = $s5
47      mfs     $r6             = $s6
48      sub     $r2             = $r5, $r6
49      sspill $r2
50      swm     [$r1 + 47]      = $r2
51
52      mfs     $r2             = $s0
53      swm     [$r1 + 31]      = $r2
54      mfs     $r2             = $s1
55      swm     [$r1 + 32]      = $r2
56      mfs     $r2             = $s2
57      swm     [$r1 + 33]      = $r2
58      mfs     $r2             = $s3
59      swm     [$r1 + 34]      = $r2
60      mfs     $r2             = $s4
61      swm     [$r1 + 35]      = $r2
62      mfs     $r2             = $s5
63      swm     [$r1 + 36]      = $r2
64      mfs     $r2             = $s6
65      swm     [$r1 + 37]      = $r2
66      mfs     $r2             = $s7
67      swm     [$r1 + 38]      = $r2
68      mfs     $r2             = $s8
69      swm     [$r1 + 39]      = $r2
70      mfs     $r2             = $s9
71      swm     [$r1 + 40]      = $r2
72      mfs     $r2             = $s10
73      swm     [$r1 + 41]      = $r2
74      mfs     $r2             = $s11
75      swm     [$r1 + 42]      = $r2
76      mfs     $r2             = $s12
77      swm     [$r1 + 43]      = $r2
78      mfs     $r2             = $s13
79      swm     [$r1 + 44]      = $r2
80      mfs     $r2             = $s14
81      swm     [$r1 + 45]      = $r2
82      mfs     $r2             = $s15
83
84      sub     $r29            = $r29, 4
85      lwc     $r2             = [$r29 + 0]
86      add     $r29            = $r29, 4
87      swm     [$r1 + 0]       = $r2
88
89      li      $r29            = pok_stack_end
90      mts     $s6             = $r29
91      mts     $s5             = $r29
92      li      $r29            = pok_shadow_stack_end
93
94      li      $r30            = _interval_ISR
```

```
95      call  pok_arch_decr_int
96      nop
97      nop
98      nop
99
100     brcf  restore_context
101     nop
102     nop
103     nop
104
105 .Ltmp3:
```

`_interval_ISR` is an assembly routine performing the following operations:

Line 6	`r0` is assured to be set to 0
Lines 8-11	To bootstrap the context switch a register has to be freed in order to hold the `current_context` value. The shadow stack is used to hold the value of register `r1` and then the value of `current_context`'s pointer is loaded into `r0`
Lines 12-39	General purpose registers from `r2` to `r28` are saved starting from the next word pointed by `current_context` according to the structure of `context_t` presented in Section 5.1.2
Lines 41-44	`r29` is reset to its original value (after having stored `r1` value) and is stored to the correct memory locations pointed by `current_context`
Lines 46-50	The size of the thread's stack stored in the cache (and not consistent with the main memory, as explained in Section 2.1.3) is computed by subtracting to the stack cache spill address the stack pointer. This value is used to spill all the cached stack to the main memory. The size of the spilled stack is then saved in the right position in the context structure pointed by `current_context`
Lines 52-82	For each special register from `s0` to `s15` its content is saved to `r2` and then saved to the context pointer by `current_context`
Lines 84-87	The content of **r2**, previously stored on the shadow stack, is retrieved and saved in the context; then the original value of the shadow stack pointer (`r29`) is restored
Lines 89-92	In order to prepare the execution of the kernel, the kernel's stack pointer is loaded into registers `r5` and `r6` and kernel's shadow stack pointer is loaded into register `r29`
Lines 94-98	The function `pok_arch_decr_int` is called
Lines 100-103	A branch is performed to `restore_context`

The control flow started by an interval interrupt is depicted in figure 5.3.

- `_interval_ISR`: saves registers to `current_context` and switches to kernel stack and kernel stack cache
- `pok_arch_decr_int`: sets the next timer interval and calls the scheduler
- `pok_sched`: performs scheduling decisions and updates `current_context` to make it point to the context of the thread elected for execution
- `restore_context`: registers are restored from `current_context` and the function returns from the interrupt

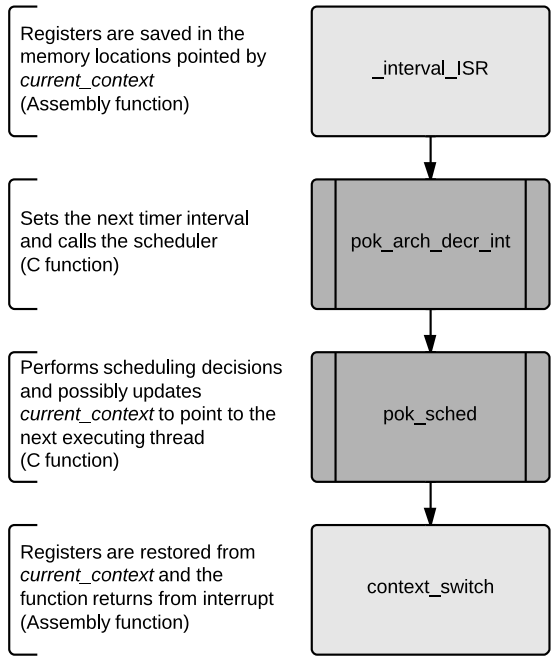


Figure 5.3: Flow of function calls needed to perform a context switch

Listing 5.13: Function `restore_context` restores context of the new selected thread

```
1     .globl    restore_context
2     .type     restore_context,@function
```

```
3       .size       restore_context, .Ltmp3-restore_context
4       .fstart     restore_context, .Ltmp3-restore_context, 4
5   restore_context:
6       and     $r0         = $r0, 0x0
7
8       li      $r1         = pok_current_context
9       lwc     $r1         = [$r1 + 0]
10
11      lwc     $r3         = [$r1 + 2]
12      lwc     $r4         = [$r1 + 3]
13      lwc     $r5         = [$r1 + 4]
14      lwc     $r6         = [$r1 + 5]
15      lwc     $r7         = [$r1 + 6]
16      lwc     $r8         = [$r1 + 7]
17      lwc     $r9         = [$r1 + 8]
18      lwc     $r10        = [$r1 + 9]
19      lwc     $r11        = [$r1 + 10]
20      lwc     $r12        = [$r1 + 11]
21      lwc     $r13        = [$r1 + 12]
22      lwc     $r14        = [$r1 + 13]
23      lwc     $r15        = [$r1 + 14]
24      lwc     $r16        = [$r1 + 15]
25      lwc     $r17        = [$r1 + 16]
26      lwc     $r18        = [$r1 + 17]
27      lwc     $r19        = [$r1 + 18]
28      lwc     $r20        = [$r1 + 19]
29      lwc     $r21        = [$r1 + 20]
30      lwc     $r22        = [$r1 + 21]
31      lwc     $r23        = [$r1 + 22]
32      lwc     $r24        = [$r1 + 23]
33      lwc     $r25        = [$r1 + 24]
34      lwc     $r26        = [$r1 + 25]
35      lwc     $r27        = [$r1 + 26]
36      lwc     $r28        = [$r1 + 27]
37      lwc     $r29        = [$r1 + 28]
38      lwc     $r30        = [$r1 + 29]
39      lwc     $r31        = [$r1 + 30]
40
41      lwm     $r2         = [$r1 + 47]
42      sens    $r2
43
44      lwc     $r2         = [$r1 + 31]
45      mts     $s0         = $r2
46      lwc     $r2         = [$r1 + 32]
47      mts     $s1         = $r2
48      lwc     $r2         = [$r1 + 33]
49      mts     $s2         = $r2
50      lwc     $r2         = [$r1 + 34]
51      mts     $s3         = $r2
52      lwc     $r2         = [$r1 + 35]
53      mts     $s4         = $r2
54      lwc     $r2         = [$r1 + 36]
55      mts     $s5         = $r2
56      lwc     $r2         = [$r1 + 37]
57      mts     $s6         = $r2
```

```
58      lwc     $r2         = [$r1 + 38]
59      mts     $s7         = $r2
60      lwc     $r2         = [$r1 + 39]
61      mts     $s8         = $r2
62      lwc     $r2         = [$r1 + 41]
63      mts     $s10        = $r2
64      lwc     $r2         = [$r1 + 42]
65      mts     $s11        = $r2
66      lwc     $r2         = [$r1 + 43]
67      mts     $s12        = $r2
68      lwc     $r2         = [$r1 + 44]
69      mts     $s13        = $r2
70      lwc     $r2         = [$r1 + 45]
71      mts     $s14        = $r2
72      lwc     $r2         = [$r1 + 46]
73      mts     $s15        = $r2
74
75      lwc     $r2         = [$r1 + 40]
76      mts     $s9         = $r2
77
78      brcf    $r2
79      lwc     $r2         = [$r1 + 1]
80      lwc     $r1         = [$r1 + 0]
81      nop
82
83  .Ltmp3:
```

`restore_context` function, shown in listing 5.13, performs the following operations:

Line 6	`r0` is assured to be set to 0
Line 8-9	`current_context` pointer value is loaded into register `r1`
Lines 11-39	All the general purpose registers are restored from the context pointed by `current_context`
Lines 41-42	The size of the stack previously hold in the cache is loaded from the context into `r2`, `r2` is then used to restore the stack into the cache though the `ensure` instruction
Lines 44-73	For each special register from `s0` to `s15` its content is loaded from the context into `r2` and then restored
Lines 75-76	As stated in Section 4.1, when an interval interrupt happens the current program counter is stored into register `s9` and represents the address to which the control has to return after the interrupt handling. So the previous content of `s9` loaded into `r2` and restored
Lines 78-81	The control flow jumps to the location pointed by `r2` (return address of the interrupt) and finally `r2` and `r1` are restored

5.4.2 Explicit Context Switching

As previously stated a context switch is explicitly requested when the period of a periodic task ends or when a sporadic task waits for an event. In this cases the `pok_context_switch` function is explicitly called. This function is called after a scheduling decision is taken and takes as a parameter the address of the context of the previously executing thread. `pok_context_switch` code is shown in listing 5.14.

Listing 5.14: Function `context_switch`: saves the context to the location pointed by `r3`

```
1     .globl    pok_context_switch
2     .type     pok_context_switch,@function
3     .size     pok_context_switch, .Ltmp6-pok_context_switch
4     .fstart   pok_context_switch, .Ltmp6-pok_context_switch, 4
5  pok_context_switch:
6     and   $r0           = $r0, 0x0
7
8     swm   [$r3 + 0]     = $r1
9     swm   [$r3 + 1]     = $r2
10    swm   [$r3 + 2]     = $r3
11    swm   [$r3 + 3]     = $r4
12    swm   [$r3 + 4]     = $r5
13    swm   [$r3 + 5]     = $r6
14    swm   [$r3 + 6]     = $r7
15    swm   [$r3 + 7]     = $r8
16    swm   [$r3 + 8]     = $r9
17    swm   [$r3 + 9]     = $r10
18    swm   [$r3 + 10]    = $r11
19    swm   [$r3 + 11]    = $r12
20    swm   [$r3 + 12]    = $r13
21    swm   [$r3 + 13]    = $r14
22    swm   [$r3 + 14]    = $r15
23    swm   [$r3 + 15]    = $r16
24    swm   [$r3 + 16]    = $r17
25    swm   [$r3 + 17]    = $r18
26    swm   [$r3 + 18]    = $r19
27    swm   [$r3 + 19]    = $r20
28    swm   [$r3 + 20]    = $r21
29    swm   [$r3 + 21]    = $r22
30    swm   [$r3 + 22]    = $r23
31    swm   [$r3 + 23]    = $r24
32    swm   [$r3 + 24]    = $r25
33    swm   [$r3 + 25]    = $r26
34    swm   [$r3 + 26]    = $r27
35    swm   [$r3 + 27]    = $r28
36    swm   [$r3 + 28]    = $r29
```

```
37      swm     [$r3 + 29]      = $r30
38      swm     [$r3 + 30]      = $r31
39
40      mfs     $r5             = $s5
41      mfs     $r6             = $s6
42      sub     $r2             = $r5, $r6
43      sspill $r2
44      swm     [$r3 + 47]      = $r2
45
46      add     $r2             = $r30, $r31
47      mts     $s9             = $r2
48
49      mfs     $r2             = $s0
50      swm     [$r3 + 31]      = $r2
51      mfs     $r2             = $s1
52      swm     [$r3 + 32]      = $r2
53      mfs     $r2             = $s2
54      swm     [$r3 + 33]      = $r2
55      mfs     $r2             = $s3
56      swm     [$r3 + 34]      = $r2
57      mfs     $r2             = $s4
58      swm     [$r3 + 35]      = $r2
59      mfs     $r2             = $s5
60      swm     [$r3 + 36]      = $r2
61      mfs     $r2             = $s6
62      swm     [$r3 + 37]      = $r2
63      mfs     $r2             = $s7
64      swm     [$r3 + 38]      = $r2
65      mfs     $r2             = $s8
66      swm     [$r3 + 39]      = $r2
67
68      mfs     $r2             = $s9
69      swm     [$r3 + 40]      = $r2
70      mfs     $r2             = $s10
71      swm     [$r3 + 41]      = $r2
72      mfs     $r2             = $s11
73      swm     [$r3 + 42]      = $r2
74      mfs     $r2             = $s12
75      swm     [$r3 + 43]      = $r2
76      mfs     $r2             = $s13
77      swm     [$r3 + 44]      = $r2
78      mfs     $r2             = $s14
79      swm     [$r3 + 45]      = $r2
80      mfs     $r2             = $s15
81      swm     [$r3 + 46]      = $r2
82
83      brcf    restore_context
84      nop
85      nop
86      nop
87  .Ltmp6:
```

`pok_context_switch`, just like the interval ISR, saves the current context to the location pointed by `r3` register (which holds the function's argument). In

order to restore the context later the return address, hold in `r30` and `r31`, has to be saved into `s9` (lines 46-47). The context of the thread elected for execution is pointed by `current_context` and restored by the function `restore_context`.

Chapter 6

Source Code Access

The operating system realized is open source and is published at the address:

`https://github.com/t-crest/ospat`

The directory structure of the published project is the following:

```
.
|-- elf2uart
|   `-- src
|-- examples
|   `-- arinc653-1event-01
|-- kernel
|   |-- arch
|   |-- core
|   |-- include
|   `-- middleware
|-- libpok
|   |-- arch
|   |-- arinc653
|   |-- core
|   |-- include
```

```
|   `-- middleware
`-- misc
    |-- ldscripts
    `-- mk
```

- `elf2uart`: contains the source code for the program streaming Patmos ELFs to the UART (see Section 5.2.1.1)
- `example`: contains source code and makefiles for a sample application
- `kernel`: contains source code of the OS kernel, `arch`, `core` and `middleware` contain the corresponding OS layers while `include` directory contains header files
- `libpok`: holds the OS library's source code
- `misc`: contains linker scripts (`ldscripts`) and make rules (`mk`)

The Patmos simulator's code can be found at the address:

`https://github.com/t-crest/patmos/tree/master/simulator`

6.1 Running an Example

As stated before, the `example` directory contains the source code of a sample application. The example code is structured as follows:

```
.
|-- cpu
|   |-- kernel
|   |   |-- deployment.h
|   |   `-- Makefile
|   |-- Makefile
|   `-- part1
|       |-- activity.c
|       |-- activity.h
|       |-- deployment.h
|       |-- main.c
|       `-- Makefile
`-- Makefile
```

The `makefile` in the root directory allows to build both the kernel and the partitions (in this example there is only one partition). The `kernel` directory contains a `makefile` to build the kernel and `deployment.h` file contains pre-compilation directives used to configure kernel's compilation. The `part1` contains the partition's code with `main.c` defining the partition's entry point and `activity.c` and `activity.h` contain partition's threads definition.

To run the sample application on the Patmos simulator, the script in listing 6.1 can be run in the OS root directory.

Listing 6.1: Bash script to run the sample application

```
1  export ARCH=patmos
2  export POK_PATH=`pwd`
3
4  cd ./examples/arinc653-1event-O1/generated-code
5  make
6
7  cd $POK_PATH/elf2uart/
8  mkdir build
9  cd build
10 cmake ..
11 make
12
13 cd $POK_PATH
14 mkdir deploy
15
16 mv examples/arinc653-1event-O1/generated-code/cpu/pok.elf deploy/
       kernel.elf
17 mv examples/arinc653-1event-O1/generated-code/cpu/part1/part1.elf
       deploy/part1.elf
18 mv elf2uart/build/elf2uart deploy/elf2uart
19
20 cd deploy
21 touch deploy.uart
22 ./elf2uart part1.elf --output=deploy.uart
23
24 pasim --in=deploy.uart --interrupt=1 kernel.elf
```

Lines 1-2	Defines variables used in the build chain: path to the OS and target architecture
Lines 4-5	Moves to the example's directory and starts the compilation
Lines 7-11	Moves to the `elf2uart` directory and compiles the `elf2uart` utility
Lines 13-14	Moves back to the root directory and creates the `deploy` directory to hold the example executables

Lines 16-18	Moves kernel, partition and `elf2uart` executables to the `deploy` directory
Lines 20-22	Moves to the `deploy` directory, creates a file to simulate the UART and streams the partition executable to the UART using `elf2uart`
Lines 24	Runs the example

Chapter 7

Conclusions

This chapter has three main purposes: summarize what I learned, point out the main contributions, and suggest possible future works.

7.1 Personal Knowledge

The skills I developed working on the project can be summarized in two main fields: *technical* and *methodological*.
From a *technical* point of view I learned several things in the different stages of the project:

- Studying the T-CREST architecture I understood the problems arising from the development of a time-predictable multi-core architecture and several valuable solutions for solving them
- Studying the original operating system I figured out what an OS requires from the underlying hardware and what a RTOS has to implement in order to support hard real-time tasks
- Developing the OS I learned Patmos assembly programming and I got in touch with advanced features of the C programming language and compiler

From a *methodological* perspective I learned how to work in a research project. Patmos processor, simulator and compiler are research work and therefore they are always changing and evolving. I learned how important is knowing what the people you work with are doing in order not to waste time. Moreover I learned how talking to other people in the research group helps to spread the knowledge and to see problems from different perspectives, which make them easier to be solved.

7.2 Main Contributions

The project work resulted in two main contributions:

- TiCOS operating system has been ported to the Patmos processor
- The Patmos processor, simulator and compiler are research works, part of the T-CREST project. As research projects they are continuously evolving. The development of an operating system for the Patmos processor allowed to identify missing requirements for the architecture (detailed in chapter 4) in order to support more complex computational models. Moreover the development itself started internal discussions on other possible extensions of the processor, such as ways of implementing interrupts and memory protection
- The T-CREST mission is to build a time-predictable multi-processor system able to simplify the analysis and guarantee better performances [DTU12c]. In order to create such a system, hardware, simulator and compiler have been developed and studied. An operating system for Patmos takes T-CREST a step further in the direction of a multi-processor, time-predictable and easily-analyzable platform made of hardware, compiler, OS and libraries

Other minor contributions follow:

- A considerable amount of code used for testing T-CREST simulator and compiler has been developed, resulting in the identification and resolution of several bugs
- Even if not fulfilling all the T-CREST's requirements, the operating system's implementation is a solid starting point for further extensions and research works

7.3 Suggested Future Works

More work could be made both extending the operating system's functionalities and testing them.

- As previously stated, T-CREST's aim is to create a multi-processor system. The developed operating system targets a single Patmos processor. A future extension could be making the operative system support multi-processors. As a starting point an approach similar to the one adopted by CompOSe could be used: having a different and independent instance of the OS running on each processor [HEM+11]
- As soon as the hardware is extended with RTC and interrupts features presented in Section 4, the operating system should be tested on it in order to evaluate its performances. Particular attention should be paid to context switching delay times in order to analyze the influence of stack cache spill and restore operations

As stated before, the hardware was not completely ready to support the execution of the OS and this is the reason why OS performances have not been measured and no functional evaluation has been performed. Moreover, Patmos is a notably innovative architecture, the developed software is the first example of operating system running on the Patmos processor. Therefore a critical comparison of performances and functionalities between the developed OS and other solutions running on the same architecture was simply impossible.
Implementing the proposed extensions in the hardware and porting other operating systems to the Patmos architecture would be a good starting point for a future evaluation of the realized OS.

Appendix A

Software Simulator of Patmos

The Patmos simulator [DTU12a] is a C++ implementation made of classes and interfaces representing processor's component and is designed to be modular and extensible. A class diagram for the Patmos simulator can be seen in figure A.1.

`simulator_t` is the main class of the simulator and controls the whole simulation. This main class must be connected to the components of the processor and so we have:

- `decoder_t`: is the component responsible for the instruction decode. The decoder translates ISA instruction and translates them to simulator instructions.
- `memory_t`: abstract class for the memory, specifies a general behavior a memory should expose such as read and write operations. This abstract class is implemented by two other classes: `ideal_memory_t` and `fixed_delay_memory_t`. The simulator uses this class to reference both processor's global and local memory.
- `register_file_t`: template class which offers methods for getting and setting each register in a group of registers. Allows to implement predicate

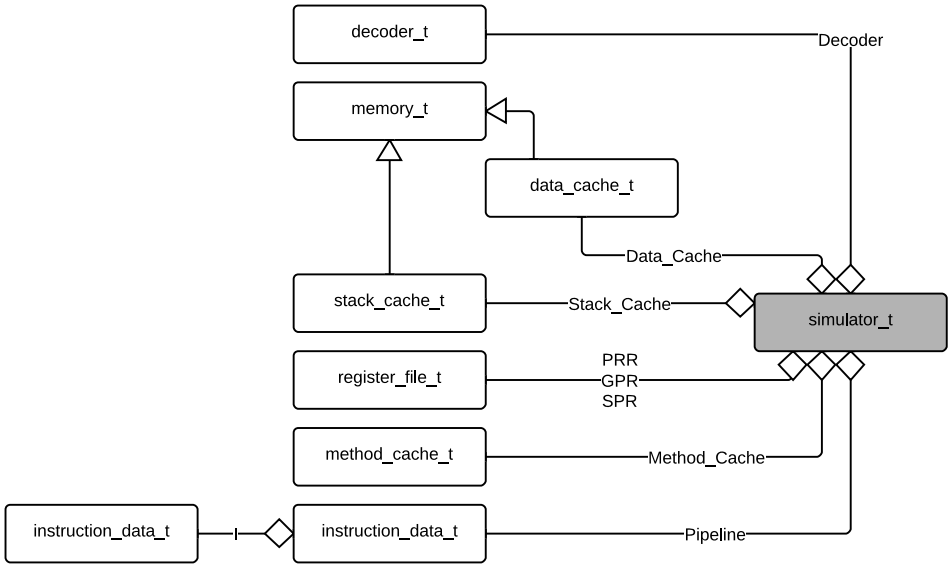


Figure A.1: Class diagram for the Patmos software simulator

register (`PGR`), general purpose registers (`GPR`) e special purpose registers (`SPR`).

- `data_cache_t`: another abstract class extending `memory_t`, is the root of a hierarchy made of two other subclasses: `ideal_data_cache_t` and `lru_data_cache_t`. Both those classes implement a type of data cache so both wrap an other memory object to cache.

- `stack_cache_t`: another abstract class extending `memory_t`, is the root of a hierarchy made of two other subclasses: `ideal_stack_cache_t` and `block_stack_cache_t`. Both those classes implement a type of stack cache and both wrap the global memory.

`instruction_t` is another important class which wraps the concept of a processor instruction. The class offers a method for each pipeline stage (`IF`, `DR`, `EX`, `MW`) which actually executes the corresponding pipeline stage. `instruction_data_t` class wraps an instruction and its operands. The simulator pipeline is nothing more than a bidimensional array of `NUM_STAGES x NUM_SLOTS` of `instruction _data_t` objects, where `NUM_STAGES` is the number of processor pipeline's stages and `NUM_SLOTS` is the number of slots in each bundle.

The simulation is handled by a loop in the `simulator_t` class. At each iteration a CPU cycle is executed and the following operations are performed:

1. The next `NUM_SLOTS` is decoded and the program counter is incremented (only if the `IF` pipeline stage is not stalled).

2. For each instruction in the pipeline the method corresponding to the stage where the instruction is placed is executed.

3. Each instruction is advanced to the next pipeline stage. In case some stage is stalled only a subset of those instructions are advanced.

4. Memory and cache state is advanced.

A.1 Instruction Simulation

The class `instruction_t` is the root of a hierarchy of classes implementing each a processor instruction, this hierarchy is shown in figure A.2 and allows to reuse implementations common to group of instructions.

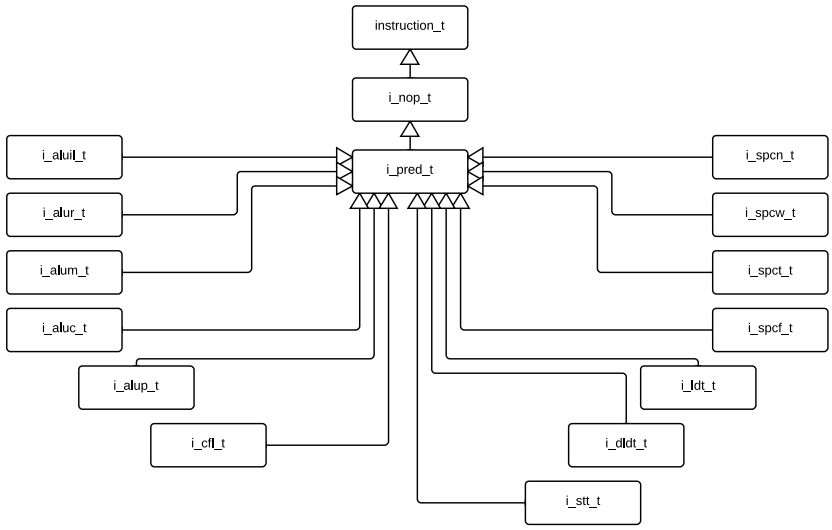


Figure A.2: Class diagram for simulator's instructions hierarchy

A.2 Memory and Cache Simulation

Memories, caches and memory mapped devices can be accessed through abstract classes which isolate from their actual implementation. The abstract class `memory_t` is extended by other three abstract classes: `data_cache_t`, `stack_cache_t` and `memory_map_t`.

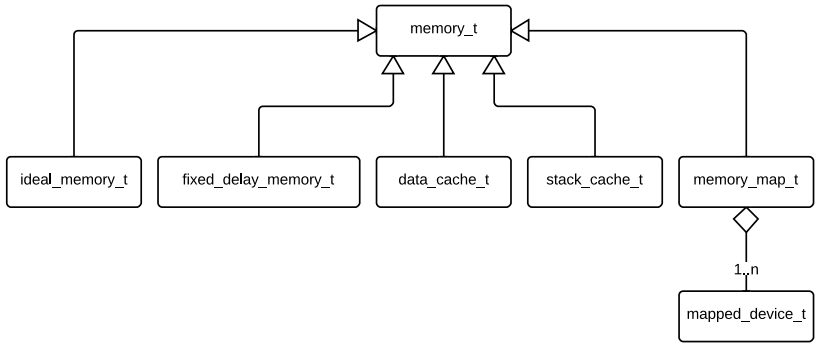


Figure A.3: Class diagram for simulator's memory hierarchy

`memory_map_t` extends `memory_t` and so it offers the same interface. `memory_map_t` wraps an object of the class `memory_t` extends its base class with a collection of `mapped_device_t` and offers methods to add object of this type to that collection. The class `mapped_device_t` represents a portion of memory which corresponds to the content of the registers of a memory mapped IO device (for example an UART). When a read/write method is called for a specific address on an object of the class `memory_map_t` all the `mapped_device_t` are inspected to look for a device having a mapped register for the address, if an object is found the read/write is forwarded otherwise the wrapped `memory_t` is accessed. In the case of Patmos simulator the `memory_map_t` object is wrapped around the local memory, so memory mapped devices can be accessed through local memory read/write instructions.

Appendix B

ELF File Structure

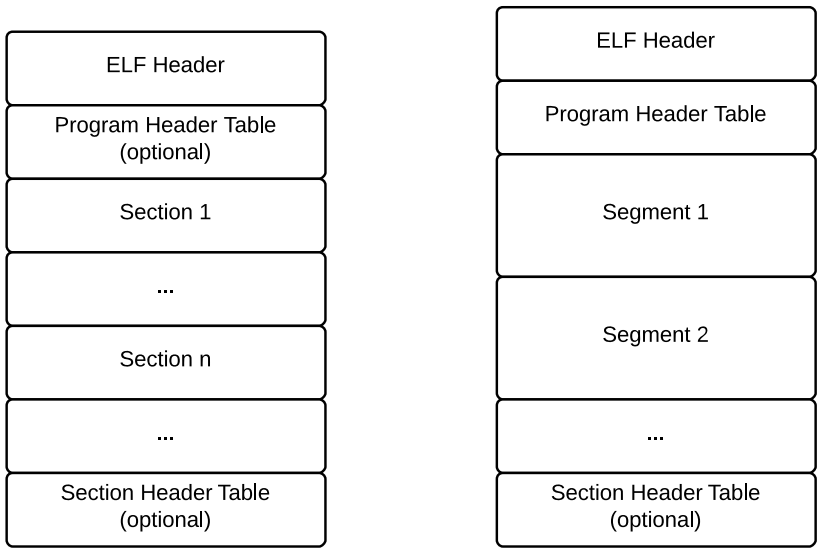
The Executable and Linking Format (`ELF`), defined in [Com95], is a standard file format used for executable files, object files and shared libraries. ELF is meant to be extensible and portable in order to make compilation and reuse of compiled code easier. ELF file format is adopted by different operating systems since it is flexible and not tied up to a particular architecture. ELF can be used to define three main types of files:

- **relocatable file**: can be used to build an executable file or a shared object file, contain code and data
- **executable file**: contain a program and can be executed
- **shared object file**: is suited for compilation with relocatable files to build other shared objects or with other shared objects to build executable files

B.1 File Structure

What emerges from the ELF file types is that an object file can both be used in the linking process and in the execution itself. The same file offers different

views on the data it contains in order to satisfy these requirements (as shown in figure B.1).



(a) Linker perspective **(b)** Execution perspective

Figure B.1: ELF file structure

B.1.1 ELF Header

The ELF header contains general information about the file that can be summarized as follows:

- `e_indent`: the first bytes are used to identify an object file
- `e_type`: the type of the file, main types are: `ET_REL` (relocatable file), `ET_EXEC` (executable file) and `ET_DYN` (shared object file)
- `e_machine`: the architecture of the file
- `e_version`: version of the object file
- `e_entry`: address of the entry point of the executable, if an entry point is specified
- `e_phoff`: offset in bytes of the program header table in the file

- `e_shoff`: offset in bytes of the section header table in the file
- `e_flags`: processor-specific flags
- `e_ehsize`: size of ELF header
- `e_phentsize`: size of an entry in the program's header table
- `e_phnum`: number of entries in the program's header table
- `e_shnum`: number of entries in the section's header table

B.1.2 Program Header

A program header table describes how to create a program from an ELF file and is needed only for executable and shared object file. A program header table is a collection of program headers each of which describes an object segment, a collection of sections.
A program header is a data structure made of the following fields:

- `p_type`: type of the segment, main values are:
  - `PT_NULL`: unused segment
  - `PT_LOAD`: loadable segment
  - `PT_DYNAMIC`: dynamic linking information
- `p_offset`: offset from the beginning of the file for the segment
- `p_vaddr`: virtual address where the segment is located in memory
- `p_addr`: physical address where the segment is located in memory
- `p_filesz`: number of bytes of the segment in the file
- `p_memsz`: number of bytes of the segment in memory
- `p_flags`: flags for the segment
- `p_align`: alignment of the segment in the file and memory, integer number (power of 2) in bytes

B.1.3 Section Header

The section header table is mandatory only for ELF files used in the linking process. Section header table contains an entry for each section in the file. Section entries contain information regarding the section like name or size.
All the information contained in an ELF file is divided in section, apart from ELF header, program header table and section header table.
An ELF sections have to meet the following requirements:

- A section must have a section header
- A section is hold by contiguous bytes in the file
- Sections do not overlap
- Sections may not cover all the data in the file (inactive data)

The fields contained in a section header are the following:

- `sh_name`: name of the section
- `sh_type`: type of the section, describes its content
- `sh_flags`: collection of 1 bit flags for the section
- `sh_addr`: if the section will be loaded in memory this field contains the address of the first byte of the section in memory
- `sh_offset`: the offset in byte of the first byte of the section from the beginning of the file
- `sh_size`: section's size in byte
- `sh_link`: section header index to link information for the section
- `sh_info`: extra information about the section
- `sh_addralign`: special information about alignment, may be needed by some sections
- `sh_entsize`: for sections that hold a table structure this field defined the entry size

B.1.3.1 Pre-defined Sections

Some sections in an header file are pre-defined containing program and control information used by the operating system.
When creating an executable more object files are put together in the linking phase, the linker resolves references between object files, updates absolute references and relocates instructions. In order to do this job some extra information is needed, this information is contained in sections like `.dynamic`.
Pre-defined sections have reserved names beginning with a `.`, they can be:

- `.bss`: holds uninitialized data and is loaded into memory and set to 0
- `.comment`: holds version control information
- `.data` and `.data1`: holds initialized data loaded into memory
- `.debug`: holds content for symbolic debugging purpose
- `.dynamic`: contains information for dynamic linking
- `.hash`: hash table of symbols
- `.line`: contains line numbering information for symbolic debugging
- `.note`: holds special information eventually put by vendors
- `.rodata` and `.rodata1`: holds read-only values used to build the memory image of the process (non-writable data)
- `.shstrtab`: contains section names data
- `.strtab`: contains string data, usually symbols names
- `.symtab`: symbol table
- `.text`: the program's instructions

Bibliography

[ABS13] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[ARM12] ARM. Application Note 245: Migrating from Power Architecture to ARM. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0245b/index.html`, August 2012.

[BA01] M.D. Bennett and N.C. Audsley. Predictable and efficient virtual addressing for safety-critical real-time systems. *2012 24th Euromicro Conference on Real-Time Systems*, 0:0183, 2001.

[Ben06] M. Ben. *Principles of concurrent and distributed programming, second edition.* Addison-Wesley, second edition, 2006.

[BMV12] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. A time-composable operating system. In *WCET*, pages 69–80, 2012.

[BMV13] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. Towards a time-composable operating system. In *Ada-Europe*, pages 143–160, 2013.

[Car04] John Carbone. Efficient memory protection for embedded systems. `http://www.rtcmagazine.com/articles/view/100120`, September 2004.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *HAND-*

BOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS. Chapman Hall/CRC, Boca, 2004.

[Com95] TIS (Tool Interface Standard) Commitee. Executable and Linking Format (ELF) Specification. `http://pdos.csail.mit.edu/6.828/2012/readings/elf.pdf`, May 1995.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[DL78] Sudarshan K Dhall and CL Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[DL11] Julien Delange and Laurent Lec. Pok, an arinc653-compliant operating system released under the bsd license. *13th Real-Time Linux Workshop*, 10 2011.

[DTU12a] DTU. D 2.1 Software Simulator of Patmos. Technical report, T-CREST: `http://www.t-crest.org/page/results`, 2012.

[DTU12b] DTU. D 5.1 ISA and Architectural Support for Generating Time-Predictable Code. Technical report, T-CREST: `http://www.t-crest.org/page/results`, 2012.

[DTU12c] DTU. D 8.2 T-CREST White Paper. Technical report, T-CREST: `http://www.t-crest.org/page/results`, 2012.

[Fre05] Freescale. Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture. `http://www.freescale.com/files/product/doc/MPCFPE32B.pdf`, 2005.

[Fre12] Freescale. Powerpc 750 microprocessor. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_750_Microprocessor`, 2012.

[Gro03] APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface, 2003.

[HEM+11] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, Andrew Nelson, Jude Ambrose, and Kees Goossens. Design and implementation of an operating system for composable processor sharing. *Microprocess. Microsyst.*, 35(2):246–260, March 2011.

[IBM98] IBM. Developing powerpc embedded application binary interface (EABI) compliant programs. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/`

852569B20050FF77852569970071B0D6/\protect\T1\
textdollarfile/eabi_app.pdf, 1998.

[Inc08] Wind River Inc. ARINC 653 - An Avionics Standard for Safe, Partitioned Systems. http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf, June 2008.

[Ins11] Texas Instruments. AM17x/AM18x ARM microprocessor peripherals overview. http://www.ti.com/lit/ug/sprufu0d/sprufu0d.pdf, September 2011.

[INT] INTEGRITY. http://www.ghs.com/products/rtos/integrity.html.

[KO02] H. Kopetz and R. Obermaisser. Temporal composability [real-time embedded systems]. *Computing Control Engineering Journal*, 13(4):156–162, 2002.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[LRL10] Isaac Liu, Jan Reineke, and Edward A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *44th Asilomar Conference on Signals, Systems, and Computers*, pages 2111–2115, November 2010.

[Lyn] LynxOS-178. http://www.lynuxworks.com/rtos/rtos-178.php.

[Mal09] R. Mall. *Real-Time Systems: Theory and Practice*. Pearson Education, 2009.

[Mok83] Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard–real–time environment. Technical Report MIT-LCS-TR-297, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1983. Ph.D. Thesis.

[Mol] Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler. In 2.5 kernel documentation (Documentation/sched-design.txt).

[Mue95] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, November 1995.

[NKG+02] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino Busa, Kees Goossens, Rafael Peset Llopis, and Paul Lippens. C-HEAP: A Heterogeneous Multiprocessor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems, 2002.

[Pik] PikeOS. `http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/`.

[SBH+] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook.

[SBWT87] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5:807–813, 1987.

[Sch04] Martin Schoeberl. A time predictable instruction cache for a java processor. In *In On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of LNCS*, pages 371–382. Springer, 2004.

[SMB05] Matthew Simpson, Bhuvan Middha, and Rajeev Barua. Segment protection for embedded systems using run-time checks. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, pages 66–77, New York, NY, USA, 2005. ACM.

[SSP+11] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[Sta08] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[TBV10] Sarah Thompson, Guillaume P. Brat, and Arnaud Venet. Software model checking of arinc-653 flight code with mcp. In *NASA Formal Methods*, pages 171–181, 2010.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.