Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Ruben S. Sevaldson

Memory protection for embedded RISC-V systems

Masteroppgave i Datateknologi
Veileder: Michael Engel
Juni 2022

NTNU
Norwegian University of
Science and Technology

Ruben S. Sevaldson

Memory protection for embedded RISC-V systems

Masteroppgave i Datateknologi
Veileder: Michael Engel
Juni 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Abstract

Operating systems for embedded devices are typically either fully-featured general-purpose operating systems, such as Linux, or much more minimal alternatives. This presents an issue for IoT (Internet of Things) devices, which require the level of security provided in fully-featured operating systems, but IoT devices are often too resource-constrained to run such operating systems. There is a need for operating systems which are both efficient and that can provide a sufficient degree of security. In this thesis, we explore the possibility of using memory protection in a microkernel on embedded devices based on the RISC-V Instruction Set Architecture (ISA) that does not have an MMU (Memory Management Unit).

The F9 Microkernel is an ARM-based microkernel that is secure, efficient, and ideal for IoT devices. F9 uses the ARM Memory Protection Unit (MPU) to implement its memory protection mechanisms. In this thesis, an experimental port of the F9 microkernel to the RISC-V ISA has been created. RISC-V has a specification for an MPU named Physical Memory Protection (PMP), which functions in much the same way as the ARM MPU, but also has an advantage. This advantage is in the form of an addressing mode on the PMP with the name Top Of Range (TOR), a feature that is exploited in the port of F9 to improve the memory protection implementation. The port of F9 to RISC-V is a good indicator that RISC-V is a suitable platform for implementing memory protection in embedded devices.

The work in this thesis also explores related topics such as memory protection in L4-style microkernels running on embedded devices without an MMU. In addition, an implementation of the LISP programming language was created, named Hoppus. Hoppus runs as a user thread on the port of F9, intending to have a dynamic garbage-collected language that serves as a basis for creating interesting integration with the memory management and memory protection mechanisms in F9. Hoppus is a platform that should facilitate further experimentation and research on microkernels and memory protection on resource-constrained devices.

Sammendrag

Operativsystemer for innvevde enheter er vanligvis enten fullt utstyrte operativsystemer, som Linux, eller minimale alternativer. Dette representerer et problem for IoT-enheter (Internet of Things-enheter), som trenger et visst nivå av sikkerhet som fullt utstyrte operativsystemer kan tilby, men som ofte har for lite ressurser til å kunne kjøre dem. Det er et behov for operativsystemer som er både effektive og som kan tilby det nødvendige nivået av sikkerhet. I denne oppgaven utforskes muligheter for å bruke minnebeskyttelse i en mikrokjerne på innvevde enheter basert på RISC-V instruksjonssettet som ikke har en minnehåndterer.

F9 er en mikrokjerne basert på ARM-instruksjonssettet, og er en sikker og effektiv mikrokjerne som er ideell for IoT enheter. F9 benytter seg av minnebeskyttelse i ARM for å implementere sine minnebeskyttelsesmekanismer. I denne oppgaven er en eksperimentell portering av F9 til RISC-V gjennomført. RISC-V inneholder en spesifikasjon for en minnebeskyttelsesenhet navngitt Physical Memory Protection (PMP), som fungerer i stor grad på en lignende måte som minnebeskyttelsesenheten i ARM, men som også har en fordel. Denne fordelen eksisterer i form av en adresseringsmodus med navnet Top Of Range (TOR), en egenskap som er utnyttet i porteringen av F9 for å forbedre implementasjonen av minnebeskyttelse. Porteringen av F9 til RISC-V er en god indikator for at RISC-V er en passende plattform for implementering av minnebeskyttelse i innvevde systemer.

Arbeidet i denne oppgaven utforsker også relaterte emner som minnebeskyttelse i L4-baserte mikrokjerner som kjører på innvevde systemer uten en minnehåndterer. I tillegg er en implementasjon av programmeringsspråket LISP laget, navngitt Hoppus. Hoppus kjører som en brukertråd på porteringen av F9, med intensjon om å ha et dynamisk språk med avfallsinnhenting som kan fungere som en basis for interessante integrasjoner med minnehåndterings- og minnebeskyttelsesmekanismene i F9. Hoppus er en plattform som skal legge til rette for videre eksperimentering og forskning på mikrokjerner og minnebeskyttelse på enheter med begrensede ressurser.

Acknowledgements

I would like to thank my supervisor, Professor Michael Engel, without his help and valuable input this thesis would not have been possible, and an additional thanks for providing the most interesting project I have ever worked on. I would also like to thank my dear friends for the many interesting conversations we had, which provided much inspiration and interesting perspectives. And i would like to give special thanks to Tale Vang Ellefsen for her unrelenting support.

Contents

Glossary

- `CLINT`: Core-Local Interruptor
- `CSR`: Control Status Register, configuration registers in RISC-V
- `GPR`: General Purpose Register
- `IPC`: Inter Process Communication
- `ISA`: Instruction Set Architecture
- `MCU`: Microcontroller unit
- `MMU`: Memory Management Unit
- `MPU`: Memory Protection Unit
- `MR`: Message Register, for IPC in L4 microkernels
- `NAPOT`: Naturally Aligned Power-Of-Two, addressing mode for PMP
- `PLIC`: Platform-Level Interrupt Controller
- `PMP`: Physical Memory Protection, memory protection unit for RISC-V
- `REPL`: Read-Eval-Print Loop
- `RTOS`: Real Time Operating System
- `SoC`: System on a chip
- `TCB`: Thread Control Block
- `TCB`: Trusted Computing Base
- `TOR`: Top Of Range, addressing mode for PMP
- `UART`: Universal Asynchronous Receiver/Transmitter
- `UTCB`: User Thread Control Block

List of Figures

Code listings

1 Introduction

The research in this thesis is focused on memory protection for RISC-V based embedded devices with constrained resources and without an MMU. Such devices do not have the capacity to run fully-featured operating systems such as Linux, but at the same time, they often require suitable protection mechanisms, which those operating systems can provide. This is especially true for IoT devices, which often fit the profile of having constrained resources and no MMU, and where security is a primary concern. RISC-V is a relatively new ISA with much ongoing research, which presents opportunities in this area. This thesis explores the RISC-V ISA and its features regarding memory protection to create a lightweight alternative to a fully-featured operating system that is also secure.

The usual way of implementing process isolation in operating systems is with virtual memory. Virtual memory is a technique that has been widely adopted and is a crucial component of most modern operating systems. However, virtual memory does not come without some overhead. The most prominent overhead of virtual memory exists in the case of potentially needing to read or write from secondary memory when reading or writing to primary memory. This overhead is usually tolerable in general-purpose operating systems, but they are not tolerable in systems such as Real-Time Operating Systems (RTOS), which have timing constraints. Virtual memory also adds overhead to the complexity of hardware design. Because of these reasons, many embedded systems are designed without an MMU, the central hardware component that enables virtual memory.

The RISC-V ISA is a free and open-source ISA that is designed to be suitable for a broad range of devices. Being an open-source ISA has several benefits, many of them a consequence of the fact that manufacturers and researchers can work with the ISA without having to pay licensing costs [24]. It also means that embedded devices can be made more cost-effective by employing RISC-V, which is an interesting opportunity for embedded device manufacturers. Such embedded devices often do not have an MMU, so physical memory protection can serve as a lightweight and robust security mechanism. There is a need to bridge the gap between these groups of systems. Embedded systems for IoT that do not have an MMU should still be able to have some degree of security to ensure they are running safely. This thesis explores the RISC-V ISA for opportunities to create an alternative that serves such a purpose. The basis for the security mechanisms is the RISC-V PMP (Physical Memory Protection), which provides the ability to protect regions of memory with read/write/execute permissions [4].

The F9 microkernel is an ARM-based microkernel in the L4-style of microkernels. It is a secure and efficient microkernel targeting embedded devices, specifically ARM-Cortex-M series microprocessors. A noteworthy feature is the usage of the ARM MPU to implement protection between threads. In this thesis, a port of F9 from ARM to RISC-V was created [28], with the primary goal of exploring memory protection in RISC-V. The port uses the RISC-V PMP to achieve the same protection functionality as the ARM version.

A popular choice for IoT devices is the ESP32 series of MCUs (Microcontroller Unit) from Espressif Systems. These are low-power devices based on the Tensilica Xtensa ISA. The ESP32 series of MCUs became popular for having good connectivity features while at the same time being relatively inexpensive. Recently Espressif Systems released a RISC-V version of these MCUs named ESP32-C3 [10], which conveniently includes a fully-featured RISC-V PMP implementation. Since the ESP32-C3 is both ideal for IoT and has memory protection, it was chosen as the target platform for the port of F9 to RISC-V.

In addition to the port, an implementation of the LISP programming language was created, nicknamed Hoppus. LISP is a language that works well for embedded systems and often has good support for remote interactivity, which could be useful on an IoT-focused device such as the ESP32-C3. LISP is also an extendable programming language, allowing for large amounts of functionality to be implemented in the language itself. One particularly fascinating story of LISP that inspired its usage in this thesis is the story of when NASA was able to fix a bug 60 million miles away using a LISP Read-Eval-Print Loop (REPL) [12]. Another interesting story is the use of LISP in the famous Roomba vacuum robots [27].

Figure 1 shows a simplified overview of the resulting system.

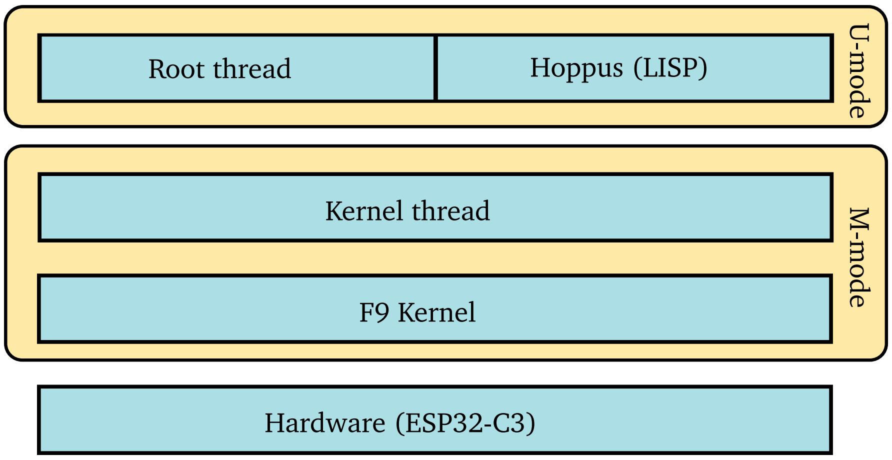


Figure 1: System overview

Some regions in this master thesis are adapted from previous work in the form of a specialization project [30]. In those cases, the start of such a regions is marked with:

(This is the beginning of a section adapted from the specialization project)

and ends with:

(End of section adapted from the specialization project)

- Source code for the ARM version of F9 can be found here [14]
- Source code for the RISC-V port of F9 can be found here [28]
- Source code for Hoppus and the garbage collector can be found here [29]

2 Background

2.1 Microkernels

In broad terms, a microkernel is the minimal amount of software required to build an operating system. In a microkernel system, the kernel is kept small, while all other required functionality is implemented in user space. user space code implements kernel functionality as servers which provide their functionality via Inter-Process Communication (IPC) messages. An advantage of this approach is that the development of the kernel becomes modular in regard to these servers. However, the principal advantage is that errors in user space code are isolated to the faulting server rather than the error occuring in kernel space.

Microkernels also have other advantages regarding the size. Larger and more complex systems usually mean a higher probability of bugs and issues in that system, and modern operating systems are incredibly complex system containing millions of lines of code, a lot of which runs in privileged mode. The TCB (Trusted Computing Base) [1] of an operating system is the collection of privileged components in the operating system. A large TCB in an operating system usually means a larger attack surface and a higher probability of bugs in the privileged part of an operating system. Therefore, in the interest of security and robustness, it is advantageous to keep the TCB small. The TCB of a microkernel is minimal, which means that the attack surface of a microkernel is reduced.

The small size of a microkernel also facilitates formal verification, which is the process of guaranteeing that the kernel behaves in a certain way by mathematical proofs. The SEL4 Microkernel is allegedly the first general-purpose operating system to have undergone complete formal verification [13]. The microkernel design is what makes this possible.

[1]In this thesis, TCB usually refers to Thread Contol Block, the exception is in this paragraph where it refers to Trusted Computing Base

Examples of Microkernels

- `Minix`
- `Zirchon`
- `FreeRTOS`

2.2 L4

(This is the beginning of a section adapted from the specialization project)

The Mach microkernel [2], a first-generation microkernel, was one of the earliest microkernels made. It was considered the flagship of industrial microkernels for a long time. While Mach was promising and had many aspects that generated interest among developers, it struggled with performance issues, as did many first-generation microkernels. The culprit of the poor performance was said to be frequent context switches, which became known as a problem intrinsic to microkernels. Common sense would indicate that this holds since microkernels perform more context switches than monolithic kernels. [20]

Jochen Liedtke was a German computer scientist who challenged the notion that microkernels were inherently slow and showed that the experienced performance issues of existing microkernels were due to poor implementations [18]. He further argued that existing microkernels had moved more functionality into the kernels to mitigate the performance issue and that this was a mistake. According to Liedtke [19]:

> ... a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

Liedtke also created his own microkernels, proving that his microkernel performance assessments were correct. Liedtke eventually made the L4 microkernel, a second-generation microkernel. By prioritizing performance and ensuring fast IPC and smaller cache footprint, L4 outperformed first-generation microkernels [20]. L4 inspired dozens of other projects, and eventually, L4 became something more of a specification than a microkernel implementation. There is also a third generation of microkernels, which improved portability from the second generation, which was inherently not portable. Today there are multiple implementations of L4 with different design goals.

- `SeL4` : which focuses on security an formal verification [16]

- `Fiasco` : which features a runtime system for development convenience
- `PikeOS` : for safety-critical real-time embedded systems [2] [15].

An overview of different kinds of L4 microkernels can be seen in figure 2

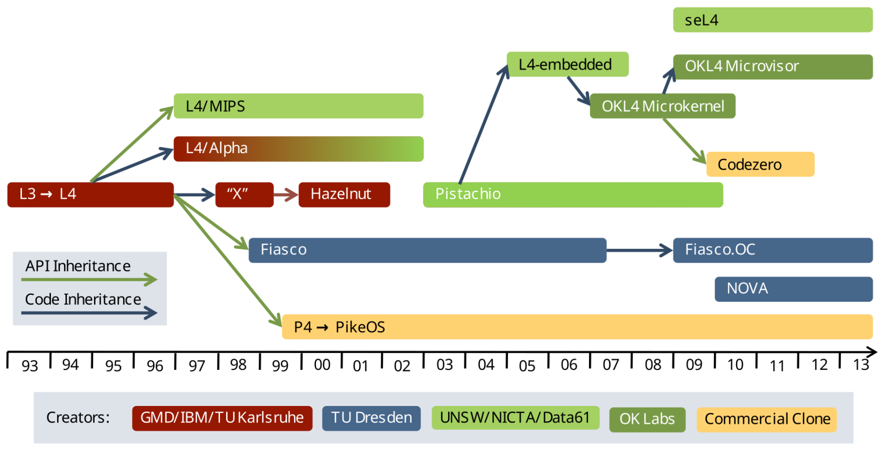


Figure 2: L4 microkernel family tree (Source: [13])

L4 provides three fundamental abstractions, threads, address spaces and IPC. A thread in L4 is the execution abstraction which includes a context and is related to an address space.

2.2.1 Address spaces

One of the criticisms levied against first-generation microkernels targeted the concept of the external pager. Critics argued that the external pager was not flexible enough because it hardwired a policy inside the kernel. The external pager could control main memory rudimentarily, but not enough for some applications [20]. A different approach was needed.

L4 provides the concept of address spaces. Address spaces consist of flex pages which are mappings of virtual addresses to physical addresses. When a context switch occurs to a new thread, the flex pages on the new thread are configured on the MPU, which sets up translations from virtual addresses to the correct physical addresses for that thread. A flex page can end up being configured as a single page by the MPU, but several pages could also be used to configure a flex page.

The purpose of this design is to allow user threads to employ recursive construction of address spaces to implement whatever memory management scheme that is required [19]. L4 provides three operations for managing address spaces:

[2] started out as an L4 microkernel, but has diverged

1. `Grant` - Removes part of an address space and adds it to another address space.
2. `Map` - Maps parts of an address space to another address space.
3. `Flush/Unmap` - Removes pages from an address space.

At system start, all address spaces are empty except for a "magic" address space σ_0 that is controlled by the kernel. All other address spaces are constructed from σ_0 by the three operations outlined above. Isolation is ensured by only allowing a thread to map, grant or unmap regions it has already been mapped or granted. [19]

(End of section adapted from the specialization project)

2.2.2 IPC

IPC is an essential part of L4 and is the driving force behind its high performance. In L4, IPC is used in different ways to achieve much functionality. For example, IPC can synchronize between threads, send messages, delegate interrupts, start newly created threads, and send an entire send-receive-reply cycle with one system call for the sender and receiver each. Using IPC as a multi-tool of communication and making it fast is one of the critical factors to the success of L4.

Many of the techniques employed to improve IPC performance are detailed by a paper by Jochen Liedtke [18]. A multitude of techniques is presented that synergize well together. A significant fraction of the techniques are related to improving virtual memory performance when doing IPC transfer, such as avoiding page faults and flushing the TLB. [3]

Some of the techniques include:

- (5.2.1) reducing the number of context switches per IPC communication.
- (5.2.5) using a shared large virtual array for Thread Control Blocks (TCB) to improve lookup times and reduce Translation Lookaside Buffer (TLB) misses
- (5.3.1) Ability to calculate TCB from thread number and embed the thread number in the lower 32-bits of the thread id
- (5.3.5) Direct process switching to the thread receiving an IPC message [4]

[3]While the RISC-V port of the F9 microkernel described in this thesis does not employ virtual memory, some techniques are still relevant.

[4]The F9 scheduler does this

- (5.3.6) Short messages via registers [5]

Another technique to improve performance is the concept of a User Thread Control Block. The idea is to split the TCB into a KTCB (Kernel Thread Control Block) and UTCB (User Thread Control Block). The UTCB contains IPC message data and is unprotected from user space. Page faults and TLB misses can be reduced by keeping the UTCB mapped in virtual memory. [21]

2.3 Memory Management

Most modern systems include an MMU and run operating systems that implement paging and virtual memory. While the work in this thesis concerns devices without an MMU, it is still useful with a short introduction to these concepts for context.

2.3.1 Paging and virtual memory

Virtual memory was invented in 1961 with design of the Atlas computer, which was the very first computer supporting virtual memory [6] [6]. Only slightly behind the Atlas computer, the first commercial computer with virtual memory was the B5000 machine from Burroughs Large Systems Group. Virtual memory is taken for granted in most computer systems today, but the idea has been the subject of much debate. Before virtual memory was in widespread use, programs were mostly accessing physical memory directly, which has some drawbacks:

- The OS must be careful to share all available memory between processes, ensuring that one process does not modify the address space of another process (unless it gets permission to do so).
- The OS either needs to have mechanisms for relocating processes in physical memory (complex), or else it needs to keep processes at the same memory location (inflexible).
- Perhaps worst, each process gets only a slice (or some slices) of available physical memory that it can use, which is restrictive.

Paging and virtual memory is the solution to these problems. With paging, memory is split into equal-sized page frames, e.g. 4kb in size. Each process gets memory in the form of pages which constitute its address space. With virtual

[5]F9 does this, and additionally, RISC-V has an opportunity to exploit this further because of its many general-purpose registers

[6]also one of the first supercomputers in existence

memory, a level of indirection is introduced where the addresses a process is accessing is not the actual physical address (although to the running process, it looks exactly like a physical address) but rather a virtual address that maps to a physical address.

Bringing paging and virtual memory together allows the OS to map virtual addresses to physical addresses for processes. Then, if the OS runs out of space in physical memory, it can back up whatever pages it needs in secondary memory. This way, processes can work with a much larger virtual address space and let the OS do the heavy lifting of swapping out pages between physical and secondary memory as needed.

There are many details and complexities involved with paging and virtual memory. For example, a page table is needed to store mappings between addresses, a `translation lookaside buffer` is needed to make page table lookups fast, and good algorithms are needed to find out what page table entries to throw out when physical memory is full. In addition, virtual memory is not desirable for use in systems with heavy timing constraints, such as real-time systems. These systems cannot afford the overhead of potentially reading from secondary memory when doing operations on primary memory, so virtual memory is usually not an option in such cases. A detailed discussion of these implementation details are omitted in this thesis, the point is that there is a certain overhead and complexity involved, and there might be advantages to a system without it.

2.3.2 Segmentation

While paging splits memory into equally-sized parts, segmentation partitions memory into segments that do not have to be the same size. An advantage of this is that contiguous regions of memory containing data for specific use can be placed in a single segment. For example, memory regions containing only code can be put in a code segment, while regions that have only data that should be read-only can be placed in a read-only data segment. However, while it is true that paging can cause internal fragmentation, segmentation tends to cause a more significant amount of external fragmentation. In modern operating systems, paging and virtual memory are most often used, but segmentation can have a supplementary role.

2.4 RISC-V

RISC-V is a specification for a RISC based ISA that was created at UC berkeley [3] [4] [23]. What is unique about RISC-V is that it is an open-source specification, which means that no licensing fees are required to implement a RISC-V based design. RISC-V is also a relatively new ISA, where the work on RISC-V started in 2010.

2.4.1 Privilege modes

RISC-V features three privilege modes that a CPU can be in; user-mode, supervisor-mode and machine-mode. Machine-mode is the most privileged mode and is the mode that is active when the CPU boots.

- `machine-mode`: In machine-mode, virtual memory is ignored, and the RISC-V PMP is ignored [7], and all privileged instructions can be executed. Usually, this mode is only active during the boot process, preferring supervisor-mode instead. However, some devices do not implement supervisor-mode, in which case machine-mode must be used.
- `supervisor-mode`: supervisor-mode is an optional mode that is less privileged than machine-mode, but more privileged than user-mode. Machine-mode instructions are unavailable in supervisor-mode, and PMP restrictions apply. Notably, supervisor-mode provides the basis for implementing page-based virtual memory.
- `user-mode`: the least privileged mode is user-mode. User-mode cannot execute machine-mode or supervisor mode instructions and is subject to PMP restriction. User-mode exists in order to execute untrusted code in a protected matter.

2.4.2 Trap handling

The RISC-V specification provides definitions of interrupts [3], exceptions and traps that this thesis will follow:

> We use the term exception to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart. We use the term interrupt to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term trap to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.

Exception/interrupt handling in RISC-V functions much like other ISAs; an exception or interrupt occurs where the CPU performs a series of steps, eventually ending with jumping to a configured address for a trap handler. By default,

[7]Unless the lock bit is set for the configuration register of the given region

all exceptions are handled by machine-mode, meaning that the trap handler will start execution with the CPU in machine-mode. However, exceptions can be configured to be delegated, or the trap handler can cause another exception into a less privileged mode.

In order to set up a trap vector, the `mtvec` Control Status Register (CSR) is used. `mtvec` can function in either direct or vectored mode, where direct mode jumps to the same address on all traps, and vectored mode can be used to define a trap vector. The `mcause` registers contain information on what kind of interrupt occurred, which is especially useful when `mtvec` is configured in direct mode. The possible types of traps can be seen in figure 3.

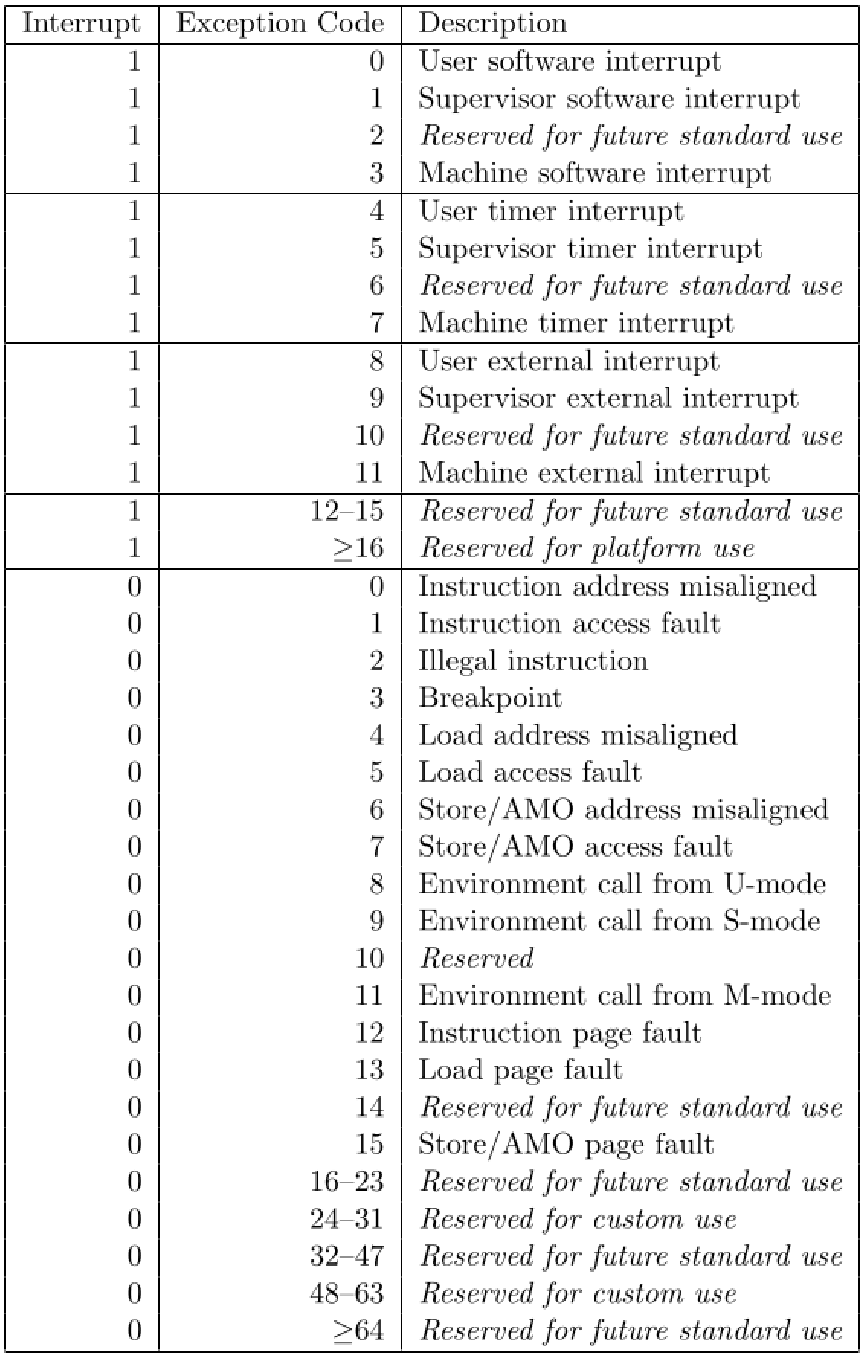
2.4.3 Memory protection

The RISC-V PMP is a scheme in the RISC-V standard that describes a MPU [4]. An MPU is a unit that provides basic memory protection while being smaller than a fully-featured MMU. It does not provide features such as virtual memory or paging. Instead, it lets the programmer define a set of regions with various privileges. In the case of the PMP, up to 16 regions may be defined, each with read/write/execute privileges.

The RISC-V PMP is managed by several CSR. `pmpaddri` registers are used to specify memory regions, while `pmpcfgi` registers are used to configure those regions. Each `pmpaddri` register can be configured to be in Naturally Aligned Power-Of-Two region (NAPOT) mode or Top Of Range region (TOR) mode (See figure 5). In NAPOT mode, each `pmpaddri` register describes a memory region on its own, where the lower order bits are used to encode the region's size. NAPOT mode has the advantage of only using one `pmpaddri` register for a region, but it requires that regions that are not naturally aligned are split up into smaller regions. TOR mode uses two `pmpaddri` registers to define a region by a range. When TOR mode is selected for a `pmpadrri` register, that register forms the top of the range of the region, while the preceding register `pmpaddr(i - 1)` forms the bottom of the range. This has the obvious disadvantage of using two registers for a memory region; however, it does not suffer any alignment limitations.

In addition to setting the addressing mode, the `pmpcfgi` registers are also used to set specific permission bits for each memory region. These permissions are read, write or execute. If a region has any of these bits set in the corresponding `pmpcfgi` register, then that type of operation is permitted. The exception to this rule is if the processor is running in machine-mode, in which case all memory is accessible.

There are 16 `pmpaddr` registers and 4 `pmpcfg` registers. The configuration of memory regions is densely packed in the configuration registers, which means that `pmpcfg0` contains the configuration for `pmpaddr0`, `pmpaddr1`, `pmpaddr2` and



Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Figure 3: Possible values for the RISC-V `mcause` register, and their meaning

`pmpaddr3`, as can be seen in figure 4.

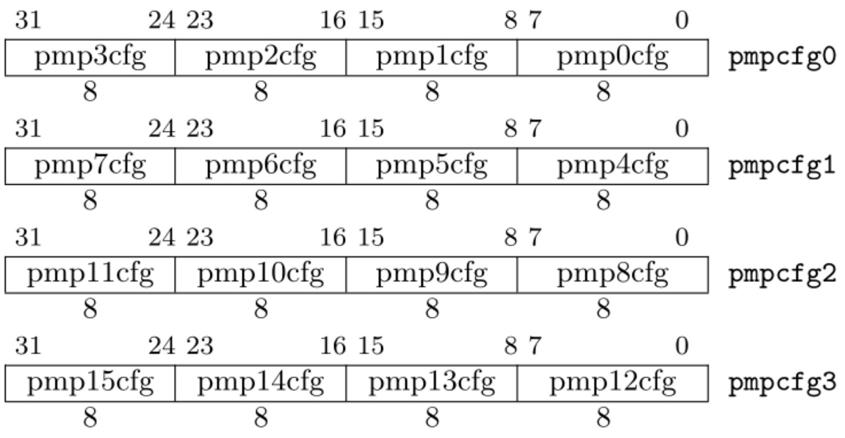

31 24 23 16 15 8 7 0
pmp3cfg pmp2cfg pmp1cfg pmp0cfg
8 8 8 8
pmpcfg0
pmp7cfg pmp6cfg pmp5cfg pmp4cfg
pmpcfg1
pmp11cfg pmp10cfg pmp9cfg pmp8cfg
pmpcfg2
pmp15cfg pmp14cfg pmp13cfg pmp12cfg
pmpcfg3

Figure 4: RV32 PMP configuration CSR layout. (Source: [4])

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

Figure 5: The different modes for a pmp region, depending on the A bit in the corresponding pmpcfg register. (Source [4])

Since there are 16 `pmpaddri` registers, there are 16 configurable memory regions when all regions use NAPOT mode addressing. When using TOR-mode addressing, two `pmpaddri` registers are required for each region resulting in a total of 8 memory regions.

2.5 Difference in memory protection on ARM and RISC-V

While RISC-V has the PMP spec for memory protection, ARM has an equivalent in the form of the ARM MPU. The ARM MPU and the RISC-V PMP function in much the same way; regions of memory are defined with certain permissions, then any memory access that does not conform to those regions triggers an appropriate interrupt that can then be handled.

Regions in the ARM MPU have the limitation that the regions are at least 32-byte in size and that they are naturally aligned, meaning that the base address must be a multiple of the size. The number of protected regions is implementation-defined [5]. ARM Cortex-M4 and -M3 cores have eight regions (the cores supported by F9 ARM).

Comparing ARM to RISC-V, an obvious advantage of the RISC-V design is the option of using TOR-mode addresses. With naturally aligned addresses, the software might have to split up a region into several naturally aligned regions in order to cover a memory region that is not naturally aligned. This is unnecessary with TOR-mode since any start and end address can set.

3 Previous work

3.1 F9

Most of the work described in this master thesis was concerning porting the F9 microkernel [14] from ARM to RISC-V and the ESP32-C3. The ARM version of F9 is an experimental microkernel inspired by L4, and tries to follow the L4 eXperimental reference manual [31] whenever possible. From the F9 `readme` [14]:

> The motivation of F9 microkernel is to deploy modern kernel techniques to support running real-time and time-sharing applications (for example, wireless communications) for ARM Cortex-M series microprocessors with efficiency (performance + power consumption) and security (memory protection + isolated execution) in mind.

F9 supports a selection of development boards based on ARM cortex cores.

- STM32F4DISCOVERY - Cortex M4F
- 32F429ISISCOVERY - Cortex M4F
- STM32-P103 - Cortex M3

3.1.1 Memory protection in F9

In L4, address spaces are implemented with flex pages representing mappings between virtual and physical addresses. This concept assumes the existence of an MMU that supports virtual memory. F9 is designed to run on systems without an MMU, so the concept of a flex page is adjusted to mean a protected region of physical memory rather than a mapping of addresses.

Each thread in F9 has an address space, which essentially are lists of flex pages. Since flex pages represent MPU regions, the address space of a thread specifies what memory that thread can access. An address space can have a number of flex pages on it, but the MPU supports a limited number of regions at a given time. When an access fault occurs, F9 will check if the address is within one of the flex pages that are not currently mapped for the current thread. If it finds such a flex page, the kernel will swap out an existing flex page on the MPU with the newly found flex page. (the algorithm explained in section 4.9.5 is from the port of F9, but it is mostly untouched from the ARM version of F9).

3.2 Specialization project

This thesis builds on work done in a specialization project completed in the autumn semester of 2021 [30]. In that project, a port of F9 was created to RISC-V on the QEMU emulator [1]. This work resulted in a functional port to RISC-V, but several features was missing:

- the memory protection subsystem was completely disabled, meaning all threads could access all addresses
- delegation of interrupts to user threads was not implemented
- IPC was in a semi-working order, where only some small subset of IPC calls were functional.

(more details in section 5 in [30])

4 F9 RISC-V

4.1 Overview

F9 RISC-V is an experimental RISC-V port of the F9 kernel, an L4-style microkernel written initially for ARM. The ARM version of F9 tries to follow the L4 eXperimental reference manual [31] whenever possible. The RISC-V port tries to keep this spirit but deviates from the specification in some places.

4.2 ESP32-C3

4.2.1 Overview

The target platform for the port is the ESP32-C3, an inexpensive and low-power RISC-V based microcontroller from Espressif Systems with Wi-Fi (IEEE 802.11 b/g/n) and Bluetooth 5 LE capabilities [8]. A block diagram can be viewed in figure 6.

While most MCUs from Espressif System use the Xtensa ISA, the ESP32-C3 differentiates itself by using RISC-V, which results in a more cost-effective MCU since RISC-V is a free and open-source specification. The ESP32-C3 is especially well suited for running secure IoT applications because of its connectivity and security features and low cost.

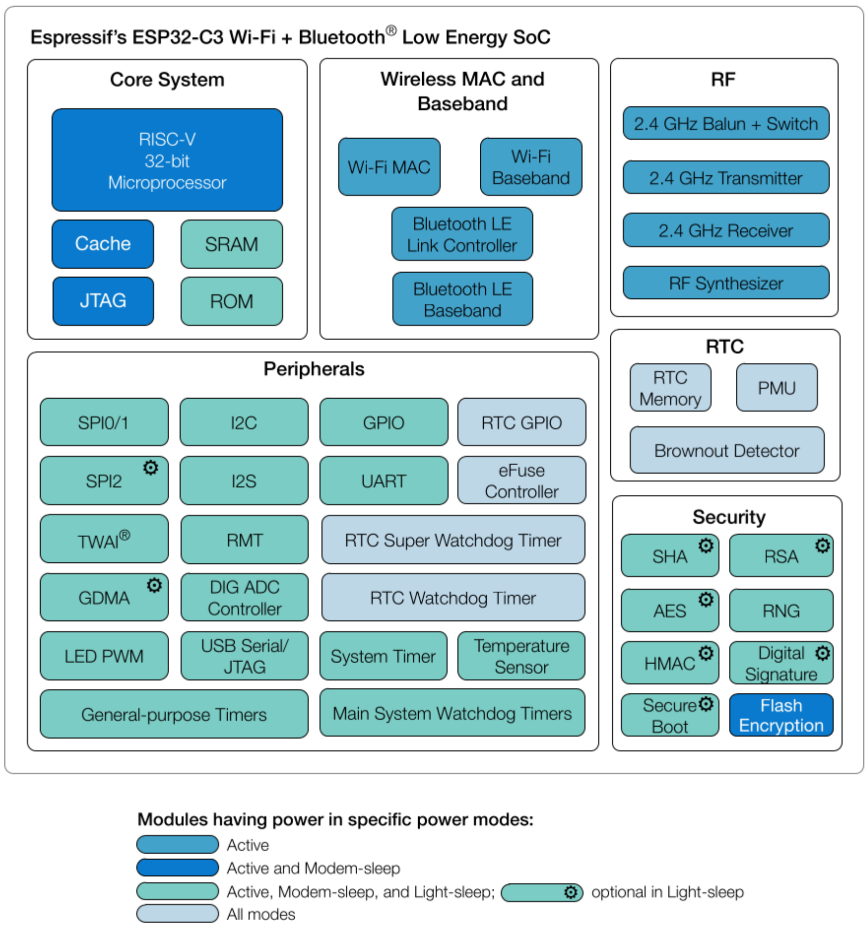


Figure 6: Block diagram of ESP32-C3 [8]

Espressif Systems also provides a comprehensive development framework for their System on a Chip (SoC), named ESP-IDF [7]. This framework can set up development projects, flash and monitor Espressif boards, and uses FreeRTOS to run applications. The F9 port described in this thesis, however, does not use this framework; the microkernel is loaded bare-metal via gdb instead.

4.2.2 RISC-V core

The ESP32-C3 features a 32-bit single-core RISC-V based processor. It has a four-stage pipeline and can operate at speeds up to 160MHz. The specific type of RISC-V architecture used is RV32IMC, meaning it implements a 32-bit version of RISC-V with the **I**nteger, **M**ultiplication/division and **C**ompressed instructions RISC-V extensions.

While the ESP32-C3 has a RISC-V core, it lacks some features from the RISC-V specification. Contrasting the `mstatus` register in the ESP32-C3 (see figure 7) with `mstatus` in the RISC-V privileged specification (see figure 8) gives an overview of the situation. The ESP32-C3 supports two privilege modes: user-mode and machine-mode, omitting supervisor-mode. It also omits the Core Local Interruptor Specification (CLINT) [25] and Platform Level Interrupt Control Specification (PLIC) [26], implementing its custom timer and interrupt controller instead.

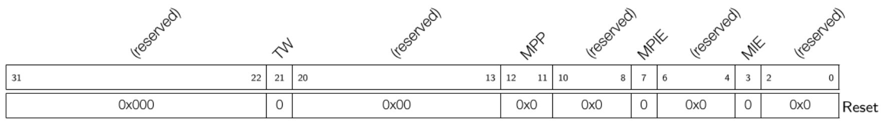


Figure 7: `mstatus` CSR in ESP32-C3. `MIE`: Global machine-mode interrupt enable. `MPIE`: Previous MIE. `MPP`: Machine previous privilege mode, 0 for user-mode, 3 for machine-mode. `TW`: Timeout wait; if this bit is set, executing a `wfi` (wait-for-interrupt) instruction from user-mode will cause an illegal instruction exception. (Source: [9])

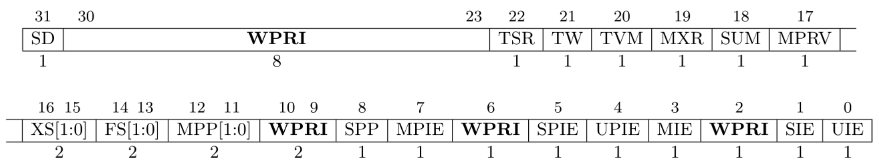


Figure 8: `mstatus` CSR in RV32. For an explanation of all the bits in the `mstatus` register for RV32, see the RISC-V privileged specification [4]. (Source: [4])

The ESP32-C provides control over machine-mode privilege levels and interrupts, as well as the ability to set `wfi` instructions as illegal in user-mode. However, it is missing support of any interrupt delegation [8] between m-mode and u-mode. The ESP32-C3 is also missing some CSR registers. For example, the `mtime` and `mtimecmp` CSRs are missing since the ESP32-C3 uses its own custom timer (See section 4.2.4). In addition, the CSRs that control machine-mode interrupt delegation in RISC-V, `mideleg` and `medelege`, are also not present.

Another deviation from the RISC-V specification is in regards to trap vector configuration. In RISC-V, the `mtvec` CSR is used to configure trap handlers. `mtvec` supports two different modes according to the RISC-V specification, direct mode and vectored mode. In direct mode, all traps will directly jump to the address specified in `mtvec`, where the trap handler has to use the `mcause` CSR to determine what kind of trap occurred. In vectored mode, the address in `mtvec` contains a vector table. The ESP32-C3 only implements vectored mode.

4.2.3 Interrupts

When code running on the ESP32-C3 is starting up, a trap vector needs to be set up via the `mtvec` register, where the ESP32-C3 only supports vectored interrupt mode. The CPU performs the following steps when a trap occurs [9]:

- saves the address of the current un-executed instruction in the `mepc` CSR for resuming execution later.
- updates the value of the `mcause` CSR with the ID of the interrupt being serviced
- copies the state of `MIE` bit in `mstatus` into `MPIE`, and subsequently clears `MIE`, thereby disabling interrupts globally.
- enters the trap by jumping to a word-aligned offset of the address stored in the `mtvec` CSR.

For peripheral interrupts, although RISC-V has a specification for an interrupt controller through the PLIC [26], the ESP32-C3 does not implement it. Instead, the ESP32-C3 features its own interrupt controller [9]. In the ESP32-C3, interrupts are categorized into peripheral interrupts and CPU interrupts. An interrupt matrix is used to map between peripheral interrupts and CPU interrupts. Peripheral interrupts are lower-level interrupts generated from hardware and are the input to the interrupt matrix, while CPU interrupts are 31 interrupts reflected in the RISC-V `mcause` register. In addition to not following the PLIC specification, the ESP32-C3 also breaks from the RISC-V specification in the way it uses the `mcause` register, as described in the technical reference manual for the ESP32-C3. [9]:

[8]Meaning the RISC-V concept of interrupt delegation, not to be confused with interrupt delegation in F9 as explained in section 4.10

Note : The interrupt controller is using up IDs in range 1-31 for all external interrupt sources. This is different from the RISC-V standard which has reserved IDs in range 0-15 for core internal interrupt sources

This divergence from the RISC-V spec regarding interrupts is the main reason for the port not working on QEMU. If it had used RISC-V PLIC, a cross-platform version would have been realistic.

In order to set up and configure interrupts in the ESP32-C3, it is required to modify both the specific peripheral interrupt in question and the interrupt matrix. The basic process is as follows:

1. Perform any required custom setup for the specific peripheral interrupt
2. Configure the peripheral interrupt as either a level or edge-triggered interrupt
3. Map the peripheral interrupt to a CPU interrupt
4. Enable the peripheral interrupt and the CPU interrupt

The port has some convenience functions where the `intr_setup` function can setup a peripheral interrupt given a peripheral interrupt and a CPU interrupt to map to, which can be viewed in listings 1 and 2.

4.2.4 System timer

While the RISC-V CLINT specification [25] provides a specification for a machine-level timer, the ESP32-C3 uses a custom system timer instead [9]. The system timer works by using a counter that increments or decrements its value at a fixed average frequency of 16MHz.

Conveniently, the timer supports a periodic alarm mode, where the counter will trigger an alarm at each multiple of a configured threshold. This is ideal for generating timer interrupts in an operating system and is what the F9 port does. In addition to configuring the timer itself, the timer interrupt must be configured on the interrupt matrix, as explained in section 4.2.3. In this case, the timer interrupt is the peripheral interrupt which needs to be mapped to a CPU interrupt. The port maps the peripheral timer interrupt to CPU interrupt 7 to have some similarity with the RISC-V specification since interrupt number 7 in the RISC-V specification is a machine timer interrupt [4].

```
void intr_set_priority(int cpu_intr_n, int pri)
{
    volatile uint32_t *interrupt_core0_cpu_int_pri = REG(INTERRUPT_MATRIX_BASE +
↪   INTERRUPT_CORE0_CPU_INT_PRI_n_REG + 0x4 * (cpu_intr_n - 1));
    *interrupt_core0_cpu_int_pri = pri;
}

void intr_set_type(int cpu_intr_n, int type)
{
    /* Set the type to edge triggered, so we can clear it with the interrupt
↪   matrix */
    volatile uint32_t *interrupt_core0_cpu_int_type = REG(INTERRUPT_MATRIX_BASE +
↪   INTERRUPT_CORE0_CPU_INT_TYPE_REG);
    if (type) {
        *interrupt_core0_cpu_int_type |= (1 << cpu_intr_n); // set edge triggered
    } else {
        *interrupt_core0_cpu_int_type &= ~(1 << cpu_intr_n); // set level
↪   triggered
    }
}

void intr_map(uint32_t *source_intr, int cpu_intr_n)
{
    volatile uint32_t *interrupt_core0_intr_map_reg = REG(INTERRUPT_MATRIX_BASE +
↪   (uint32_t) source_intr);
    *interrupt_core0_intr_map_reg = cpu_intr_n;
}
```

Listing 1: Peripheral interrupt setup convenience functions part 1

```c
void intr_en(int cpu_intr_n)
{
    volatile uint32_t *interrupt_core0_cpi_in_enable = REG(INTERRUPT_MATRIX_BASE
↪   + INTERRUPT_CORE0_CPU_INT_ENABLE_REG);
    *interrupt_core0_cpi_in_enable |= (1 << cpu_intr_n);
}

void intr_clear(int cpu_intr_n)
{
    /* Clear uart interrupt in interrupt matrix (not sure if this is necessary,
↪   but lets do it to be sure) */
    volatile uint32_t *interrupt_core0_cpu_int_clear = REG(INTERRUPT_MATRIX_BASE
↪   + INTERRUPT_CORE0_CPU_INT_CLEAR_REG);
    *interrupt_core0_cpu_int_clear |= (1 << cpu_intr_n);
    *interrupt_core0_cpu_int_clear &= ~(1 << cpu_intr_n);
}

void intr_setup(unsigned int cpu_intr_n, unsigned int priority)
{
    intr_map((uint32_t *)interrupt_core0_uart_target0_int_map_reg, cpu_intr_n);
    intr_set_priority(cpu_intr_n, priority);
    intr_set_type(cpu_intr_n, 1);
    intr_en(cpu_intr_n);
    intr_clear(cpu_intr_n);
}
```

Listing 2: Peripheral interrupt setup convenience functions part 2

The system timer interrupt is the only interrupt that is managed by the kernel itself, and using CPU interrupt 7 for this internal timer interrupt means that interrupt 7 is not available for interrupt delegation (see section 4.10)

4.2.5 Memory protection in the ESP32-C3

While the ESP32-C3 lacks several features from the RISC-V specification, it does have a fully-featured RISC-V PMP unit for memory protection. The PMP on the ESP32-C3 is configured as expected in a RISC-V device using the `pmpaddri` and `pmpcfg` CSRs. Being a fully-featured PMP, it also supports TOR addressing mode, which is an important feature for F9 RISC-V.

4.3 Kernel tables

F9 uses an internal table data structure implemented with bitmaps to provide a form of simple and fast memory management inside the kernel. Kernel tables are mostly used for tables of essential data such as TCBs and kernel timer events. The ARM version of F9 provides the option of using bit-banding to optimize kernel table operations. Bit-banding is a feature in which words of memory are mapped to bits, where a write to one of the words will set the corresponding bit. In the port, the kernel table is used without such optimizations.

Kernel tables have to be declared before they are initialized, which is done with the `DECLARE_KTABLE` macro (See listing 3. When F9 starts, all required kernel tables are initialized with the `ktable_init` function, which does some basic setup. When an object should be allocated on the kernel table, the `ktable_alloc` function is used (See listing 4). Accessing elements is usually done outside of the tables; for example, the kernel timer events are linked in a queue and accessed via that queue, but they are still allocated via the kernel table system.

```
#define DECLARE_KTABLE(type, name, num_)            \
    DECLARE_BITMAP(kt_ ## name ## _bitmap, num_);       \
    static __KTABLE type kt_ ## name ## _data [num_];    \
    ktable_t name = {                   \
            .tname = #name,                 \
            .bitmap = kt_ ## name ## _bitmap,    \
            .data = (uint32_t *) kt_ ## name ## _data,  \
            .num = num_, .size = sizeof(type)    \
    }
```

Listing 3: Macro for declaring ktables

```
void *ktable_alloc(ktable_t *kt)
{
    bitmap_cursor_t     cursor;
    /* Search for free element */
    for_each_in_bitmap(cursor, kt->bitmap, kt->num, 0) {
        if (bitmap_test_and_set_bit(cursor)) {
            int i = bitmap_cursor_id(cursor);
            return (void *) kt->data + (i * kt->size);
        }
    }
    return NULL;
}
```

Listing 4: Function for allocating objects on kernel tables

4.4 Threads

(This is the beginning of a section adapted from the specialization project)

A thread in F9 is represented by a TCB (see listing 5), which includes a User Thread Control Block, a context and an address space. User threads are managed as a tree of threads. Each TCB has a pointer to its parent, one of its siblings, and one of its children. To get all the children of a thread, one has to follow the child pointer, then follow the sibling pointers until a null pointer is reached. In F9, the UTCB is always present in the address space of the thread that has the UTCB, meaning that it is accessible to it concerning memory protection.

The context on the TCB contains a field for the `mepc` register and a field for the stack pointer. The `mepc` register is essential since it saves the value of the program counter prior to an interrupt happening, which is what is required to store on a context switch. The general-purpose registers are all stored on the stack. The definition of the context struct can be seen in listing 6.

(End of section adapted from the specialization project)

4.5 System threads

4.5.1 Kernel and kernel thread

The kernel is split into two parts, the kernel trap handler and the kernel thread. The trap handler is responsible for handling traps, as well as scheduling and context switching. Splitting the kernel into a trap handler and a kernel thread allows the trap handler to execute the most important tasks as fast as possible, outsourc-

```c
struct tcb {
    l4_thread_t t_globalid;
    l4_thread_t t_localid;

    char* name;

    thread_state_t state;

    memptr_t stack_base;
    size_t stack_size;

    context_t ctx;

    as_t *as;
    struct utcb *utcb;

    l4_thread_t ipc_from;

    struct tcb *t_sibling;
    struct tcb *t_parent;
    struct tcb *t_child;

    uint32_t timeout_event;
};
```

Listing 5: TCB struct definition

```c
typedef struct {
        uint32_t mepc;
        uint32_t sp;
} context_t;
```

Listing 6: Context struct definition

ing the rest to the kernel thread. This structure is the same as in the ARM version of F9; the difference is that the trap handler code is larger and takes on somewhat more responsibility in the port.

The kernel thread is a privileged thread running in machine-mode, which, among other things, means that it ignores all memory protection and can access all memory. It runs in machine-mode primarily because it needs to access memory in kernel space but also because of performance reasons by avoiding access fault exceptions. (no access faults will occur). The responsibilities of the kernel thread are explained in more detail in section 4.7.

4.5.2 Root thread

The root thread is the entry-point for user space code, with the responsibility of creating, configuring and starting all other user space threads. Managing the different memory regions for the user space threads is the most complicated task that the root thread has to perform. Each new user thread needs a stack pointer and a location for the UTCB of the new thread. Additionally, all required memory regions must be mapped into the thread's address space, which is done via special IPC messages to the kernel.

In order to map memory to user threads, the root thread itself starts with all user space regions in its address space. When the root thread is created, these regions are magically mapped into the address space by the kernel. When the root thread starts, it has all user space memory regions available in its address space and can freely map it as required to other user threads. An example of starting a thread can be seen in listing 7.

4.5.3 Idle thread

In addition to the root and kernel thread, there is also an idle thread. The idle thread continuously runs the `wfi` (Wait For Interrupt) in a while loop. The scheduler also has a dedicated slot for the idle thread, which is of the lowest priority. In order for the idle thread to work on the ESP32-C3, the `TW` bit in the `mstatus` register must be unset so that the `wfi` instruction does not cause an illegal instruction exception from user-mode.

```c
void run_printer(kip_t *kip_ptr, utcb_t *utcb_ptr) {
    L4_ThreadId_t myself = {.raw = utcb_ptr->t_globalid};
    char *free_mem = (char *) get_free_base(kip_ptr);
    free_mem += 512;

    printer_id = (L4_ThreadId_t){.raw = TID_TO_GLOBALID(60)};
    user_log_printf("Starting printer thread with id %d\n", printer_id);

    L4_ThreadControl(printer_id, printer_id, L4_nilthread, myself, free_mem);

    map_user_text(kip_ptr, printer_id);

    L4_map((uint32_t)&printer_thread_stack_start,
        (char *)&printer_thread_stack_end - (char *)&printer_thread_stack_start,
        printer_id);
    L4_map((uint32_t)&user_threads_data_start,
        (char *)&user_threads_data_end - (char *)&user_threads_data_start,
        printer_id);

    L4_Msg_t msg;
    L4_MsgClear(&msg);
    L4_Word_t msgs[5] = {
        (L4_Word_t) printer,
        (L4_Word_t) &printer_thread_stack_end,
        (L4_Word_t)(((uint32_t) &printer_thread_stack_end) - ((uint32_t)
↪   &printer_thread_stack_start)),
        0,
        0
    };

    L4_MsgPut(&msg, 0, 5, msgs, 0, NULL);
    L4_MsgLoad(&msg);
    L4_Ipc(printer_id, myself, 0, (L4_ThreadId_t *)0);
}
```

Listing 7: Root thread creating a user thread, named `printer`

4.6 Scheduling and context switch

4.6.1 Timer interrupt

Scheduling is performed in the trap handler, triggered at every interrupt, resulting in scheduling being performed at least each timer interrupt. The system timer on the ESP32-C3 generates the timer interrupt (see section 4.2.4). An alarm threshold value can be configured to set at what value a timer interrupt should be fired. When the system timer is set to `period_mode`, an alarm will be fired whenever the counter reaches a multiple of the alarm threshold value.

The setup of timer interrupts can be seen in listings 8 and 9.

4.6.2 Scheduling

(This is the beginning of a section adapted from the specialization project)

Before a context switch occurs, the scheduler is responsible for finding a new thread to switch to. In F9, the scheduler is implemented with the `schedule_select()` function 10

`schedule_select()` loops through an array of scheduler slot entries 12 11, where the earlier entries are prioritized over later ones. Each scheduler slot entry can have a pointer to a TCB or a function pointer that can implement custom scheduling functionality. For each slot, the scheduler first checks if there is a valid and runnable TCB entry in the slot and attempts to schedule that TCB. If it cannot find a TCB in the slot, or if it is not runnable, it checks if a function pointer is present and attempts to execute it to select a new thread. If neither a TCB nor a custom scheduler function exists, the scheduler goes on to the next slot.

The kernel, root and idle system threads in F9 are set as scheduler slots with TCB entries. The first thread in the slot list is the kernel thread, which needs to have the highest priority because it is responsible for executing pending software interrupts. Another of the scheduler slot entries, identified by `SSI_NORMAL_THREAD`, is responsible for scheduling user threads. This scheduler entry contains a custom scheduler function `thread_sched()`, which fires `thread_select(root)` 13

The `thread_select` function goes through a tree of threads, where the root thread is, not surprisingly, the root of the tree. Each thread has a child, sibling, and parent entry. To traverse the tree, we need to get the child of the current node, then go through all the siblings of the child (which will be all the children of the current node), then go to the child of the child and repeat. For each node, if a thread is found to be runnable at any point, it is scheduled. A disadvantage

```
void system_timer_init() {
    /* 1. Set SYSTIMER_TARGETx_TIMER_UNIT_SEL to select the counter (UNIT0 or
↪   UNIT1) used for COMPx. */
    volatile uint32_t *systimer_target0_conf = REG(SYSTEM_TIMER_BASE +
↪   SYSTIMER_TARGET0_CONF_REG);
    *systimer_target0_conf &= ~(1 <<
↪   SYSTIMER_TARGET0_CONF_REG__SYSTIMER_TARGET0_TIMER_UNIT_SEL);
    /* 2. Set an alarm period (dt), and fill it to SYSTIMER_TARGETx_PERIOD. */
    int alarm = CONFIG_SYSTEM_TIMER_ALARM_THRESH;
    *systimer_target0_conf &= ~0x3ffffff; // zero out whatever is in period field
↪   (bits 0 - 25)
    *systimer_target0_conf |= alarm;
    /* 3. Set SYSTIMER_TIMER_COMPx_LOAD to synchronize the alarm period (dt) to
↪   COMPx, i.e. load the alarm period (dt) to COMPx. */
    volatile uint32_t *systimer_comp0_load = REG(SYSTEM_TIMER_BASE +
↪   SYSTIMER_COMP0_LOAD_REG);
    *systimer_comp0_load = 1;
    /* 4. Set SYSTIMER_TARGETx_PERIOD_MODE to configure COMPx into period mode.
↪   */
    *systimer_target0_conf |= (1 <<
↪   SYSTIMER_TARGET0_CONF_REG__SYSTIMER_TARGET0_PERIOD_MODE);
    /* 5. Set SYSTIMER_TARGETx_WORK_EN to enable the selected COMPx. COMPx starts
↪   comparing the count value with the sum of start value + n*dt (n = 1, 2,
↪   3...). */
    volatile uint32_t *systimer_conf_reg = REG(SYSTEM_TIMER_BASE +
↪   SYSTIMER_CONF_REG);
    *systimer_conf_reg |= (1 << SYSTIMER_CONF_REG__SYSTIMER_TARGET0_WORK_EN);
    /* 6. Set SYSTIMER_TARGETx_INT_ENA to enable timer interrupt. A
↪   SYSTIMER_TARGETx_INT interrupt is triggered when Unitn counts to start value
↪   + n*dt (n = 1, 2, 3...) set in step 2. */
    volatile uint32_t *systimer_int_ena = REG(SYSTEM_TIMER_BASE +
↪   SYSTIMER_INT_ENA_REG);
    *systimer_int_ena |= 1; // enable interrups for target 0
    ...
```

Listing 8: System timer setup part 1

```
    ...
    /* Map peripheral interrupt system timer to a CPU interrupt number */
    volatile uint32_t *interrupt_core0_systimer_target0_int_map_reg =
↪   REG(INTERRUPT_MATRIX_BASE + INTERRUPT_CORE0_SYSTIMER_TARGET0_INT_MAP_REG);
    *interrupt_core0_systimer_target0_int_map_reg = CONFIG_SYSTEM_TIMER_CPU_INTR;

    /* Set interrupt type to level-type interrupt, as stated in TRM 10.4.4. */
    volatile uint32_t *interrupt_core0_cpu_int_type = REG(INTERRUPT_MATRIX_BASE +
↪   INTERRUPT_CORE0_CPU_INT_TYPE_REG);
    *interrupt_core0_cpu_int_type &= ~(1 << CONFIG_SYSTEM_TIMER_CPU_INTR); // set
↪   to level type by writing zero

    /* Set the priority of the interrupt */
    volatile uint32_t *interrupt_core0_cpu_int_pri = REG(INTERRUPT_MATRIX_BASE +
↪   INTERRUPT_CORE0_CPU_INT_PRI_n_REG + 0x4 * (CONFIG_SYSTEM_TIMER_CPU_INTR -
↪   1));
    *interrupt_core0_cpu_int_pri = 10; // Set hightes priority for now TODO: Set
↪   a more sensible priority

    /* Set the threshold of interrupt priorities to 1, so that all interrupts are
↪   taken except those with priority = 0 */
    /* TODO: This should be in a more general interrupt_init() type of function
↪   */
    volatile uint32_t *interrupt_core0_cpu_int_thresh = REG(INTERRUPT_MATRIX_BASE
↪   + INTERRUPT_CORE0_CPU_INT_THRESH_REG);
    *interrupt_core0_cpu_int_thresh = 1;

    /* After mapping system timer to CPU interrupt, we enable the CPU interrupt
↪   number */
    volatile uint32_t *interrupt_core0_cpi_in_enable = REG(INTERRUPT_MATRIX_BASE
↪   + INTERRUPT_CORE0_CPU_INT_ENABLE_REG);
    *interrupt_core0_cpi_in_enable |= (1 << CONFIG_SYSTEM_TIMER_CPU_INTR);


    /* And finally start the counter */
    *systimer_conf_reg |= (1 << SYSTIMER_CONF_REG__SYSTIMER_TIMER_UNIT0_WORK_EN);
}
```

Listing 9: System timer setup part 2

```c
tcb_t *schedule_select()
{
        /* For each scheduler slot try to dispatch thread from it */
        for (int slot_id = 0; slot_id < NUM_SCHED_SLOTS; ++slot_id) {
                tcb_t *scheduled = slots[slot_id].ss_scheduled;

                if (scheduled && thread_isrunnable(scheduled)) {
                        /* Found thread, try to dispatch it */
                        return scheduled;
                } else if (slots[slot_id].ss_handler) {
                        /* No appropriate thread found (i.e. timeslice
                         * exhausted, no softirqs in kernel),
                         * try to redispatch another thread
                         */
                        scheduled = slots[slot_id].ss_handler(&slots[slot_id]);

                        if (scheduled) {
                                slots[slot_id].ss_scheduled = scheduled;
                                return scheduled;
                        }
                }
        }

        /* not reached (last slot is IDLE which is always runnable) */
        panic("Reached end of schedule()\n");
        return NULL;
}
```

Listing 10: Scheduler function

```c
typedef struct sched_slot {
        tcb_t *ss_scheduled;
        sched_handler_t ss_handler;
} sched_slot_t;
```

Listing 11: Scheduler slot struct

```c
static sched_slot_t slots[NUM_SCHED_SLOTS];
```

Listing 12: Scheduler array

```c
/* Select normal thread to run
 *
 * NOTE: all threads are derived from root
 */
static tcb_t *thread_select(tcb_t *parent)
{
	tcb_t *thr = parent->t_child;
	if (!thr)
		return NULL;

	while (1) {
		if (thread_isrunnable(thr))
			return thr;

		if (thr->t_child) {
			thr = thr->t_child;
			continue;
		}

		if (thr->t_sibling) {
			thr = thr->t_sibling;
			continue;
		}

		do {
			if (thr->t_parent == parent)
				return NULL;
			thr = thr->t_parent;
		} while (!thr->t_sibling);

		thr = thr->t_sibling;
	}
}
```

Listing 13: Thread select function

of this scheduling algorithm is that if two user threads are running without any IPC calls or interrupt delegation, one of the two threads will always be prioritized (the thread closest to the root in the thread tree).

(End of section adapted from the specialization project)

The priority of the scheduler is as follows:

1. `kernel thread`: the kernel thread is responsible for handling F9 software interrupts, so it needs the highest priority
2. `interrupts`: threads that are waiting to accept and IPC message converted from an interrupt (more on this in section 4.10). They need a high priority because they might need to handle the interrupt quickly.
3. `root thread`: the root thread is a user space thread responsible for setting up other user threads, so it needs a fairly high priority.
4. `IPC`: threads waiting for IPC operations
5. `normal threads`: all other user threads. The scheduler scans the thread three from the root thread and schedules the first runnable user thread it finds from there.
6. `idle thread`: if no other threads are available, the kernel schedules the idle thread, which only waits for interrupts in a while loop.

4.6.3 Context switch

When an interrupt or exception occurs, the program counter is set to the trap vector with an offset, where the offset depends on the kind of interrupt or exception that occurred. In F9 RISC-V, the trap vector is filled with identical instruction; a jump to a centralized trap handler. The entry point for this trap handler is defined in assembly. The assembly code increments the stack pointer and then stores all general-purpose registers on the stack. Before jumping to the rest of the trap handler defined in C code, the stack pointer is stored in the `a0` register, resulting in the stack pointer being available as the first argument in the trap handler function.

The TCB of the thread that is currently in execution is kept in the global `current` pointer. If the scheduler chooses a new thread, a context switch is initiated to switch to that thread. This is done by making the `current` point to the selected thread and swapping the active address space by configuring PMP regions (explained in more detail in 4.9.5). The `mepc` on the context on the TCB of the selected thread is loaded into the `mepc` register, which makes the program counter jump to the interrupted instruction of the selected thread when leaving the trap handler. Then the stack pointer `sp` is set from the stack pointer on the TCB. All general-purpose registers are loaded from the stack pointer, and finally, the stack pointer is decremented so that it points to the value it had when the trap happened.

4.7 System calls and F9 software interrupts

4.7.1 Overview

System calls in operating systems such as Linux are relatively complex. For example, the `fork` system call creates a new process by duplicating the calling process, which includes copying the entire address space [9]. System calls in F9 work on a much lower level. F9 has a total of two system calls. The simplest of the two is the `ThreadControL` system call, which can create or delete user threads. The other system call is the `IPC` system call which is used to send or receive IPC messages, as well handling special cases of IPC messages.

A system call is initiated by a thread invoking the `ecall` RISC-V instruction, resulting in a user or machine environment call exception. Hardware registers `A0-A5` are used to configure the system call. The system call number deciding if it is a `ThreadControl` or `IPC` system call is configured via the `A0` register. The meaning of the rest of the registers `A1-A5` depends on the kind of system call (see table 1).

register	IPC	ThreadControl
A0	`SYS_IPC`	`SYS_THREAD_CONTROL`
A1	id of thread to send IPC to	thread id to create / delete
A2	id of thread to receive IPC from	thread id of address space owner
A3	timeout value	thread id of scheduler (not implemented)
A4	not used	thread if of pager (not implemented)
A5	not used	UTCB location

Table 1: Meaning of registers for the two system calls in F9

When an `ecall` instruction is executed by a thread, an environment call exception is triggered and control transferred to the trap handler. The kernel detects an environment call exception, which means that a system call should be executed, but it does not handle the system call itself. Instead, it schedules a system call in the form of an F9 software interrupt [10]; it sets a global variable `caller` to the caller thread and sets the state of the caller thread to `SVC_BLOCKED`. Next, the kernel initiates scheduling and context switch as usual, which will result in a context

[9]in reality, with copy-on-write in Linux, the address space is copied as required

[10]Not to be confused with the interrupt type named software interrupt in RISC-V

switch to the kernel thread.

4.7.2 Kernel thread

In F9, the kernel consists of the trap handler and the kernel thread. While the trap handler is responsible for handling interrupts and scheduling, the kernel thread handles all software interrupts. Software interrupts include system calls and kernel timer events.

When a system call is requested through the `ecall` instruction, the kernel thread will be prioritized by the scheduler and a context switch to the kernel thread wil be performed. The kernel thread continuously performs two operations in a while loop; it handles all pending software interrupts and executes the `ecall` instruction to give up its timeslice to any other thread. Pending software interrupts are kept in an array of `softirq_t` structures, which has an integer indicating if it is pending and a handler function. The kernel thread loops through this table and checks if any software interrupts are pending, as soon as it finds one, it fires the corresponding handler, which could for example be a system call.

The flow of F9 software interrupts in the case of a system call is shown in figure 9

4.8 IPC

4.8.1 Overview of IPC

At the heart of F9 (and L4 microkernels in general) is IPC. In F9, IPC allows user threads to implement much of the functionality that would otherwise be implemented with system calls (which are handled in kernel space). In F9, IPC is initiated by an IPC system call, using the registers shown in table 1. Instead of having many system calls for different kinds of IPC, F9 prefers to have a single system call for IPC that can handle many cases. Almost any combination of threads and timeouts can be used, where some special thread IDs are used to signify no thread or any thread. When using the id that signifies any thread, the IPC call will send or receive one message to or from any thread. For an overview of what an IPC transfer between two threads looks like, see figure 10.

In addition to the “standard” ways of doing IPC from user thread(s) to user thread(s), there are also some special cases.

- IPC is used to configure whether peripheral interrupts should be sent to a thread (more on this here 4.10).

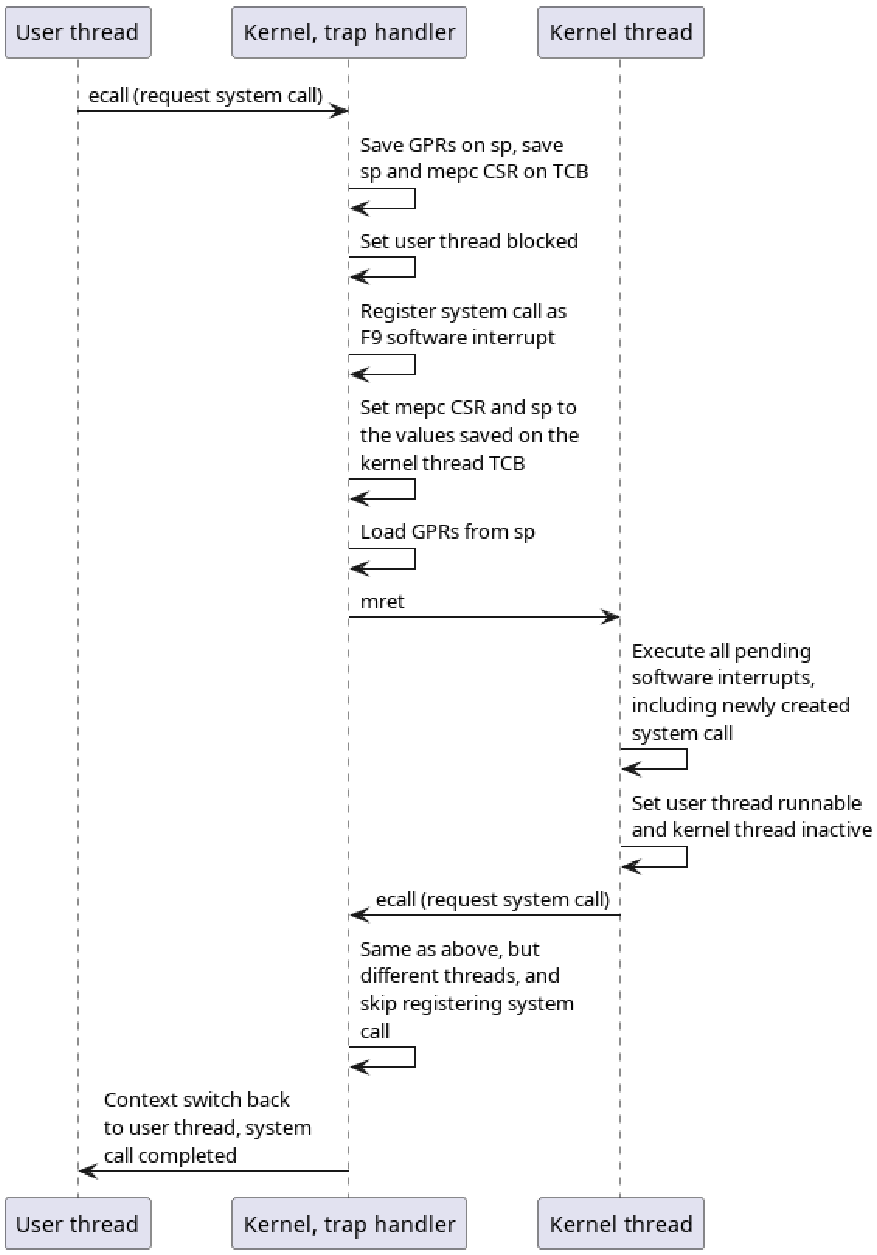

User thread
Kernel, trap handler
Kernel thread
ecall (request system call)
Save GPRs on sp, save sp and mepc CSR on TCB
Set user thread blocked
Register system call as F9 software interrupt
Set mepc CSR and sp to the values saved on the kernel thread TCB
Load GPRs from sp
mret
Execute all pending software interrupts, including newly created system call
Set user thread runnable and kernel thread inactive
ecall (request system call)
Same as above, but different threads, and skip registering system call
Context switch back to user thread, system call completed
User thread
Kernel, trap handler
Kernel thread

Figure 9: The flow of F9 software interrupts in, using a system call as an example. Other threads might be scheduled before control is returned to the user thread. This is an updated figure from the specialization project that preceeded this thesis [30] .

- IPC is also used to implement the MAP, GRANT and FLUSH/UNMAP operations on address spaces. Information on what region to modify is sent through the `typed` words of the IPC message
- IPC is used to start threads. While the thread control system call is used to create threads, they are not started after this creation. Additional configuration is usually performed, such as mapping in required memory regions, and then after this, the thread is started by sending a special IPC message. The kernel detects this IPC message by checking if the `to thread` is in the inactive state and that the IPC message contains a certain number of untyped arguments. All required information should be passed in the IPC message, such as the initial program counter, the stack pointer, and space for the UTCB.
- IPC allows multiple user threads to output data on a single Universal Asynchronous Receiver/Transmitter (UART) controller. Data of variable arguments are sent to the kernel via IPC, and the kernel will unpack the variable arguments and print the data using its logging system.

4.8.2 IPC API

The IPC interface described in the reference manual [31] is used to send IPC messages. User threads will store data in a data structure representing an IPC message. When all data is copied over to this message, the function `L4_MsgLoad` should be invoked, which will load all the data in the message into Message Registers (MR).

To send an IPC message from a user thread, `thread 1`, to another user thread, `thread 2`, first `thread 1` invokes the `L4_Ipc` function with the `to` argument set to the thread id of `thread 2` and the `FromSpecifier` argument set to the null thread id. `thread 2` then needs to receive the IPC message with a corresponding `L4_Ipc` invocation, setting the `to` field to the null thread id and setting `FromSpecifier` to the thread id of thread 1. This will result in a single message being sent from `thread 1` to `thread 2`, with no reply. If `thread 1` wants a reply, it can set the `FromSepcifier` argument to the id of `thread 2`, where `thread 2` needs to send an additional IPC after it has received the first message.

There is an argument to be made that using the API from the L4 eXperimental reference manual [31] can negate some of the optimizations to make IPC fast in L4. The argument here is that the IPC in L4 is designed to transfer data as fast as possible, going as far as using hardware registers to speed up the transfer. By writing and reading data into an auxiliary data structure such as `L4_Msg_t` and then loading that data structure, it defeats the purpose of other optimizations (see section 4.3.5 in [17])

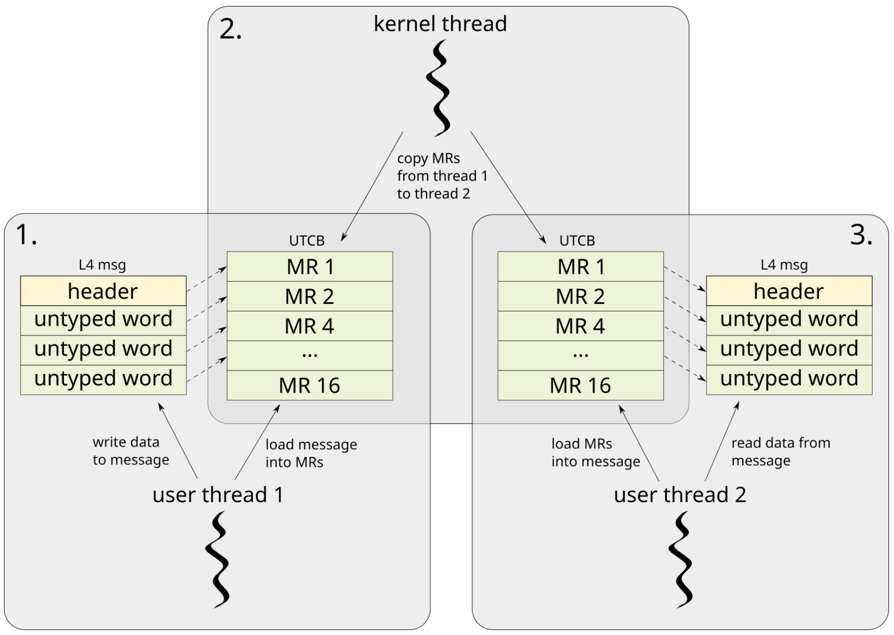


Figure 10: Overview of IPC transfer. Transfer begins with an L4 message that has a header block and a number of typed and untyped words. An untyped word is just 32 bits of data without any other information. The transfer is performed in three steps as indicated by the `1.` `2.` and `3.` markers in the figure. In the first block, user thread 1 creates an L4 message and loads it into the Message Registers (MR) on its UTCB. In the second block, the kernel thread transfers the MRs on the UTCB of user thread 1, to the MRs belonging to user thread 2. In the third block, user thread 2 reads the MRs into an L4 messages and can then use them freely. The header block has information on how many untyped and typed words are present in the message, so in this case it would indicate that there are three untyped words and no typed words.

```
__USER_TEXT L4_MsgTag_t L4_Ipc(L4_ThreadId_t to,
                               L4_ThreadId_t FromSpecifier,
                               L4_Word_t Timeouts,
                               L4_ThreadId_t *from)
{
    L4_MsgTag_t result;
    L4_ThreadId_t from_ret;

    __asm__ __volatile__(
        "mv a0, %[SYS_NUM]\n\t\
        mv a1, %[to]\n\t\
        mv a2, %[from_specifier]\n\t\
        mv a3, %[timeout]\n\t\
        ecall\n\t\
        mv %[from_ret], a0\n\t"
        : [from_ret] "=r" (from_ret)
        : [SYS_NUM] "r"(SYS_IPC), [to] "r"(to), [from_specifier]
↪   "r"(FromSpecifier), [timeout] "r"(Timeouts)
        : "a0", "a1", "a2", "a3");

    result.raw = __L4_MR0;
    if (from)
        *from = from_ret;
    return result;
}
```

Listing 14: L4$_{\text{Ipc}}$ function, for sending IPC from user space

4.8.3 Internals of IPC

IPC is sent by loading MRs with data and initiating an IPC transfer. In L4, MRs can be implemented in memory, hardware registers or a combination of both. In F9, the first 8 MRs in an IPC message use hardware registers, while the next eight use memory in the UTCB. Each MR holds a word, which can be either typed or untyped. Untyped words are data without any more information, so it is the task of the communicating threads to decide what to do with it. Typed words are used to MAP or GRANT regions in address spaces.

Since IPC is implemented by a system call in F9, the kernel thread has the responsibility of performing IPC operations (the kernel thread is responsible for handling all F9 software interrupts, which include system calls). There are two kernel functions that are especially noteworthy in regards to performing the IPC transfer, `sys_ipc` and `do_ipc` (See listings 15, 16 and 17).

- `sys_ipc`: Since there are several special IPC operations (such as logging, address space operations, and starting of threads) in addition to the normal case of two user threads passing messages to each other, the `sys_ipc` functions need to orchestrate what to do in each case. The `sys~ipc` function checks the `to_tid` (the id of the thread to send the IPC message to) argument if it is equal to one of the special thread ids. If it is the case, `sys_ipc` calls the corresponding function. For example, if `to_tid` is equal to the special thread id for logging user space messages, then `sys_ipc` calls the `user_log` function to perform the logging. If `to_tid` is defined but does not match any of the special thread ids, the `sys_ipc` function will call `do_ipc` to perform a normal IPC transfer. If the `sys_ipc` function receives a request where the `to_tid` is equal to the null thread but the `from_tid` argument is defined (the id of the thread to receive an IPC message from), it checks if there exists some other thread with thread id `from_tid` that is trying to send an IPC message to the caller. If it finds a thread waiting to send an IPC to the caller, it performs that IPC transfer. Otherwise, it sets the state of the caller to `RECV_BLOCKED` (see listing 5) and the `ipc_from` field (see listing 5) to `from_tid`.
- `do_ipc`: The `do_ipc` function performs the transfer of data between the threads. The first word in all IPC messages contains a tag which describes how many typed and untyped words are contained in the IPC message. The `do_ipc` function will look at the tag and copy over the corresponding number of MRs between the threads. After copying, if a `from_tid` was defined with the request, it also set the caller thread to `RECV_BLOCKED`, and sets the `ipc_from` field on the TCB of the caller to `from_tid`.

Lastly, the scheduler also plays a part since the kernel can indicate to the scheduler that it should prioritize a thread because its IPC state is resolved.

4.9 Memory protection in F9

4.9.1 Address spaces as memory protection

In F9 RISC-V, memory protection is abstracted in the concept of address spaces. Each thread is related to an address space, where multiple threads can relate to/share the same address space. Conceptually, an address space represents all the addresses accessible to the owner thread(s) regarding memory protection. Address spaces are implemented as lists of flex pages, where a flex page is a contiguous region of memory that represent some protected memory region. The idea is that as a thread executes, all memory accesses must conform to the flex pages in its address space.

Enabling memory protection for a flex page is done with the RISC-V PMP [4]. Whenever a context switch occurs to a new thread, the new thread's address space is configured on the PMP, so that the thread can access the addresses on its address space without an access fault occurring. Of course, the PMP has a limited amount of regions available, so the kernel needs to make some decisions on which flex pages to prioritize and potentially swap out (See section 4.9.5). The definitions for address spaces and flex pages can be seen in listings 18 and 19 respectively.

4.9.2 Memory pools

In addition to address spaces and flex pages, there is a third concept related to memory protection: memory pools. Memory pools are used to define what static regions of memory are available and what permissions they should have. For example, all user space data is placed in a specific section by the linker script, and there is a memory pool that covers the region of that section with appropriate permissions and tags. This is useful in several places; one of them is when mapping all user space code to the root thread. The kernel only needs to iterate all existing memory pool entries and check if they have the appropriate tags, and then map that region into the root thread's address space. The macro used for defining memory pools can be viewed in listing 20, while some declarations can be viewed in listing 21.

```c
void sys_ipc(uint32_t* sc_param1)
{
    /* TODO: Checking of recv-mask */
    tcb_t *to_thr = NULL;
    l4_thread_t to_tid = sc_param1[REG_A1], from_tid = sc_param1[REG_A2];
    uint32_t timeout = sc_param1[REG_A3];

    if (to_tid == L4_NILTHREAD &&
        from_tid == L4_NILTHREAD) {
        caller->state = T_INACTIVE;
        if (timeout)
            sys_ipc_timeout(timeout);
        return;
    }

    if (to_tid != L4_NILTHREAD) {
        to_thr = thread_by_globalid(to_tid);

        if (to_tid == TID_TO_GLOBALID(THREAD_LOG)) {
            user_log(caller);
            caller->state = T_RUNNABLE;
            return;
        ...
        ...
        } else  {
            /* No waiting, block myself */
            caller->state = T_SEND_BLOCKED;
            caller->utcb->intended_receiver = to_tid;
            dbg_printf(DL_IPC,
                       "IPC: %t sending\n", caller->t_globalid);

            if (timeout)
                sys_ipc_timeout(timeout);

            return;
        }
    }
    ...
```

Listing 15: sys_ipc function part 1. This is the part of sys_ipc that checks if to_tid is equal to one of the special thread ids, otherwise it performs a “normal” IPC transfer.

```c
void sys_ipc(uint32_t* sc_param1)
{
    ...
    if (from_tid != L4_NILTHREAD) {
        tcb_t *thr = NULL;

        if (from_tid == L4_ANYTHREAD) {
            /* Find out if there is any sending thread waiting
             * for caller
             */
            for (int i = 1; i < thread_count; ++i) {
                thr = thread_map[i];
                if (thr->state == T_SEND_BLOCKED &&
                    thr->utcb->intended_receiver ==
                    caller->t_globalid) {
                    do_ipc(thr, caller);
                    return;
                }
            }
        } else if (from_tid != TID_TO_GLOBALID(THREAD_INTERRUPT)) {
            thr = thread_by_globalid(from_tid);

            if (thr->state == T_SEND_BLOCKED &&
                thr->utcb->intended_receiver ==
                caller->t_globalid) {
                do_ipc(thr, caller);
                return;
            }
        }

        /* Only receive phases, simply lock myself */
        caller->state = T_RECV_BLOCKED;
        caller->ipc_from = from_tid;

        if (from_tid == TID_TO_GLOBALID(THREAD_INTERRUPT)) {
            /* Threaded interrupt is ready */
            user_interrupt_handler_update(caller);
        }

        if (timeout)
            sys_ipc_timeout(timeout);

        dbg_printf(DL_IPC, "IPC: %t receiving\n", caller->t_globalid);

        return;
    }

    caller->state = T_SEND_BLOCKED;
}
```

Listing 16: sys_ipc function part 2. This is the part of sys_ipc that happens if to_thr is equal to the null thread, but from_tid is not. This means that the caller of the IPC message expects to receive an IPC message from the thread with the id equal to from_tid.

```c
static void do_ipc(tcb_t *from, tcb_t *to)
{
    ...
    /* Copy tag of message */
    ipc_msg_tag_t tag = { .raw = ipc_read_mr(from, 0) };
    int untyped_last = tag.s.n_untyped + 1;
    int typed_last = untyped_last + tag.s.n_typed;
    ...
    ipc_write_mr(to, 0, tag.raw);
    /* Copy untyped words */
    for (untyped_idx = 1; untyped_idx < untyped_last; ++untyped_idx) {
        ipc_write_mr(to, untyped_idx, ipc_read_mr(from, untyped_idx));
    }

    typed_item_idx = -1;
    /* Copy typed words
     * FSM: j - number of byte */
    for (typed_idx = untyped_idx; typed_idx < typed_last; ++typed_idx) {
        uint32_t mr_data = ipc_read_mr(from, typed_idx);

        /* Write typed mr data to 'to' thread */
        ipc_write_mr(to, typed_idx, mr_data);

        if (typed_item_idx == -1) {
            /* If typed_item_idx == -1 - read typed item's tag */
            typed_item.raw = mr_data;
            ++typed_item_idx;
        } else if (typed_item.s.header & IPC_TI_MAP_GRANT) {
            /* MapItem / GrantItem have 1xxx in header */
            int ret;
            typed_data = mr_data;

            ret = map_area(from->as, to->as,
                           typed_item.raw & 0xFFFFFFC0,
                           typed_data & 0xFFFFFFC0,
                           (typed_item.s.header & IPC_TI_GRANT) ?
                               GRANT : MAP,
                           thread_isprivilived(from));
            typed_item_idx = -1;

            ...
        }
    }
    ...
```

Listing 17: `do_ipc` function. This function performs the copying of words from one thread to another thread during IPC. Note that if there are typed words present, a map operation is performed (See section 4.9.4). Some parts omitted.

```c
typedef struct {
    uint32_t as_spaceid;
    struct fpage *first;

    struct fpage *mpu_first;
    struct fpage *mpu_stack_first;
    uint32_t shared;
} as_t;
```

Listing 18: Address space struct definition

```c
struct fpage {
    struct fpage *as_next;
    struct fpage *map_next;
    struct fpage *mpu_next;

    union {
        struct {
            uint32_t base;
            uint32_t mpid : 6;
            uint32_t flags : 6;
            uint32_t size : 16;
            uint32_t rwx : 4;
        } fpage;
        uint32_t raw[2];
    };

    int used;
};
```

Listing 19: Fpage struct definition

```
#define DECLARE_MEMPOOL(name_, start_, end_, flags_, tag_) \
{                        \
    .name = name_,               \
    .start = (memptr_t) (start_),       \
    .end  = (memptr_t) (end_),      \
    .flags = flags_,             \
    .tag = tag_                  \
}

#define DECLARE_MEMPOOL_2(name, prefix, flags, tag) \
    DECLARE_MEMPOOL(name, \
            &(prefix ## _start), &(prefix ## _end), flags, tag)
#else
```

Listing 20: Macros for declaring mempools

```
static mempool_t memmap[] = {
    DECLARE_MEMPOOL_2("KTEXT", kernel_text,
        MP_KR | MP_KX | MP_NO_FPAGE, MPT_KERNEL_TEXT),
    DECLARE_MEMPOOL_2("UTEXT", user_text,
        MP_UR | MP_UX | MP_MEMPOOL | MP_MAP_ALWAYS, MPT_USER_TEXT),
    ...
    DECLARE_MEMPOOL_2("UDATA", user_data,
        MP_UR | MP_UW | MP_MEMPOOL | MP_MAP_ALWAYS, MPT_USER_DATA),
    DECLARE_MEMPOOL_2("UBSS",  user_bss,
        MP_UR | MP_UW | MP_MEMPOOL  | MP_MAP_ALWAYS, MPT_USER_DATA),
    ...
}
```

Listing 21: Declaration of mempools

4.9.3 More flexible flex pages

In L4, a flex page is a mapping between virtual addresses and physical addresses. The mappings described by flex pages are realized using hardware pages. In F9, however, which does not support virtual memory or paging, a flex page is a protected region of memory. It is a significant departure from traditional L4 microkernels since the concept of flex pages was meant to be used with virtual memory and an MMU in the first place. An illustration of a potential situation in F9 RISC-V where two threads have address spaces where a flex page is shared can be seen in figure 11.

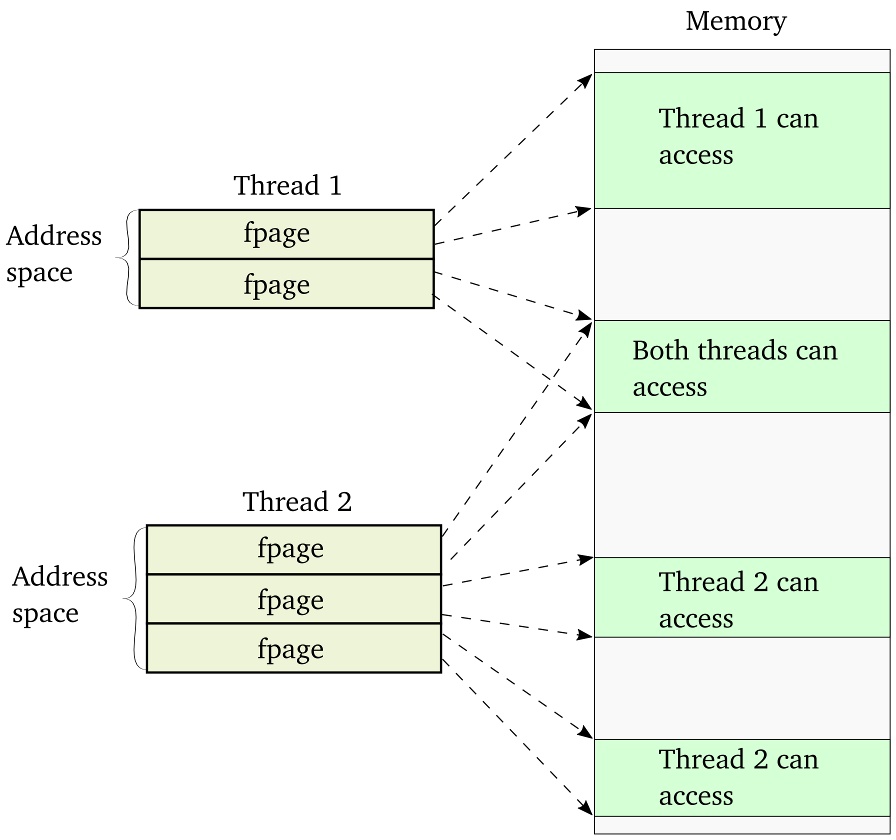


Figure 11: Snapshot of the memory management data structures of some hypothetical situation. Here thread 1 and thread 2 have multiple flex pages in their address spaces, sharing one flex page between them. This means that both threads can access the memory region described by the shared flex page.

A limitation on the ARMv7 MPU is that the protected memory regions must have a size being a power of two, and a base address being a multiple of the size. This means that software must split up regions into several smaller ones in order to cover regions that do not satisfy this requirement. It can have an impact on code complexity and performance since there are relatively few MPU regions.

The RISC-V PMP works in a similar manner as the ARM MPU, but it does not share the limitations on address alignment and size. This means that the RISC-V port of F9 does not have to split up flex pages to fit a size that is not aligned or a power of two, which has several advantages.

1. A memory region can always be covered by a single flex page, which reduces the total number of flex pages in existence. This reduces memory consumption in the kernel.
2. When the kernel thread is performing mappings between address spaces, less work on splitting up flex pages means less work for the kernel.
3. When the kernel thread is creating new flex pages for memory regions, less work on splitting up flex pages means less work for the kernel.

The port could have made use of the NAPOT addressing mode on the RISC-V PMP, and then used the same algorithms that already existed in the ARM version of F9. An advantage of this approach would be double the amount PMP regions available since only one `pmpaddri` register is needed for a region when using NAPOT addressing mode. However, the advantages of not having to split up flex pages are assumed to outweigh more available PMP regions.

4.9.4 Operations on flex pages and address spaces

The function `assign_fpages_ext` (See listings 22, 23 and 24) is used to initialize address spaces with flex pages. It again uses the `create_fpage_chain` function to create the new flex pages that should be inserted. In the ARM version of F9, the `create_fpage_chain` function is needed since flex pages have constraints on the base pointer and size (as explained in section 4.9.3). In the port however the `create_fpage_chain` function doesn't create a chain but rather just creates a new fpage, making the code simpler.

Even without having to split flex pages, there is still some complexity involved when creating new address spaces. The `assign_fpages_ext` function takes into consideration if other flex pages exist in the area where the new flex pages should be created. If that is the case, it creates flex pages around the existing ones. However, a limitation is that it does not adjust permissions on the existing flex pages, which could lead to problems. An alternative approach could be not performing the mapping if other flex pages exist in the memory region.

Mapping flex pages from thread to thread is initiated via IPC (See listing 17), and is performed via the `map_area` function (See listings 25, 26). The `map_area` function differentiates between privileged and unprivileged threads. If it is a privileged thread, the “mapping” is done via the `assign_fpages_ext()` function, which does a “magic” mapping of flex pages into the privileged thread. Otherwise, it performs the somewhat complicated action of finding the flex pages for the requested memory region, potentially splitting them up, and inserting them into the other address space.

```
int assign_fpages_ext(int mpid, as_t *as, memptr_t base, size_t size,
                      fpage_t **pfirst, fpage_t **plast)
{
    fpage_t **fp;
    memptr_t  end;

    if (size <= 0)
        return -1;

    /* if mpid is unknown, search using base addr */
    if (mpid == -1) {
        if ((mpid = mempool_search(base, size)) == -1) {
            /* Cannot find appropriate mempool, return error */
            dbg_printf(DL_MEMORY,
                       "ERROR: Cannot find appropriate mempool. mpid: %d, base:
↪    0x%p, size: %d\n",
                       mpid, base, size);
            return -1;
        }
    }
    ...
```

Listing 22: `assign_fpages_ext` function part 1

4.9.5 Switching address spaces

A context switch in F9 means switching from executing the current thread to a new thread. This procedure includes switching the address space, which requires manipulating regions on the PMP unit. Old regions should be disabled, and new regions configured and enabled. Since PMP has a limited number of regions, it might be the case that a thread has more flex pages than there are available PMP regions, so we need some mechanism to decide what flex page to configure on the PMP unit. F9 manages a FIFO queue of flex pages that should be scheduled, where the first flex page has the highest priority. However, there are also some flex pages that we need to have configured on the PMP unit at all times, such as the memory region for the stack. Each address space in F9 organize the flex page in three lists (these can also be seen in figure 18 and 19):

1. A list of all the flex pages in the thread’s address space, including all the flex

```c
...
end = base + size;

if (as) {
    /* find unmapped space */
    fp = &as->first;
    while (base < end && *fp) {
        if (base < FPAGE_BASE(*fp)) {
            fpage_t *first = NULL, *last = NULL;
            size = (end < FPAGE_BASE(*fp) ? end : FPAGE_BASE(*fp)) - base;

            dbg_printf(DL_MEMORY,
                       "MEM: fpage chain %s [b:%p, sz:%p] as %p\n",
                       mempool_getbyid(mpid)->name, base, size, as);

            create_fpage_chain(mempool_align(mpid, base),
                               mempool_align(mpid, size),
                               mpid, &first, &last);

            last->as_next = *fp;
            *fp = first;
            fp = &last->as_next;

            if (!*pfirst)
                *pfirst = first;
            *plast = last;

            base = FPAGE_END(*fp);
        } else if (base < FPAGE_END(*fp)) {
            if (!*pfirst)
                *pfirst = *fp;
            *plast = *fp;

            base = FPAGE_END(*fp);
        }

        fp = &(*fp)->as_next;
    }
    ...
```

Listing 23: `assign_fpages_ext` function part 2

```
        ...
        if (base < end) {
            fpage_t *first = NULL, *last = NULL;
            size = end - base;

            dbg_printf(DL_MEMORY,
                       "MEM: fpage chain %s [b:%p, sz:%p] as %p\n",
                       mempool_getbyid(mpid)->name, base, size, as);

            create_fpage_chain(mempool_align(mpid, base),
                               mempool_align(mpid, size),
                               mpid, &first, &last);

            *fp = first;

            if (!*pfirst)
                *pfirst = first;
            *plast = last;
        }
    } else {
        dbg_printf(DL_MEMORY,
                   "MEM: fpage chain %s [b:%p, sz:%p] as %p\n",
                   mempool_getbyid(mpid)->name, base, size, as);

        create_fpage_chain(mempool_align(mpid, base),
                           mempool_align(mpid, size),
                           mpid, pfirst, plast);
    }

    return 0;
}
```

Listing 24: `assign_fpages_ext` function part 3

```c
int map_area(as_t *src, as_t *dst, memptr_t base, size_t size, /*  */
             map_action_t action, int is_priviliged)
{
    /* Most complicated part of mapping subsystem */
    memptr_t end = base + size, probe = base;
    fpage_t *fp = src->first, *first = NULL, *last = NULL;
    int first_last_equal = 0;

    if (is_priviliged) {
        assign_fpages_ext(-1, src, base, size, &first, &last);
        if (src == dst) {
            /* Maps to itself, ignore other actions */
            return 0;
        }
    } else {
        if (src == dst) {
            /* Maps to itself, ignore other actions */
            return 0;
        }
        while (fp) {
            if (!first && addr_in_fpage(base, fp, 0)) {
                first = fp;
            }

            if (!last && addr_in_fpage(end, fp, 1)) {
                last = fp;
                break;
            }

            if (first) {
                /* Check weather if addresses in fpage list
                 * are sequental */
                if (!addr_in_fpage(probe, fp, 1)) {
                    dbg_printf(DL_EMERG,
                            "ERROR: Addresses in fpage list not sequential");
                    return -1;
                }

                probe += FPAGE_SIZE(fp);
            }

            fp = fp->as_next;
        }
    }
...
```

Listing 25: `map_area` function part 1

```
...
    if (!last || !first) {
        dbg_printf(DL_EMERG,
                   "ERROR: Requested map area not in address space of mapper
↪   thread, or other error\n");
        return -1;
    }

    if (first == last)
        first_last_equal = 1;

    first = split_fpage(src, first, base);
    /* If we performed a split, then we need to grab the second fpage of the new
↪   pair of fpages from the split */
    if (first != last) {
        first = first->as_next;
    }
    /* If first was equal to last, then last fpage is invalidated at this point,
↪   so we update it */
    if (first_last_equal) {
        last = first;
    }
    last = split_fpage(src, last, end);
    /* If first was equal to last, then first fpage is invalidated at this point,
↪   so we update it */
    if (first_last_equal) {
        first = last;
    }
    if (!last || !first) {
        return -1;
    }
    /* Map chain of fpages */
    fp = first;
    while (fp != last) {
        map_fpage(src, dst, fp, action);
        fp = fp->as_next;
    }
    map_fpage(src, dst, fp, action);
    return 0;
}
```

Listing 26: `map_area` function part 2

pages in the other lists. The `first` field on the address space is the entry to the list, and subsequent entries are on the `as_next` field on the flex pages.

2. A list of flex pages covering the thread's stack. The `mpu_stack_first` field on the address space is the entry to the list, and subsequent entries are on the `mpu_next` field on the flex pages.

3. A list of all other flex pages that are (or should be) on the PMP. The `mpu_first` field on the address space is the entry to the list, and subsequent entries are on the `mpu_next` field on the flex pages.

The function that decides what flex pages that goes on the PMP and which does not is the responsibility of the `as_setup_mpu` function, which can be viewed in listings 27, 28 and 29. This function also does the actual setup on the PMP when appropriate.

When `as_setup_mpu` runs on an address space for the first time, it sets up lists to manage what flex pages should be prioritized to be configured on the PMP. First, it finds all flex pages covering the thread's stack and program counter. These flex pages are prioritized and put at the front of the queue. Next, flex pages that are tagged with `FPAGE_ALWAYS` are prioritized. Lastly, all other flex pages are put at the back of the queue. A visualization can be viewed in figure 12.

```c
void as_setup_mpu(as_t *as, memptr_t sp, memptr_t pc,
                  memptr_t stack_base, size_t stack_size)
{
    fpage_t *mpu[8] = { NULL };
    fpage_t *fp;
    int mpu_first_i; // The first available entry in mpu[] after stack fpages are
↪   inserted
    int i, j;

    fpage_t *mpu_stack_first = NULL;

    /* Iterator for walking through stack */
    memptr_t start = stack_base;
    memptr_t end = stack_base + stack_size;

    /* Find stack fpages */
    fp = as->first;
    i = 0;
    while (i < 8 && fp && start < end) {
        if (addr_in_fpage(start, fp, 0)) {
            if (!mpu_stack_first)
                mpu_stack_first = fp;

            mpu[i++] = fp;
            start = FPAGE_END(fp);
        }
        fp = fp->as_next;
    }

    as->mpu_stack_first = mpu_stack_first;
    mpu_first_i = i; // Save the place in mpu[] after stack fpages have been
↪   inserted
    ...
```

Listing 27: as_setup_mpu function part 1

```
...
/*
 * We walk through fpage list
 * mpu_fp[0] are pc
 * mpu_fp[1] are always-mapped fpages
 * mpu_fp[2] are others
 *
 * What we do here is setup three linked lists of fpages that are meant to be setup by
↪ MPU.
 * For each fpage in the address space, we check what "type" it is. Is it overlapping the
↪ PC,
 * should it be always mapped, or all other fpages. For each of these categories, we build
 * a list of ->mpu_next entries, linking them together.

 * When we are done building the three different linked lists, we chain them together.
 * Fpages overlapping the PC goes first, then the alwyas mapped fpages, then all the
↪ others.
 */
fp = as->mpu_first;
if (!fp) {
    fpage_t *mpu_first[3] = {NULL};
    fpage_t *mpu_fp[3] = {NULL};

    fp = as->first;
    /* Build three linked lists of ->mpu_next entries */
    while (fp) {
        int priv = 2;

        if (addr_in_fpage(pc, fp, 0)) {
            priv = 0;
        } else if (fp->fpage.flags & FPAGE_ALWAYS) {
            priv = 1;
        }

        if (!mpu_first[priv]) {
            mpu_first[priv] = fp;
            mpu_fp[priv] = fp;
        } else {
            mpu_fp[priv]->mpu_next = fp;
            mpu_fp[priv] = fp;
        }

        fp = fp->as_next;
    }

    /* Chain together the three different linked lists to one linked list */
    if (mpu_first[1]) {
        mpu_fp[1]->mpu_next = mpu_first[2];
    } else {
        mpu_first[1] = mpu_first[2];
    }
    if (mpu_first[0]) {
        mpu_fp[0]->mpu_next = mpu_first[1];
    } else {
        mpu_first[0] = mpu_first[1];
    }
    /* Set the ->mpu_first of the address space to the final linked list, beginning with
       fpages overlapping with the PC */
    as->mpu_first = mpu_first[0];
}
...
```

Listing 28: `as_setup_mpu` function part 3

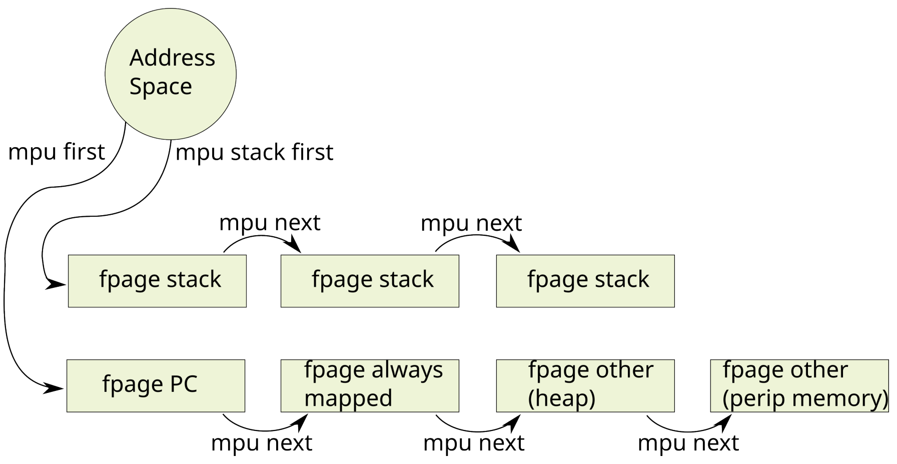


Figure 12: Snapshots of the fpage queue. The queue is actually two queues, where the fpages that contain the stack are in one queue and all others in another queue. The stack queue is prioritized. Definition of address spaces and fpages can be viewed here 18 19.

4.9.6 Access faults

In RISC-V, an access fault occurs when an instruction accesses or writes some memory that it does not have proper permissions for in regards to memory protection. There are a number of types of access faults that depend on what type of permission was breached. The access fault manifests itself as an exception where the exception number in the `mcause` CSR corresponds to the type of access fault that happened. In the RISC-V port of F9, the kernel trap handler is responsible for handling such access faults and taking appropriate action.

When a thread encounters an access fault, control is immediately given up to the kernel in the form of an exception. The kernel detects that it was indeed an access fault that happened. At this point, there could be an unrecoverable error for the running thread, or we could be in the situation that the running thread has a flex page with the correct permissions that is not configured on the PMP at this time since the PMP has a limited amount of regions. Therefore, the kernel will run the `mpu_select_lru` (See listing 30) function to search through all unmapped flex pages on the address space of the running thread and search for a flex page that would prevent the access fault from happening. If the kernel finds such a flex page, it configures the PMP with it. If it does not finds such flex page, the kernel panics and ends execution.

```
    ...
     /*
      * Here, we walk through the ->mpu_first linked list (which was either newly
↪   created, or
      * persisted from a previous run of this function with the same address
↪   space). We check
      * if the fpage is a stack fpage, in which case it should already have been
↪   added to mpu[]
      * with an index lower than mpu_first_i. As long the fpage is NOT a stack
↪   fpage, then we
      * add it to the mpu[] array, with an index AFTER mpu_first_i.
      */
    for (fp = as->mpu_first; i < 8 && fp; fp = fp->mpu_next) {
        for (j = 0; j < mpu_first_i; j++) {
            if (fp == mpu[j]) {
                break;
            }
        }
        if (j == mpu_first_i) {
            mpu[i++] = fp;
        }
    }

    as->mpu_first = mpu[mpu_first_i];

    /* Setup MPU stack regions */
    for (j = 0; j < mpu_first_i; ++j) {
        mpu_setup_region(j, mpu[j]);

        if (j < mpu_first_i - 1)
            mpu[j]->mpu_next = mpu[j + 1];
        else
            mpu[j]->mpu_next = NULL;
    }

    /* Setup MPU fifo regions */
    for (; j < i; ++j) {
        mpu_setup_region(j, mpu[j]);

        if (j < i - 1)
            mpu[j]->mpu_next = mpu[j + 1];
    }

    /* Clean unused MPU regions */
    for (; j < 8; ++j) {
        mpu_setup_region(j, NULL);
    }
}
```

Listing 29: `as_setup_mpu` function part 3

```c
/* mpu_select_lru checks if address addr is in the addres space as, and then
   sets up an mpu region for it. It uses the FIFO list as->mpu_first to find
   what fpages to remove if there are not enough mpu entries */
int mpu_select_lru(as_t *as, uint32_t addr)
{
    fpage_t *fp = NULL;
    int i;

    /* Kernel fault? */
    if (!as)
        return 1;

    if (addr_in_mpu(addr))
        return 1;

    fp = as->first;
    while (fp) {
        if (addr_in_fpage(addr, fp, 0)) {
            fpage_t *sfp = as->mpu_stack_first;

            /* Remove the fpage from the list first, otherwise we might get a
 ↪  circular list */
            remove_fpage_from_list(as, fp, mpu_first, mpu_next);

            fp->mpu_next = as->mpu_first;
            as->mpu_first = fp;

            /* Get first avalible MPU index */
            i = 0;
            while (sfp) {
                ++i;
                sfp = sfp->mpu_next;
            }

            /* Update MPU */
            mpu_setup_region(i++, fp);

            while (i < 8 && fp->mpu_next) {
                mpu_setup_region(i++, fp->mpu_next);
                fp = fp->mpu_next;
            }

            return 0;
        }

        fp = fp->as_next;
    }
    return 1;
}
```

Listing 30: `mpu_select_lru` function

4.9.7 Configuring the RISC-V PMP

Whenever a context switch occurs to a new thread, the PMP must change its regions to configure it for the address space of the new thread. The addresses of the old addresses must be completely cleared so that the new thread does not accidentally have access to some regions of the address space of the old thread. If an access fault occurs, but the kernel figures out that the violating address is on a flex page for the thread but not configured on the PMP, then the flex page list is updated, and the entire address space is reconfigured on the PMP, which has the effect of swapping out one of the flex pages on the PMP. In order to configure the regions, the relevant `pmpaddr` CSRs must be matched with the correct bits in the correct `pmpcfg` CSR. Functions for setting up PMP regions for flex pages can be viewed in listings 31, 32, 33.

```c
void (*w_pmpaddrarr[16])(uint32_t) = {
    w_pmpaddr0,
    w_pmpaddr1,
    ...
    w_pmpaddr14,
    w_pmpaddr15,
};
uint32_t (*r_pmpaddrarr[16])() = {
    r_pmpaddr0,
    r_pmpaddr1,
    ...
    r_pmpaddr14,
    r_pmpaddr15,
};

/* w_pmpaddri writes a pmp addr register based on the pmp entry (1-16). */
void w_pmpaddri(int pmp_entry, uint32_t data) {
    w_pmpaddrarr[pmp_entry](data);
}
/* r_pmpaddri reads a pmp addr register based on the pmp entry (1-16). */
uint32_t r_pmpaddri(int pmp_entry) {
    return r_pmpaddrarr[pmp_entry]();
}
```

Listing 31: Arrays of functions for reading and writing pmp registers dynamically.

4.10 Interrupt delegation

A hallmark of L4 microkernels is their ability to have critical functionality in user space code in a secure and isolated manner. A significant part of the machinery that makes this possible is the way that L4 microkernels delegate interrupts to

```c
/* w_pmpcfgi_region writes a pmp cfg register based on the pmp entry (1-16).
 * Its a bit complex since the cfg configuration is densly packed in
 * registers cfg0-cfg3 (in rv32). Four bytes in each cfg register correspond
 * to the configuration of the 16 pmp entries. */
void w_pmpcfgi_region(int pmp_entry, uint8_t new_data_region, uint32_t old_data)
↪  {
    // Get the shift for the correct byte in the cfg register
    int shift = (pmp_entry % 4) * 8;
    uint32_t mask = 0xFF << shift;
    uint32_t shifted = new_data_region << shift;
    // Clear out the old data in the region we want to operate on in the chosen
↪   cfg
    uint32_t old_masked = old_data & ~mask;
    uint32_t final = old_masked | shifted;

    if (pmp_entry < 4) {
        w_pmpcfg0(final);
    } else if (pmp_entry < 8) {
        w_pmpcfg1(final);
    } else if (pmp_entry < 12) {
        w_pmpcfg2(final);
    } else {
        w_pmpcfg3(final);
    }
}

uint32_t r_pmpcfgi(int pmp_entry) {
    if (pmp_entry < 4) {
        return r_pmpcfg0();
    } else if (pmp_entry < 8) {
        return r_pmpcfg1();
    } else if (pmp_entry < 12) {
        return r_pmpcfg2();
    } else {
        return r_pmpcfg3();
    }
}
```

Listing 32: Functions for setting up PMP regions

```c
/* mpu_setup_region sets up pmp for an fpage.
 * n is the number for a range corresponding to two pmp entries. For example:
 *     n = 0 means pmp entries 0 and 1
 *     n = 3 means pmp entries 6 and 7 */
void mpu_setup_region(int n, fpage_t *fp) {
    if (n > (CONFIG_MAX_MAPPED_THREAD_FPAGES - 1)) {
        dbg_printf(DL_MEMORY, "MEMORY: ignoring request to activate fpage num %d
↪   since the maximum number of fpages has been activated: %d\n", n,
↪   CONFIG_MAX_MAPPED_THREAD_FPAGES);
        return;
    }
    int pmp_entry_lower = n * 2;
    int pmp_entry_upper = pmp_entry_lower + 1;
    uint32_t old_cfg_data = r_pmpcfgi(pmp_entry_upper);

    if (fp) {
        w_pmpaddri(pmp_entry_lower, (FPAGE_BASE(fp) >> 2));
        w_pmpaddri(pmp_entry_upper, (FPAGE_END(fp) >> 2));

        /* Check if we need to write to cfg, perform write if necessary */
        int shift = (pmp_entry_upper % 4) * 8;
        uint32_t mask = 0xFF << shift;
        uint32_t masked = old_cfg_data & mask;
        uint32_t shifted = masked >> shift;
        if ((shifted & 0xF) != 0xF) {
            w_pmpcfgi_region(pmp_entry_upper, 0xF, old_cfg_data);
        }
    } else {
        /* Clear region */
        w_pmpcfgi_region(pmp_entry_upper, 0x0, old_cfg_data);
    }
}
```

Listing 33: The function that sets up a flex page on the PMP

user threads via IPC messages. A user thread can request the kernel to delegate a specific interrupt to the user thread, which results in the kernel sending a magic IPC message to the user thread whenever the interrupt occurs. In this way, user threads can dynamically be configured as interrupt handlers via IPC messages while the kernel keeps full control of the interrupt configuration.

F9 implements interrupt delegation in the L4 style. A special case of IPC is provided that user threads can utilize to request delegation of an interrupt. When an IPC message is sent to the special thread id `THREAD_IRQ_REQUEST`, F9 will attempt to delegate an interrupt to the caller thread. Information about interrupt delegation is kept in a table `struct user_irq user_irqs[]`, which consists of structs that can be viewed in listing 34. When a user thread requests an interrupt, the kernel finds the corresponding interrupt in the `user_irqs[]` table and sets the `thr` and `handler` fields accordingly. The function that performs this setup can be viewed in listing 35.

```c
struct user_irq {
    tcb_t *thr;
    int irq;
    uint16_t action;
    uint16_t priority;
    irq_handler_t handler;
    struct user_irq *next;
};
```

Listing 34: Interrupt delegation struct defenition

When the kernel detects that an interrupt occurs, it first clears the corresponding CPU interrupt [11]. Then it finds the interrupt in the `user_irqs` table and pushes this entry unto a queue of pending interrupts. It then creates a magic IPC message to the thread that has the interrupt delegated to it, containing the interrupt number and handler function. In order to facilitate that interrupts are handled quickly, the kernel also enables the use of a special scheduler function, which is highly prioritized. When the scheduler calls this function, it will pop an entry from the queue of pending interrupts and schedule the thread for that pending interrupt, allowing the user thread to handle the interrupt quickly.

The entire process of interrupt delegation, from requesting the delegation to converting the interrupt to an IPC message, is shown in figure 13.

[11] the user thread usually still needs to clear the peripheral interrupt, more on this in section 4.2.3

```c
void user_interrupt_config(tcb_t *from)
{
    ipc_msg_tag_t tag = { .raw = ipc_read_mr(from, 0) };
    if (tag.s.label != USER_INTERRUPT_LABEL)
        return;

    int irq = (uint16_t) ipc_read_mr(from, 1);
    l4_thread_t tid = (l4_thread_t) ipc_read_mr(from, 2);
    int action = (uint16_t) ipc_read_mr(from, 3);
    irq_handler_t handler = (irq_handler_t) ipc_read_mr(from, 4);
    int priority = (uint16_t) ipc_read_mr(from, 5);

    user_irq_disable(irq);

    if (!IS_VALID_IRQ_NUM(irq))
        return;

    struct user_irq *uirq = user_irq_fetch(irq);

    if (!uirq)
        return;

    /* update user irq config */
    if (tid != L4_NILTHREAD)
        uirq->thr = thread_by_globalid(tid);

    uirq->action = (uint16_t) action;

    if (handler)
        uirq->handler = handler;

    if (priority > 0)
        uirq->priority = (uint16_t)priority;
}
```

Listing 35: Function that configures interrupt delegation

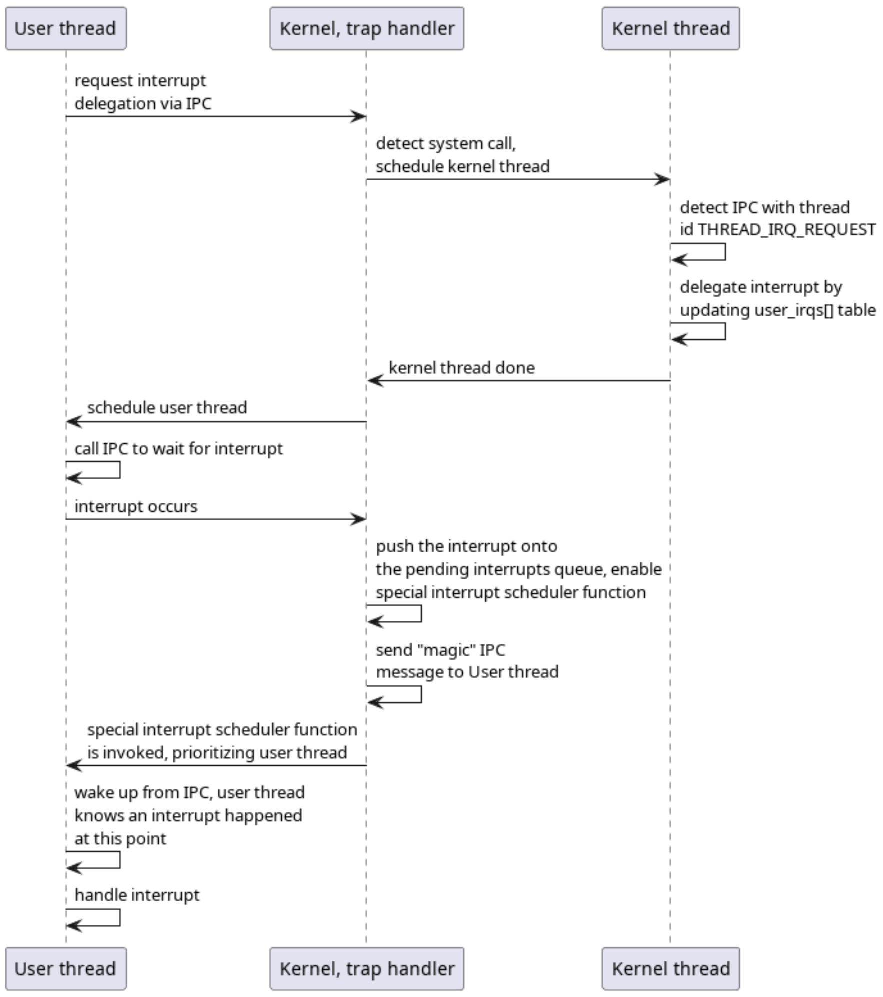

User thread
Kernel, trap handler
Kernel thread
request interrupt delegation via IPC
detect system call, schedule kernel thread
detect IPC with thread id THREAD_IRQ_REQUEST
delegate interrupt by updating user_irqs[] table
kernel thread done
schedule user thread
call IPC to wait for interrupt
interrupt occurs
push the interrupt onto the pending interrupts queue, enable special interrupt scheduler function
send "magic" IPC message to User thread
special interrupt scheduler function is invoked, prioritizing user thread
wake up from IPC, user thread knows an interrupt happened at this point
handle interrupt
User thread
Kernel, trap handler
Kernel thread

Figure 13: Interrupt delegation flow. The figure is slightly inaccurate when the user thread is waiting for interrupt, in actuallity the kernel would schedule another thread e.g. the idle thread, but this is omitted.

4.11 Event queue and kernel timer events

F9 manages an event queue of kernel timer events, based on the periodic timer interrupts generated from the system timer. This event queue can be used for various kinds of events in the kernel. Currently, it is used for IPC timeout events and an event that checks for outstanding IPC transfers, but it is flexible and can be used for other events as well.

4.11.1 Ticks

The kernel timer is based on kernel timer ticks, where one tick is one timer interrupt. Whenever a timer interrupt appears, the kernel trap handler manipulates counters that keep track of the current timer 36

```c
void ktimer_handler(void)
{
    ++ktimer_now;
    if (ktimer_enabled && ktimer_delta > 0) {
        ++ktimer_time;
        --ktimer_delta;
        if (ktimer_delta == 0) {
            ktimer_enabled = 0;
            ktimer_time = ktimer_delta = 0;
            softirq_schedule(KTE_SOFTIRQ);
        }
    }
}
```

Listing 36: `ktimer` tick function

The `ktimer_delta` variable keeps the number of ticks until the next kernel event and is decremented on each tick. When `ktimer_delta` reaches zero, a kernel timer event is ready to be fired. Instead of handling the event, the tick function schedules a special F9 software interrupt. As with all F9 software interrupts, it is eventually handled by the kernel thread, where the proper kernel event is fetched from the event queue and handled.

The event queue managed by the kernel timer is a queue of kernel timer events where the first event is the one closest in time. Each entry in the queue has a `delta` field, the number of ticks between the preceding event and this event, forming a queue of deltas. Whenever a new event is scheduled, or an event is handled, the deltas need to be recalculated.

```
typedef struct ktimer_event {
    struct ktimer_event *next;
    ktimer_event_handler_t handler;

    uint32_t delta;
    void *data;
} ktimer_event_t;
```

Listing 37: Kernel timer event struct definition

4.11.2 Scheduling timer events

When scheduling a new timer event, it must be inserted into the event queue correctly.

1. First, the current number of ticks (the `ktimer_now` variable) is added to the requested ticks to account for the number of ticks that have occurred towards the next kernel event. [12]
2. The event queue is traversed, and the deltas of each kernel event traversed are accumulated into a variable `etime`.
3. When a kernel event is encountered that makes `etime` have an accumulated value that is larger than the requested amount of ticks, the new kernel event is inserted before it in the event queue.
4. Lastly, the kernel event after the newly inserted event has its delta value recalculated.

Inserting a new kernel timer event into the event queue is visualized in 14.

4.11.3 Event handler functions

Each kernel timer event has a handler, a function pointer that will be called when the event should be handled. The signature of such a handler can be viewed in 38.

When an event is handled, the kernel event object (37) of the event that is being handled is passed through the `data` argument of the handler function (38).

[12]The ARM version subtracts this value instead, which is believed to be a potential bug. See section 7.4

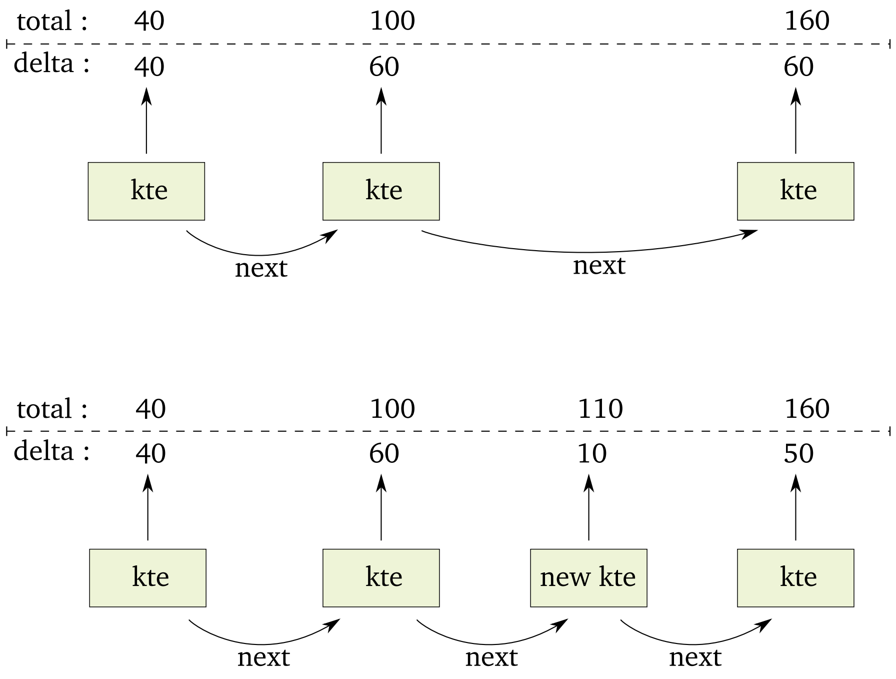


Figure 14: The process of inserting a kte (kernel timer event) into the event queue. In the top timeline, the event queue has three events with deltas 40, 60 and 60. The numbers above the lines are the accumulated deltas i.e. the total number of ticks until that event should be handled. In the bottom timeline, a new event is inserted with ticks = 110. It is inserted after the second existing event and gets a delta of 10 (40 + 60 - 110). The delta of the last event is recalculated to 50 to keep the total number of ticks until that event at 160.

```
uint32_t handler(void *data);
```

Listing 38: Kernel timer event handler signature

The handler function can choose to reschedule the event by returning a non-zero integer, where the integer returned will become the new amount of ticks to reschedule the event. A periodic event can be created by using a handler function that always returns the same integer. An example can be seen in 39.

```
uint32_t ipc_deliver(void *data)
{
    l4_thread_t receiver;
    tcb_t *from_thr = NULL, *to_thr = NULL;

    for (int i = 1; i < thread_count; ++i) {
        tcb_t *thr = thread_map[i];
        switch (thr->state) {
        case T_RECV_BLOCKED:
            if (thr->ipc_from != L4_NILTHREAD &&
                thr->ipc_from != L4_ANYTHREAD &&
                thr->ipc_from != TID_TO_GLOBALID(THREAD_INTERRUPT)) {
                from_thr = thread_by_globalid(thr->ipc_from);
                /* NOTE: Must check from_thr intend to send*/
                if (from_thr->state == T_SEND_BLOCKED &&
                    from_thr->utcb->intended_receiver == thr->t_globalid)
                    do_ipc(from_thr, thr);
            }
            break;
        case T_SEND_BLOCKED:
            receiver = thr->utcb->intended_receiver;
            if (receiver != L4_NILTHREAD &&
                receiver != L4_ANYTHREAD) {
                to_thr = thread_by_globalid(receiver);
                if (to_thr->state == T_RECV_BLOCKED)
                    do_ipc(thr, to_thr);
            }
            break;
        default:
            break;
        }
    }

    return 4096;
```

Listing 39: IPC timeout event handler function

4.12 Boot process

F9 also performs several required tasks steps when starting up.

- PMP regions are reset

- Kernel tables are initialized
- The different subsystems are initialized
- UART is set up for debugging
- Kernel thread, root thread and idle thread are created
- Interrupts, and the interrupt delegation system is set up
- Initial context switch to the kernel thread is performed, where another context switch to the kernel trap handler quickly follows

The ESP32-C3 only supports vectored interrupt mode. The alternative would be direct mode, where all traps always jump to a specified address. Vectored mode is not a problem to set up, but there is an issue regarding the PMP. When a trap occurs, F9 executes a single jump instruction in the trap vector while in user-mode, which means it needs a region on the PMP with correct permissions covering the trap vector. Therefore, F9 sets up a permanent region on the PMP that is dedicated to the trap vector and that omits write permissions. This is also locked in place by using the lock bit on the corresponding `pmpcfgi` register.

4.13 User space code

4.13.1 Starting a user thread

User space code is compiled along with the kernel binary, where the positions are defined in the linker script. All user space code begins with the root thread. It is the responsibility of the root thread to create and start the first user threads and set up and map all required memory regions to them. The root thread starts with all user memory in its address space so that it can perform these mappings.

A user thread is created when the root thread performs a `ThreadControl` system call. This system call allocates a TCB in the thread kernel table, sets up an empty address space (unless it should be a shared address space), and sets up a UTCB. At this point, the thread exists in the kernel, but the thread is in `inactive` state, and it does not have an initial value for the program counter, nor does it have a stack or a stack pointer. Before the thread is started, it needs an address space that covers its code, data and stack memory regions. The root thread can do this by performing a special case of IPC that maps the relevant memory regions from the address space of the root thread to the (still inactive) user thread. If this step is not done, or it is misconfigured, the user thread will probably crash with an unrecoverable access fault.

After the user thread is created and it has all required memory regions in its address space, the root thread can issue a special case of IPC to start the user thread, where the root thread sends a program counter and a stack region via the IPC message. The kernel thread detects this special case and performs the relevant setup. After this is done, the user thread will be scheduled and start execution at some point.

4.13.2 UART I/O from user threads

UART input from user threads is performed via a standard `printf` function that accepts a format string and optionally some variables. There are two ways of printing from user threads, either via UART or the kernel's logging system.

- `output via direct UART` Direct UART output is fairly simple. The user thread that wants to print something on UART needs to have the correct UART memory regions mapped in its address space, and then it should be able to manipulate the required registers in such a way that data should be transferred. UART input is a bit more complex. When data arrives through UART, an interrupt is generated through the ESP32-C3 UART controller. This means that in order for a user thread to use UART input, it needs to request for interrupt delegation from the kernel for that interrupt. When a user thread makes such a request, the F9 kernel will register the interrupt for the user thread. Then when a UART interrupt occurs, the kernel will intercept it, convert it to an IPC message, and send it to the correct user thread (as explained in section 4.10). Once the user thread receives a UART interrupt via an IPC message, it knows that input is available, so it can proceed to read data from the UART controller.
- `output via kernel logging`: A special IPC message can be sent in order for user threads to output data via the kernel logging facilities. The IPC message must be sent to a special `THREAD_LOG` thread, with two words containing descriptors for variable arguments as used by `printf` functions. The kernel will detect such an IPC message, extract the variable arguments, and log the message using the internal kernel logging system. A user space `printf` style function is provided that converts its arguments to the correct IPC message (see listing 40). When user threads are using the kernel logging system for output, it has the advantage of multiplexing one UART controller between multiple user threads in a controlled manner. However, it is substantially slower than using UART directly.

```c
int __USER_TEXT __user_log_printf(const char *format, ...)
{
    L4_Msg_t msg;
    va_list va;

    va_start(va, format);

    L4_MsgClear(&msg);
    L4_MsgAppendWord(&msg, (L4_Word_t)format);
    L4_MsgAppendWord(&msg, (L4_Word_t)&va);

    L4_MsgLoad(&msg);
    L4_Send((L4_ThreadId_t) {
        .raw = TID_TO_GLOBALID(THREAD_LOG)
    });

    va_end(va);

    return 0;
}
```

Listing 40: Printf function that uses the kernel logging system for output

4.13.3 User space timer

In addition to the system timer, another time was needed in order to measure the performance of various tasks. It is possible to use the system timer for this, but since the ESP32-C3 has additional timers available, those were used instead. Two such additional timers on the ESP32-C3 exist in the form of timer group 0 and timer group 1. They function in the same manner, differing only in the base address used to interface with them. More info can be found in section 11 in the ESP32-C3 TRM [9].

The timers use counters that can source their frequency from either the `APB_CLK` or `XTAL_CLK`. In order to keep things simple, the port configures the timer to use `XTAL_CLK`, since it has a fixed frequency of 40MHz, while the frequency of the `APB_CLK` is determined by the clock source of the CPU clock (see section 6.2.4.2 in the ESP32-C3 TRM [9]).

In addition to setting the clock source, a 16-bit prescaler is configured to divide the clock source by a value. The least possible value for the prescaler is 2 (if it is set to 1, the timer uses it as 2), which means a 20MHz frequency for the timer counter if the clock source is `XTAL_CLK`. The port sets the prescaler value to 10000.

These steps must be followed to use the timer:

- Configure the timer, i.e. set timer direction, prescaler, clock source etc.
- Reset the timer if required
- Start the timer
- Latch the timer value
- Read the latched timer value

Some convenience functions are provided to deal with the timer is shown in 41 42.

5 Hoppus

5.1 Overview

In order to test the microkernel port, some user space code was needed. What started out as a simple experiment for getting some user space code running quickly developed into a LISP interpreter experiment. As development progressed, having a LISP interpreter on the platform with F9 made more and more sense.

Hoppus is a dialect of LISP [22], designed to run on F9 RISC-V with a goal of having some interesting and functional user space code for F9 RISC-V. Since F9 RISC-V is made to run on embedded systems, Hoppus also has some optimization and design choices that make it more suitable for embedded systems. Hoppus aims to be small and simple. It is a LISP of type LISP-1 [11], meaning it has a single namespace for variables and functions, and it uses only dynamic scope (as opposed to lexical scope). Hoppus uses a tree-walk interpreter for evaluation which is a simple but slow approach. A hello world example in Hoppus can be viewed in listing 43.

The syntax of Hoppus is simple, as is common with LISP dialects. Hoppus code is constructed with an abbreviated form of S-expressions. An s-expression is defined with the use of the characters:

) (.

and the following two rules (from [22]):

1. Atomic symbols are S-expressions
2. If e_1 and e_2 are S-expresions, so is $(e_1 \cdot e_2)$

```c
__USER_TEXT void timer_init() {
    volatile uint32_t *timg_0_conf = REG(TIMER_GROUP_0_BASE + TIMG_T0CONFIG_REG);
    /* Make sure the timer is stopped */
    *timg_0_conf &= ~(1 << 31);
    /* Select clock source by setting or clearing TIMG_T0_USE_XTAL field */
    *timg_0_conf |= (1 << 9);
    /* Configure the 16-bit prescaler by setting TIMG_T0_DIVIDER */
    *timg_0_conf &= ~(0xffff << 13);
    *timg_0_conf |= (10000 << 13);
    *timg_0_conf |= (1 << 12);
    /* Configure the timer direction by setting or clearing TIMG_T0_INCREASE */
    *timg_0_conf |= (1 << 30);
    /* Set the timer's starting value by writing the starting value to
↪   TIMG_T0_LOAD_LO and
       TIMG_T0_LOAD_HI, then reloading it into the timer by writing any value to
↪   TIMG_T0LOAD_REG */
    volatile uint32_t *timg_t0loadlo = REG(TIMER_GROUP_0_BASE +
↪   TIMG_T0LOADLO_REG);
    volatile uint32_t *timg_t0loadhi = REG(TIMER_GROUP_0_BASE +
↪   TIMG_T0LOADHI_REG);
    *timg_t0loadlo = *timg_t0loadhi = 0;
    volatile uint32_t *timg_t0load = REG(TIMER_GROUP_0_BASE + TIMG_T0LOAD_REG);
    *timg_t0load = 1;
}

__USER_TEXT void timer_reset() {
    volatile uint32_t *timg_0_conf = REG(TIMER_GROUP_0_BASE + TIMG_T0CONFIG_REG);
    *timg_0_conf &= ~(1 << 31);
    volatile uint32_t *timg_t0load = REG(TIMER_GROUP_0_BASE + TIMG_T0LOAD_REG);
    *timg_t0load = 1;
}

__USER_TEXT uint64_t timer_counter_to_microseconds(int counter_value) {
    int clk_freq = 4000;
    int useconds_per_cycle = 1000000 / clk_freq;
    return counter_value * useconds_per_cycle;
}

__USER_TEXT uint64_t timer_get() {
    volatile uint32_t *timg_t0lo = REG(TIMER_GROUP_0_BASE + TIMG_T0LO_REG);
    volatile uint32_t *timg_t0hi = REG(TIMER_GROUP_0_BASE + TIMG_T0HI_REG);
    uint64_t res = 0;
    res = *timg_t0lo;
    res |= ((uint64_t) (*timg_t0hi)) << 32;
    return res;
}
```

Listing 41: Timer convenience functions

```c
extern volatile uint32_t *timg_0_update;
extern volatile uint32_t *timg_0_conf;
extern uint32_t timer_conf_en_bit;

#define TIMER_LATCH() (*timg_0_update = 1)
#define TIMER_START() (*timg_0_conf |= timer_conf_en_bit)
```

Listing 42: Timer convenience macros

```
(print "Hello, world!")
```

Listing 43: Hello world in Hoppus

5.2 Why LISP

LISP is a fascinating programming language that has been the subject of much research. It is a flexible language with many use-cases. Two properties of LISP are especially valuable for the project in this thesis: Its flexibility and simple syntax. The simple syntax of LISP meant that it was feasible to create a working implementation in a relatively short period of time. At the same time, the flexibility would provide a lot of power per effort when working on the implementation.

LISP was invented by John McCarthy and introduced to the world by his famous paper published in 1960 [22]. John McCarthy introduced a new formalism for defining recursive functions, as well as a type of expressions called S-expressions. He further showed how one could create a large amount of functionality using only a few primitive functions and went on to demonstrate this by showing how one can create an evaluator using just those primitives. McCarthy also proposed how these techniques could be used to implement a whole new "programming system": LISP

While LISP has many aspects that are worthy of note, perhaps the most defining characteristic is the attribute that everything is a list, both code and data. This attribute has many consequences, and it is a feature that synergizes with other features of LISP. For example, variable arguments, macros, quoting and quasiquoting are all features that play well with the fact that code and data are represented by lists.

The macro system in LISP deserves special attention. Many languages provide some form of macros, and they have some programming constructs to define code that defines code: metaprogramming. For example, in the C programming language, one can define macros by using the preprocessor. However, the language for defining macros in C is different from the language of C itself. In addition, the input to the macros is not in the same structure as the C programming language. In LISP, macros are written in the LISP syntax, the input to the macros are lists, and they generate LISP code (also lists). This gives the developer the ability to manipulate LISP code using LISP code, using all the constructs and facilities normally available in a LISP programming environment. This makes macros safer and more powerful in LISP than in most other programming languages, where they are usually notoriously hard to program correctly.

5.3 Goals of this implementation

The primary goal was to create a working but simple LISP interpreter running on F9 on the ESP32-C3. To have a suitably feature rich LISP interpreter, some secondary goals were defined as well:

1. There should be some integrations with F9 RISC-V, so that one can experiment with F9 constructs in LISP.
2. Performance is not a primary concern, but it should be good enough to be usable
3. It should have a simple but working garbage collector.
4. An interactive shell should be made so that a live demo would be possible.

5.4 Features

5.4.1 Macros

Hoppus supports macros, which are defined with the `defmacro` built-in. In a compiled LISP, macros are evaluated in the compile phase, and the resulting code is inserted. In an interpreted LISP, there is more freedom concerning when to evaluate macros. In Hoppus, macros are evaluated as they are encountered in the runtime, not before. Whenever a macro is encountered, it is immediately evaluated, and then the resulting code is evaluated. This is a straightforward approach, but it has the disadvantage that semantic errors are not detected until the macro is evaluated for the first time.

Hoppus also has some other features that complement macros:

- quoting to halt evaluation of a form

- quasiquoting and comma to conditionally halt evaluation of a form
- comma-at (`,@`) to evaluate a form within a quasiquote and splice the results

5.4.2 REPL

Hoppus features a REPL, which continuously reads user input, evaluates the input as Hoppus code, and then prints back the result. Variables, functions and macros can be defined, which are added to the REPL environment.

The parser and tokenizer control the REPL so that Hoppus can (somewhat) intelligently decide when to wait for more input and when the current expression is ready for evaluation. The parser will detect unmatched opening parenthesis and wait for input when required. This design makes the parser and tokenizer depend on the REPL, which is not strictly necessary but gives more control over the user experience of the REPL.

5.4.3 Standard library

A simple standard library is provided, where the implementation is baked into the source code as strings. When Hoppus starts up, it evaluates the expressions in the standard library and adds them to the environment before starting the REPL. The standard library provides convenience functions such as `cadr` and `first`. The `concat` standard library function (See listing 44) is noteworthy, it is designed to work on lists, but since strings are implemented as lists in Hoppus it works perfectly fine on strings as well.

```
(defun concat (seq1 seq2)
  (if seq1
      (cons (car seq1) (concat (cdr seq1) seq2))
      seq2))
```

Listing 44: concat standard library function in Hoppus

5.4.4 Other features

- `variable arguments`: variable arguments are implemented with the `&rest` keyword. This synergizes especially well with macros and the `comma-at` operator.

- `strings as lists`: There is no string type in Hoppus; instead, the printer function will detect when a list contains only characters and print it as a string. This has the advantage that all functions that can operate on lists can be used on strings as well.

5.5 Internals

5.5.1 Tokenizer

Hoppus has a standard tokenizer. It reads characters from UART line by line and splits those strings into tokens. The tokenizer needs some string handling functions in order to perform the tokenization. Many (but not all) of the functions created have been made with inspiration from the C standard library but does not necessarily follow the spec exactly:

- `strlen`
- `strcmp`
- `strcpy`
- `strtok`

These functions are unfortunately vulnerable to memory safety problems. For example, a buffer overflow could easily occur using the `strcpy` function. Implementing versions of these functions that accept a size parameter would help mitigate these issues.

5.5.2 Parser

The parser takes tokens as input from the tokenizer and outputs an Abstract Syntax Tree (AST). The AST is constructed by cons cells. A cons cell is a simple data structure common in LISP, shown in listing 45.

Cons cells are often used to form linked lists, using the `car` field as a data field and the `cdr` field as a next pointer. However, cons cells are flexible data structures, and the fields can be used for anything. The parser uses cons cells to construct ASTs. It uses both the `car` and the `cdr` field to point to other cons cells, but it never uses the `cdr` field for pure data; only the `car` field contains data when appropriate. A visualization of the output from the parser can be seen in figure 15.

```c
typedef struct expr_t expr;
struct expr_t {
    expr *car;  /* Data field if other type than cons cell */
    expr *cdr;  /* LSB designates if it is cons cell or anything else. */
};
```

Listing 45: Cons cell struct definition. LSB stands for Least Signifanct Bit. Cons cells in Hoppus deviate somewhat from cons cells in most other LISPs, see 5.5.6 for an explanation.

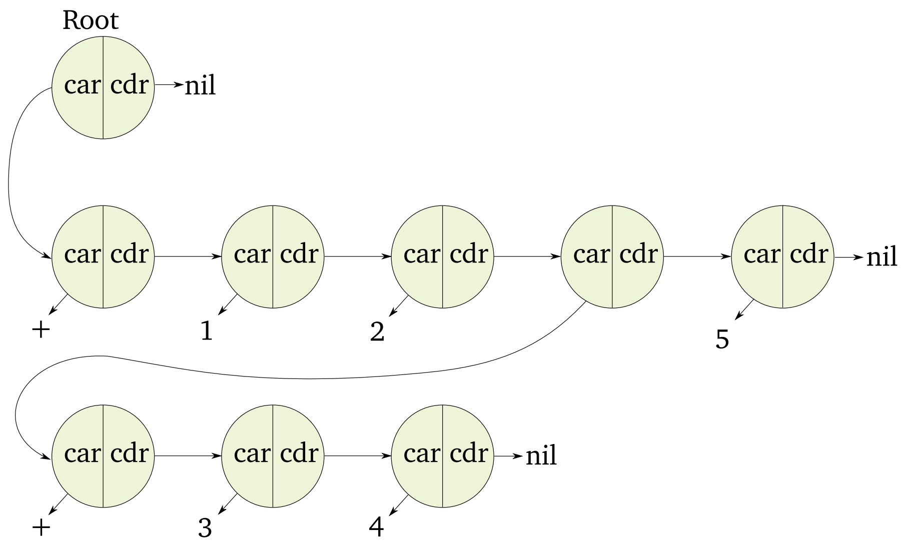


Figure 15: The AST that the parser generates when it has parsed the LISP code `(+ 1 2 (+ 3 4) 5)`

However, the reality is a bit more complex than what is presented in figure 15 since each cons cell does not have its own data in the car field. Instead, the car field points to another cons cell with its car field filled with the appropriate data. This is explained in more detail in section 5.5.6.

5.5.3 Evaluator

The purpose of the evaluator is to analyze ASTs (the result of the parser) and take action depending on the contents and shape of the AST. In many cases, the evaluator will collapse expressions into a single cons cell representing the resulting data. However, other times the evaluator might transform an AST into another tree, or it might perform side effects, which is any form of output independent of the resulting tree.

An example where the evaluator returns a single cons cell of data:

```
(+ 1 2)
```

the result would be a cons cell where the car field has the value 3.

An example where the evaluator returns a different AST:

```
(macroexpand (if (eq 1 1) "yes!" "no..."))
```

The result would be another tree representing the expanded macro of `if` (`if` being a macro that uses `cond` internally). The LISP code representing that tree would look like this:

```
(cond ((eq 1 1) "yes!") (true "no..."))
```

An example where the evaluator performs a side effect:

```
(print "I must not fear. Fear is the mind-killer.")
```

In this case, the argument to `print` will be printed through UART. The `print` built-in also returns its arguments, so the evaluator will both print the argument via UART, and return the argument as a cons cell with data pointing to the string.

5.5.4 Built-ins

Hoppus has built-in functions that represent the primitive functions. These functions are coded in C, and are created as symbols (see section 5.5.5) by setting the built-in field in the symbol struct to a function pointer to the correct C function. Usually, when Hoppus wants to evaluate a function, it looks for the expression field in the symbol struct. However, if the symbol is a built-in, it instead uses the built-in function pointer to invoke the corresponding C function. The built-ins provided in hoppus are building blocks that can be used to define more sophisticated functionality. Some of the most important built-ins:

name	description
define	defines a variable
defun	defines a function
defmacro	defines a macro
cond	conditional expression
quote	don't evaluate following form
quasiquote	don't evaluate following form, conditionally (see comma and comma-at)
comma	inside a quasiquote, evaluate the following form
comma-at	inside a quasiquote, evaluate the following form, splice the result
time	times the evaluation of a form
ipc	sends an IPC message to another F9 thread

An example of one of the simpler built-ins, `add` can be viewed in listing 46. If the code `(+ 1 2 3)` is being evaluated, the `arg` argument will be the first entry in a linked list containing the numbers `1 2 3`. If a built-in is marked as a special form, then the arguments to that function is not evaluated before sending the form to the built-in. An example of a built-in that requires this feature is the `cond` built-in which can be viewed in listing 47. Cond is a construct that exists in most lisp dialects and is roughly comparable to switch statements in other languages. A `cond` is a list of pairs of predicates and forms. Hoppus goes through each predicate and returns the result of the first form in which its predicate is true. The important part is the ability of Hoppus to not evaluate any forms with predicates that are false or that come after the first true one. In order to achieve this, `cond` needs to receive its arguments unevaluated so that it can only evaluate the required first form with a true predicate.

Another interesting built-in is the `time` built-in, which was created in order to time the evaluation of forms. The fact that Hoppus is a LISP helped with keeping the implementation simple. By marking the built-in as a special form, it gets its argument unevaluated, which is convenient. The `time` built-in can time the evaluation of the form by starting a user space-time, evaluate the form as usual,

```
__USER_TEXT int bi_add(expr *arg, expr **out) {
    int acc = 0;
    expr *curr = arg;
    for_each(curr) {
        if (type(car(curr)) != NUMBER) return TYPE_ERROR;
        acc += (int)dar(curr);
    }
    expr *_out = expr_new_val(NUMBER, (uintptr_t)acc);
    *out = _out;
    return 0;
}
```

Listing 46: Add built-in

```
__USER_TEXT int bi_cond(expr *arg, expr **out) {
    int ret_code;
    expr *curr = arg;
    for_each(curr) {
        if (!car(curr) || !cdr(car(curr)))
            return INVALID_FORM_ERROR;
        expr *pred = car(car(curr)), *form = car(cdr(car(curr)));
        expr *evald;
        if ((ret_code = eval(pred, &evald)) < 0) return ret_code;
        if (expr_is_true(evald)) {
            if ((ret_code = eval(form, out)) < 0) return ret_code;
            return 0;
        }
    }
    *out = NULL;
    return 0;
}
```

Listing 47: Cond built-in

and then latch the timer value when the evaluation is complete. The `time` built-in returns the value of the evaluation of the form and prints the timing result as a side effect. See listing 48 for the source code of `time`. For more info on the timers, see section 4.13.3. For example, timing the calculation of the tenth Fibonacci number can be done in this manner: `(time (fib 10))`, the return value of this form will be the return value of `(fib 10)`, while the time will be printed as a side effect.

```c
/** Time the evaluation of a form, return the result of evaluation and print the
↪   time.
    Does not support a time invocation inside another time invocation */
int bi_time(expr *arg, expr **out) {
    int ret_code;
    TIMER_START();
    ret_code = eval(car(arg), out);
    TIMER_LATCH();
    if (ret_code < 0) {
        timer_reset();
        hoppus_puts("evaluation failed of form to time\n");
        return ret_code;
    }
    int counter_val = timer_get();
    timer_reset();
    int us = timer_counter_to_microseconds(counter_val);
    hoppus_printf("counter: %d, time: %d us \n", counter_val, us);
    return 0;
}
```

Listing 48: Time built-in

5.5.5 Symbols

Symbols are defined via the `symbol` struct (see listing 49). Symbols are added to a global `symbols` table that Hoppus uses to look up a symbol given its name. Hoppus loops through the symbol list and compares each name with the symbol it looks for, returning the match. The `expr *e` field is the value of the symbol. In the case of a variable, this can be a Hoppus number, or in the case of a function, it contains the function forms and parameter names. The `built-in_fn_t *built-in_fn` serves both as a check for if the symbol is a built-in and a function pointer to the C function of the built-in. If the built-in field is not NULL, then the symbol is a built-in function. Finally, the `is_special_operator` field designates if this symbol is a special operator function. In Hoppus, a special operator is a function that does not evaluate its argument. All Hoppus functions defined with the `defun` built-in are not special operators, but many built-in functions are special operators, such as the `defmacro` built-in.

```
typedef struct symbol_t {
    char *name;
    symbol_type type;
    expr *e;
    built-in_fn_t *built-in_fn;
    int is_special_operator;
} symbol;
```

Listing 49: Struct representing a Hoppus symbol

5.5.6 Cons cells

Cons cells in Hoppus are similar to cons cells in other LISPs, and they are also used for general Hoppus expressions [13]. They also have some restrictions in the interest of reducing memory consumption. Cons cells in Hoppus consist of a `car` and a `cdr` field, where the `car` can hold any 32-bit data, and the `cdr` field is either a pointer or contains type information. Since all pointers are aligned to 4-bytes, we can use some of the lower bits for type information. In Hoppus, if a cons cell has the Least Significant Bit (LSB) of the `cdr` cell unset (equal to zero), then it is of type cons cell. If the LSB is set then it is of any other type, and bits 1 - 31 of the `cdr` field contain additional type information. This is illustrated in figure 16.

The advantage of this approach is that we can fit a cons cell in 64 bits, but there are some disadvantages. The main problem is that for a cons cell to hold data, the `cdr` field cannot be a pointer because it holds type information. So in order to create a list of cons cells, each `car` field needs to point to another cons cell containing the data, negating some of the savings in memory consumption.

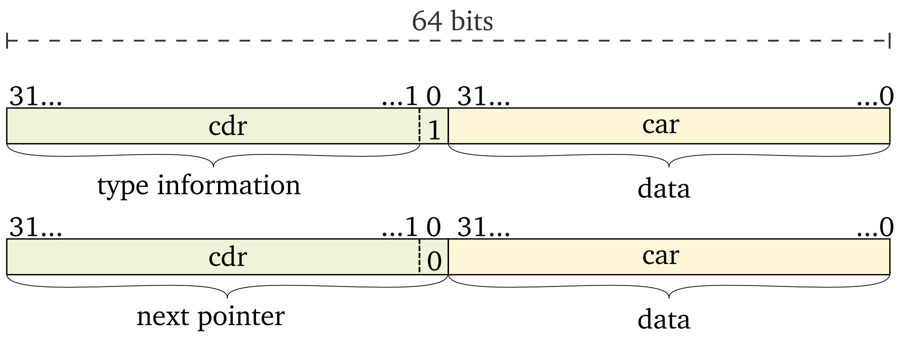


Figure 16: Memory layout of cons cells. If the LSB of cdr is unset, then the type is a cons cell, and the cdr field is the next pointer of a cons cell. If the LSB of cdr is set, then the rest of the bits contain additional type information.

[13] which is why they are named `expr` in the code, see listing 45

```
#define CDR_UNTAG(cdr) ((expr *)(((uint32_t) cdr) & 0xFFFFFFFE))
#define CDR_IS_CONS(cdr) (!((expr *)(((uint32_t) (cdr)) & 1)))
#define CDR_OTHER_TYPE(cdr) ((expr_type)(((uint32_t) (cdr)) >> 1))

expr *cdr(expr *e) {
    return (expr *)((uint32_t) e->cdr & 0xFFFFFFFE);
};
void set_cdr(expr *e, expr* new_cdr) {
    e->cdr = (expr *)((uint32_t) CDR_UNTAG(new_cdr) | !CDR_IS_CONS(e->cdr));
}
expr_type type(expr *e) {
    if (CDR_IS_CONS(e->cdr))
        return CONS;
    return CDR_OTHER_TYPE(e->cdr);
};
void set_type(expr *e, expr_type type) {
    if (type == CONS) {
        e->cdr = (expr *)((uint32_t)(e->cdr) & 0xFFFFFFFE);
    } else {
        e->cdr = (expr *)((((uint32_t) type) << 1) | 1);
    }
};
```

Listing 50: Functions for filtering and dealing with bits in cons cell fields

5.6 Memory allocation and garbage collection

Hoppus has its own memory allocator and garbage collector. The memory allocator is implemented as a malloc function, while the garbage collector is a mark-and-sweep garbage collector that is triggered when a malloc detects that it lacks the required space for an allocation.

5.6.1 Memory allocation

Since user space code for F9 RISC-V does not have access to the C standard library, Hoppus needs its own memory allocation implementation if it wants to use dynamic memory. An implementation of `malloc` was therefore created.

The memory allocator in Hoppus is written as a standard `malloc` C function, with the signature `void *gc_malloc(unsigned int size)`. It accepts a size parameter and outputs a pointer to the newly allocated memory area. It returns a NULL pointer if an error occurs, such as if there is not enough space for the requested size (after the garbage collector has attempted to free more space). By convention, we usually call the memory region that is handled by a dynamic memory allocator as the `heap`. The heap area for the memory allocator is a dedicated memory area defined in the linker script for F9, which is mapped into the Hoppus user thread by the root thread before Hoppus starts.

The provided malloc function is used throughout Hoppus for all dynamic memory allocation. This includes memory that is directly used by Hoppus in LISP land and internal objects used in C. The consequence is not only that the Hoppus programming language becomes more dynamic but also that the development of Hoppus becomes more dynamic since one does not have to worry about freeing used memory when programming Hoppus source code.

The malloc implementation works by keeping track of all allocated memory. Conceptually, as more memory requests appear, the memory allocator builds a list of all the used memory blocks. For each allocated region the memory allocator keeps track of the starting point of that memory region and the size of it.

The memory allocator is split into two parts, one for small objects of a defined size and another for all other allocations, which is given the name large allocations [14]. Each part of the memory allocator has its own algorithms for allocating and freeing memory, as well as its own part of the heap.

1. Allocating large objects

   The part of the memory allocator that deals with large objects work by embedding `header blocks` (shown in listing 51) ahead of the memory region that is allocated. The header blocks contain a next pointer and a size. The LSB of the next pointer also indicates whether the header block is marked by the garbage collector, which we are able to do since all memory regions here are aligned to the size of the header struct.

   The next pointer on the header blocks forms a linked list of allocated memory regions sorted by the base address in ascending order. When a new memory request appears through an invocation of malloc, the memory allocator will search through the list of allocated memory regions. It searches for space between the allocated regions for a free region that is large enough to accommodate the requested size. The first such region it finds is allocated, and the corresponding pointer is returned. In order to avoid searching through the entire list on each invocation of malloc, a pointer to the last allocated block is kept, so the allocator can start from that block the next time it is invoked. An alternative approach could be to search the linked list for the best-fit place to allocate the memory region. This would decrease fragmentation but increase the time to find a place to allocate the region. Since most of the allocations in Hoppus are on the other part of the heap (2), next-fit was deemed sufficient.

[14]a name that is not entirely appropriate since it is possible that they could include relatively small objects as well

```c
struct header {
    unsigned int size;
    header *next;
};
```

Listing 51: Embedded header block for large allocations

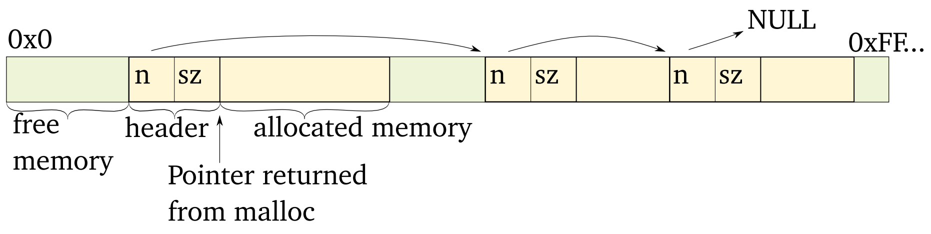


Figure 17: Layout of embedded header block. n: next pointer, sz: size.

2. Allocating small objects of a predefined size Using embedded header blocks works for all allocations, but it is wasteful in a LISP implementation. Usually, LISP needs to allocate lots of cons cells, which are small objects of the same size. In Hoppus, the header block is as large as the cons cell, so each allocation takes up double the amount of what is strictly required. It is for this reason that the memory allocator is split in two.

   For small allocations of a size equal to a cons cell, Hoppus uses a bitmap for memory allocation. The heap is split into equal-sized parts, and each bit in the bitmap indicates whether the corresponding part is allocated or not. An illustration of how this looks is shown in figure 18

   Using a bitmap for cons cells reduces the memory consumption to one bit per allocation. It also has the advantage that we can directly get a bitmap index from an address and vice-versa, which is beneficial both when allocating memory and when running the garbage collector. Since the vast majority of memory allocations are cons cells, it is a significant improvement to the performance of the garbage collector.

5.6.2 Garbage collection

A garbage collector is a type of automatic memory management that will scan for what allocated memory regions are being used and free up the regions that are not being used. Hoppus comes with a mark-and-sweep general-purpose garbage collector that should, in theory, work with any C program. It works by scanning some predefined memory regions for references to root objects and then recursively scanning the regions of those roots for other objects. In Hoppus, the garbage collector is given the stack, data and heap regions. For the stack region, the garbage

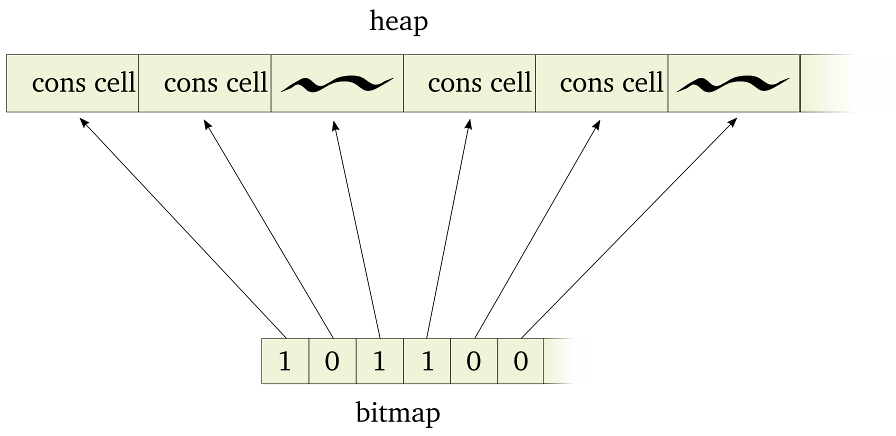


Figure 18: Example of bitmap allocation. Each bit in the bitmap points to a region of the heap. Squiggly lines indicate garbage data. The three first bits from the left contain the most notable examples. The first bit from the left is an allocated cons cell. The second bit is an unallocated memory region that contains an old cons cell. The third bit contains a memory region that has been allocated but not written to yet, so it still contains garbage data.

collector scans from the top until the stack pointer. The garbage collector is conservative, meaning it assumes that any data with the value of an address within the region of an allocated object is always a pointer to that object.

As Hoppus makes calls to malloc, it fills up the heap with allocated blocks. After a while, it should reach a point where it receives a request to allocate a portion of memory it does not have any size left for. At this time, the memory allocator will trigger the garbage collector to run its mark and sweep algorithm. After garbage collection is complete, the memory allocator will make another attempt at allocating the requested region. If the garbage collector managed to free up enough space, which it usually manages to do, the second attempt should succeed. If the garbage collector fails to free up the required amount of space, NULL will be returned from malloc.

The garbage collector is aware that the memory allocator is split into two parts, one for large objects and one for small ones. When the garbage collector is in the scanning phase, it uses different algorithms depending on which part of the heap it is currently scanning, but conceptually the procedure is the same. The garbage collector first needs to find all the root objects on the heap from which it should traverse. Once it has found all the roots, it checks each root to see if it contains a pointer to another object on the heap, then it checks those objects if they contain pointers to other objects and so on. For each such object, the garbage collector marks them. After it has traversed all the roots (end of the scanning phase), it iterates through all the objects on the heap and frees the unmarked ones.

A more detailed explanation of how this is implemented in Hoppus follows:

1. The garbage collector is provided with some regions to scan initially, which are the data and stack regions. Whenever it finds a pointer in those regions that point to an allocated object on the heap, that object is marked. If the object is on the large objects heap, then the linked list of embedded header blocks is traversed from the beginning of the list in order to find the block that contains the address, and then the LSB of the next pointer in the header is marked. If the object is on the small objects heap, then it is marked by setting the corresponding bit in an additional bitmap.

2. At this point, all the marked objects are root nodes. All objects on both heaps are iterated in order to scan for marked objects. For each marked object, the object's region is scanned, and each pointer in that region that points to another object is marked and followed recursively. When an object is followed (recursively scanned), and it is marked, then it is not followed further since it means it has already been followed.

3. After the scanning phase is done, the next thing to do is to free all unmarked objects. This is a much simpler procedure than scanning since all that needs to be done is to iterate the heap and free all unmarked objects. For large objects, they are freed by making the previous header point to the next header. For small objects, they are freed by clearing the corresponding bit in the allocation bitmap.

5.7 Integration to F9

5.7.1 Printing

Hoppus uses kernel logging functions for output (See section 4.13.2). This means that a context switch to the kernel thread happens on every print statement, which is much slower than using the UART controller directly, but it ensures synchronous access to the UART controller. However, there are no such mechanisms for UART input, so Hoppus uses the UART directly for input [15].

5.7.2 IPC built-in

An `ipc` built-in is provided to send F9 RISC-V IPC messages. The syntax for the `ipc` built-in is as follows `(ipc to_tid timeout msg1 msg2 ...)`. `to_tid` is the

[15]While Hoppus uses the interrupt delegation mechanisms of F9 RISC-V to get notified when UART data is available, the data is still read directly from the UART controller

id of the thread to send the IPC message to, `timout` is the IPC timeout, and the rest is a variable number of arguments that will be sent through MRs. The built-in will update the tag of the IPC message depending on how many message arguments there are. The built-in can be used to send IPC messages with timeouts and 0-16 untyped arguments to any other thread. This can be used to send messages to other user threads. However, it can also conceivably be used to request some of the special functionality that can be performed via IPC in F9, such as logging or requesting interrupt delegation. The built-in only supports sending of IPC messages but could be extended to support receiving messages as well. The implementation of the built-in can be viewed in listing 52.

An implementation of a printer user thread is provided to test the `ipc` built-in. The first thing the printer thread does is to print its own id, so that the user of Hoppus can use that id when invoking `ipc`. The printer thread then waits for IPC messages from the Hoppus thread in a while loop and prints the contents of the IPC messages it receives.

```c
/**
   Built-In for sending an F9 IPC message to another thread.
   example: (ipc to_thread_id timeout arg1 arg2 ...)
 */
__USER_TEXT int bi_ipc(expr *arg, expr **out) {
    int ret_code;

    L4_ThreadId_t to_tid = {.raw = dar(arg)};

    int timeout = dar(cdr(arg));
    expr *curr = cdr(cdr(arg));
    int num_msgs = list_length(curr);

    L4_Word_t *msgs = my_malloc(num_msgs * sizeof(L4_Word_t));
    int msg_i = 0;

    for_each(curr) {
        msgs[msg_i++] = dar(curr);
    }

    L4_Msg_t msg;
    L4_MsgClear(&msg);
    L4_MsgPut(&msg, 0, num_msgs, msgs, 0, NULL);

    L4_MsgLoad(&msg);
    L4_Ipc(to_tid, L4_nilthread, timeout, (L4_ThreadId_t *) 0);

    return 0;
}
```

Listing 52: IPC built-in in Hoppus, for sending F9 IPC messages to other threads.

6 Measurements

6.1 F9 IPC

In order to measure the performance of IPC, a pingpong program was created inspired by the approach used in Liedtke's paper on IPC performance [18]. The program consists of two user threads sending 200,000 IPC messages (as opposed to 20,000 in Liedtke's paper) to each other. The ping thread starts by sending an IPC message to the pong thread, and the pong thread waits for the message and replies. The average execution time is calculated by dividing the total execution time by 200,000 [16].

When performing IPC in the benchmark, the API inspired by the eXperimental reference manual is used. This has some overhead since IPC messages are loaded into memory before they are loaded into MRs. However, it should provide a more realistic measurement since most user space programs should use that API [17].

The CPU runs at 160MHz, and the code is compiled with `-Og -gdwarf-2`. The results can be viewed in table 2.

words	total time	time per IPC	bits per second
4	5.782 s	28.91 μs	4.43 Mbps
8	6.067 s	30.34 μs	8.44 Mbps
12	6.440 s	32.20 μs	11.93 Mbps
16	6.880 s	34.38 μs	14.89 Mbps

Table 2: Results from pingpong benchmark, measuring IPC performance. Words are 32 bits.

7 Discussion

7.1 Achieved objectives

The result of the work in this thesis is a functional but experimental port of F9 to RISC-V and the ESP32-C3. The port is able to provide process isolation by dynamically configuring flex pages on the RISC-V PMP for the running thread. The port is missing some features compared to the ARM version, but the memory protection subsystem is improved by utilizing TOR-mode addressing in the RISC-V PMP. While the ARM version needs to split up flex pages into smaller ones to

[16]The code used for the benchmark can be viewed in the git repository for the F9 port [28] in commit `de61b4d`, in the `ping_pong.c` file.

[17]The user manual for the eXperimental API [17] also mentions this issue in section 4.3.5.

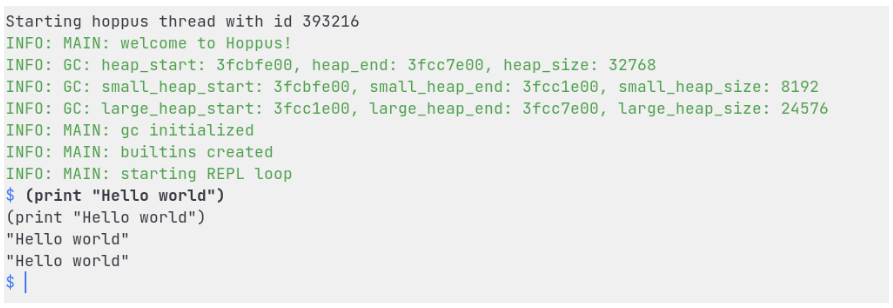
adhere to its restrictions on MPU regions, the port can configure any memory region on the PMP without splitting. This results in higher performance and less error-prone code in the kernel.

However, there are also some potential advantages for using ARM. Of special note is the `pendsv` exception, which is a low-priority exception made for performing context switches. In addition, the ARM version optimizes operations on kernel tables by utilizing ARM bit-banding. ARM also has instructions that are capable of loading and storing multiple values and registers at the same time, resulting in higher performance and more concise code when entering and leaving the kernel trap handler. RISC-V is a more minimalistic ISA, lacking such mechanisms. This means that more code is needed to achieve the same result as the ARM version, potentially with a performance penalty. On the other hand, there are good arguments for keeping instructions simple and the instruction set small, so this might be an advantage for RISC-V considering the system as a whole.

The memory protection mechanisms in the port help to make the system more secure in several ways. It guards against memory access to random addresses, not within the threads address space. It can detect stack overflows by making the end of the stack inaccessible. If flex page permissions are implemented (see section 7.3), it can guard against user code being modified by setting it non-writeable, and it can guard against jumps to data regions by setting those regions non-executable.

The LISP implementation, Hoppus, serves as a starting point for future research on the ESP32-C3. It is a working LISP implementation with a REPL that can be used for experimentation. By using the kernel logging system to output data, and accessing many different memory regions, it serves a test for running semi-complex user space code on top of F9, with frequent context switches into the kernel and back. The garbage collector in Hoppus serves as a starting point for interesting integrations with memory management in F9.

The Hoppus REPL running on F9 on the ESP32-C3 can be seen in figure 19.



```
Starting hoppus thread with id 393216
INFO: MAIN: welcome to Hoppus!
INFO: GC: heap_start: 3fcbfe00, heap_end: 3fcc7e00, heap_size: 32768
INFO: GC: small_heap_start: 3fcbfe00, small_heap_end: 3fcc1e00, small_heap_size: 8192
INFO: GC: large_heap_start: 3fcc1e00, large_heap_end: 3fcc7e00, large_heap_size: 24576
INFO: MAIN: gc initialized
INFO: MAIN: builtins created
INFO: MAIN: starting REPL loop
$ (print "Hello world")
(print "Hello world")
"Hello world"
"Hello world"
$ |
```

Figure 19: Hoppus running on F9, on the ESP32-C3.

7.2 Comparison

The main advantage of the RISC-V port is that flex pages do not need to be split up by utilizing RISC-V PMP TOR mode addressing (See section 4.9.3).

The ideal situation would be for the port to have feature-parity with the ARM version, but this was not possible because of the complexity involved with some of the features. Therefore, the port is a stripped-down version of the original.

The most notable features that are changed or removed:

- `simplified build system`: There is a comprehensive build system in the ARM version that can compile for several platforms with configurable options. The configuration is done with a TUI (Text-based User Interface) provided via `kconfig`, which is a selection-based configuration system initially developed for the Linux kernel. This build system is convenient and well-built but also quite complex. To reduce complexity, the build system was removed, and a single Makefile was used instead.
- `init hooks`: The ARM version of F9 has a hook-based system for running code at initialization time, with different priority levels. This was deemed unnecessary in the port and removed to reduce complexity. All initialization is done in the entry function instead.
- `tickless timer`: The ARM version has support for a tickless kernel timer to reduce power consumption. This would be interesting to investigate on RISC-V and the ESP32-C3, but it was removed in the interest of time.
- `preemptible kernel thread`: The kernel thread is preemptible in the ARM version. An attempt was made to make the kernel thread preemptible, but after a certain amount of bugs appeared, it was deemed to take too long to fix. In order to increase stability, the kernel thread has interrupts disabled. The kernel thread should ideally be preemptible, so this should be fixed, which requires disabling interrupts only at required points in the kernel thread.
- `permissions on fpages`: See section 7.3
- `kbrobes`: Kprobes in F9 is an in-kernel dynamic instrumentation mechanism inspired by the Linux kernel. It is implemented using the ARMv7 Debug Architecture and so was removed in the interest of time since it seems to be very platform-dependent.
- `in-kernel debugger`: The ARM version supports a mechanism for dumping state information from a running kernel. The port was developed exclusively with GDB (GNU Project Debugger), so it was deemed a non-critical feature while developing the port.

- `user runtime`: The ARM version provides a framework for defining, creating, configuring and starting user threads. A macro is provided for defining user threads, which accepts a variable number of flex pages that should be mapped into that user threads address space. The root thread is able to iterate all the defined user threads and partitions its memory regions for all the required UTCBs and stack regions of the user threads. The port uses a more primitive approach, defining separate memory regions for each user thread in the linker script and defining separate regions for the UTCBs and stacks of each user thread. This primitive approach works for a small number of threads in a system with low complexity, but a framework that is similar to the ARM version would be highly beneficial in the long run.

- `kernel table bit-banding`: The ARM version provides the option of using bit-banding with kernel tables, which was not possible to do in the RISC-V port.

7.3 Known issues

There are a number of issues in the port that should be fixed; here is a list of the major ones that are known. Being an experimental port, F9 RISC-V probably has more issues that have not been discovered yet.

- `access faults`: When an unrecoverable access fault happens for a thread, meaning the violating address is not in the thread's address space at all, the kernel panics and stops execution. A much better approach would be to stop the thread that caused the access fault or perhaps take on the concept of a pager thread that could potentially attempt to map in a memory region covering the violating address. This is an important issue to fix since one of the major advantages of microkernels is to recover from errors of system functionality in user space.

- `flex page permissions` Flex pages always have all permissions, meaning that they can be read, written and executed. This is unfortunate since there are multiple use cases for configuring permissions in certain regions. Implementing permissions on flex pages should be simple; the logic around flex page permissions is mostly unchanged from the original ARM version, so most of the remaining work should be to map flex page permissions to RISC-V PMP permissions when configuring the region on the PMP.

- `preemptible kernel thread`: To improve the stability of the port, interrupts are disabled in the kernel thread. This prevents some bugs, but it negates much of the purpose of using a kernel thread at all. One of the advantages of moving kernel functionality to a preemptible kernel thread is that interrupts can be serviced with low latency even while running in the privileged kernel thread. Disabling interrupts in the kernel thread negates this advantage.

7.4 Possible issues/bugs in F9 ARM

Some issues encountered when developing the port seem to be present in the ARM version F9 as well. The port provides solutions to some of these issues. It could be that the issues were introduced in the port via some side-effect that is difficult to debug, so more research is needed to verify that the bugs are indeed present in the ARM version.

```
...
        if (addr_in_fpage(addr, fp, 0)) {
            fpage_t *sfp = as->mpu_stack_first;

            /* Remove the fpage from the list first, otherwise we might get a
 ↪  circular list */
            remove_fpage_from_list(as, fp, mpu_first, mpu_next);

            fp->mpu_next = as->mpu_first;
            as->mpu_first = fp;
...
```

Listing 53: Fpage select function `mpu_select_lru()`

```
static void ktimer_enable(uint32_t delta)
{
    if (!ktimer_enabled) {
        ktimer_delta = delta;
        ktimer_time = 0;
        ktimer_enabled = 1;
    }
}
```

Listing 54: Kernel timer enable function

- `circular flex page list`: There is a chance that one of the flex page lists can become circular, causing an infinite loop. The cause of the issue is the creation of a duplicate entry in the list when the entry should only have been moved instead. The solution in the port is to ensure the duplicate entry is removed, as shown in 53.

- `kernel timer event scheduling`: When scheduling a kernel timer event, the tick value is decremented by the value of `ktimer_time`. This seems to cause the wrong number of ticks to be scheduled on the event queue. A possible solution is to increment the tick value instead. Doing this in the port fixes insertion into the event queue
- `kernel timer early event`: If an event is inserted at the start of the kernel timer event queue, the kernel counters are not updated accordingly, and the event will be skipped. The reason for this is because of the `ktimer_enable` function, which supposedly should "reset" the timer, but it doesn't do a reset if the kernel timer subsystem is already enabled. A fix for this is to provide a new function `ktimer_reset` that properly resets the kernel timer counters.
- `scheduler starvation`: There is a chance that some threads may never be scheduled under certain conditions. When no IPC or interrupt occurs, scheduling is done by traversing the thread tree from the root thread and scheduling the first runnable thread. This means that if two user threads are running in infinite while loops, only the thread closest to the root thread in the thread tree will be scheduled.

7.5 Other interesting problems encountered

Some problems encountered during the development of the port were especially intriguing and are listed here

7.5.1 PMP and literal strings

A surprising problem appeared when working on printing strings through UART; literal strings caused PMP access fault exceptions. The issue here is that literal strings are by default put in the section named `.rodata`, which is kernel space and should never be in any user thread's address space. Some macros were created to circumvent this issue; see listing 55. These macros automate the procedure of declaring a string in the `.user_data` section and then printing that string. Without these macros, all strings would have to be declared somewhere and then utilized in the corresponding invocation of `printf`.

7.5.2 Struct arguments in user space

`struct arguments`: When passing structs by value in arguments to functions in user space, an access fault occurs. The access fault happens because the program counter jumps to the `memcpy` implementation in kernel space, which should never

```
#define __USER_DATA         __attribute__ ((section(".user_data")))

#define user_printf(str, ...) { \
    static __USER_DATA char s[] = str; \
    printf(s, __VA_ARGS__); \
}
#define user_puts(str) { \
    static __USER_DATA char s[] = str; \
    printf(s); \
}
```

Listing 55: Macros for printing data through UART from user space. These macros are needed since literal strings are by default put in the `.rodata` section.

exist in the address space of any user thread. This jump probably happens because GCC uses `memcpy` to copy structs when they are passed by value in functions.

7.6 Discussion of measurements

Measurements of IPC performance was done with a pingpong program with the results presented in section 6. The benchmark followed the example in Liedtke's paper on IPC performance [18]. Different hardware was used for the measurements, but the results should still be interesting to compare. In Liedtke's paper, an Intel 486 DX-50 was used for the measurements, running at 50MHz, with a 256 KB external cache. The measurements of the F9 port were done with the ESP32-C3 running at max speed (160MHz). The ESP32-C3 is a 32-bit single-core processor with a 16 KB cache.

An interesting thing to note is that the time of the measurements increase roughly in proportion to the number of words sent. This is not what should be expected since the 8 first words in an IPC transfer are sent via hardware registers, and the 8 next are sent via the memory on the UTCB. This means that there probably are other bottlenecks slowing down IPC transfer to such a degree that this performance optimization has little effect.

The results from Liedkte's paper (see figure 20) are significantly better than the results obtained from the port of F9 2, even though the Intel 486 DX-50 runs at less than half of the frequency of the ESP32-C3. There might be several reasons for this difference:

- `L3 optimizations`: The version of L3 used in the paper is highly optimized, which was the purpose of the paper.

- `F9 port optimizations` The port of F9 is in an experimental stage with a focus on exploration and memory protection rather than performance.
- `L4 reference manual API`: When performing IPC in the benchmark, the API inspired by the eXperimental reference manual [31] is used. This has some overhead since IPC messages are loaded into memory before they are loaded into MRs.

7.7 Future work

Achieving feature-parity with the ARM version, or at least implementing the most important missing features, should be a goal for the future (See section 7.2 for a list of missing features). In addition, three are a number of other interesting opportunities to work on:

- `uart driver in user thread`: While logging in the kernel works fine for multiplexing access to the UART controller, an even better approach would be to implement a user thread that is dedicated to printing and receiving UART data for other threads. UART output would be similar to how it is done in the kernel and simple to implement. Input should also be possible to implement but would be more complicated.
- `thread scheduling` The scheduling algorithm for user thread has the consequence of always prioritizing one thread if no IPC calls or interrupt delegation is performed. Therefore, a fairer scheduling algorithm should be used.
- `ELF loader` While including user space in the kernel is a simple approach that works fine for small code examples, in a real-world setting a more flexible approach should be developed. An interesting alternative could be an ELF file loader. This would enable user space code to be compiled separately, which has many advantages. User space code could also be compiled as position-independent code. The root thread could then become much more dynamic in its management of user threads. If the root thread had an ELF loader available with position-independent user space code, it could search for available memory regions in its address space and find the most appropriate spots for new threads to run. It could free up memory of finished threads to allow more threads to run as well. It could even conceivably perform swapping to some secondary storage if that is available. A prerequisite for these ideas is a file system and secondary storage, which is not currently available.

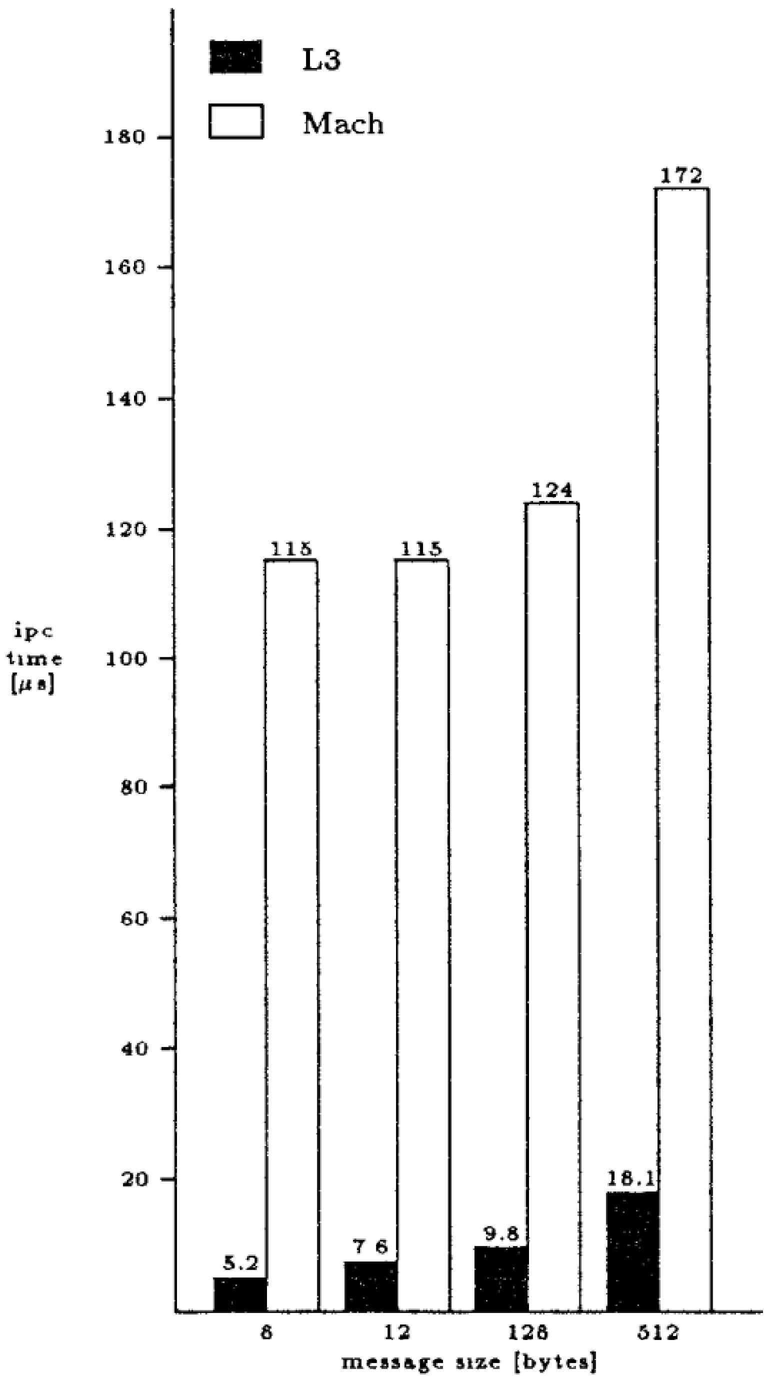


Figure 20: Results of measurements from Liedte's paper on IPC performance (Source [18])

- `pager threads`: The concept of a pager thread exists in L4 as a thread that intercepts page faults as IPC messages and handles them in some way. The F9 port currently does not use this concept at all, but there could be a place for them regarding PMP access faults. For example, instead of intercepting page faults, an F9 pager could intercept PMP access faults. One possible way of handling an access fault could be for the pager to search its address space for a flex page that covers the address that caused the access fault, and it could then map that flex page into the running thread's address space.

There are also some interesting opportunities between Hoppus and F9.

- `thread support`: A `thread` built-in could be implemented, which would be a special operator (arguments are not evaluated), and could start a new Hoppus thread which only evaluates the form given to the invocation of `thread` then exits. This would be a simple and effective way of bringing multithreading to Hoppus. The main issue that must be dealt with is to make the garbage collector thread-safe, which could be done with IPC synchronization.
- `garbage collector`: The garbage collector in Hoppus gets a fixed region of memory from the linker script as its heap. When that region of memory is full, and the garbage collector is not able to free up the required amount of space, Hoppus crashes. What would be interesting here is if, instead of crashing, the garbage collector could request additional memory from the root thread or some other specialized thread. This requires that the garbage collector can function with a heap that is not contiguous but would also make it more flexible.

With regards to optimizations, there is an interesting opportunity in regards to the RISC-V ISA. In F9, IPC data is sent through MRs, where the eight first words are sent through hardware registers and the eight next words are sent via memory on the UTCB. One of the benefits of using RISC-V is the availability of more general-purpose registers, 31 in total. An optimization that would be interesting to explore would be to utilize more of these registers for IPC transfer.

8 Summary

The goal of the research in this thesis was to explore memory protection on RISC-V based embedded devices. This was done by creating an experimental port of F9 from ARM to RISC-V and utilizing the RISC-V PMP to implement memory protection. The exploration and creation of the port showed that RISC-V devices with a PMP have great potential to implement isolation between threads and can even be advantageous relative to other ISA, which can have restrictions on the regions of memory on the memory protection unit. The port of F9 exists as a platform for further research and experimentation in this regard. The implementation of LISP that was also created should help facilitate research and experimentation in several areas, especially in regards to integrations between the memory management in F9 and the memory management in Hoppus.

While the port is functional, it is still in a highly experimental state and will need more work in order to be usable in a real-world setting. Fixing at least the most important missing features from the ARM version stands out as something that would be beneficial. In addition, the full potential of Hoppus and F9 is not realized as more integrations between the two systems are needed.

Porting F9 to RISC-V with the target platform being ESP32-C3 should also help to facilitate research on the ESP32-C3 and microkernels in general. Since F9 is unique in the way it targets embedded devices without an MMU, this could be an interesting prospect in the future.

References

[1] Qemu, a generic and open source machine emulator and virtualizer. https://www.qemu.org.

[2] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. 1986.

[3] W. Andrew, K. Asanović, SiFive Inc., and RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, December 2019.

[4] W. Andrew, K. Asanović, SiFive Inc., and RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, June 2019.

[5] ARM. *ARMv7-M Architecture Reference Manual*, february 2021.

[6] P. J. Denning. Virtual memory. 1970.

[7] Espressif Systems. Esp-idf, development framework for espressif socs. https://github.com/espressif/esp-idf.

[8] Espressif Systems. *ESP32-C3 datasheet v1.2*. https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.

[9] Espressif Systems. *ESP32-C3 Technical Reference Manual 2022 pre-release version 0.5*, 2022.

[10] Espressif Systems. *ESP32 datasheet v3.9*, 2022. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[11] R. P. Gabriel and K. M. Pitman. Endpaper: Technical issues of separation in function cells and value cells. June 1988.

[12] R. Garret. The remote agent experiment debugging code from 60 million miles away, February 2012. https://flownet.com/ron/RAX2.pdf.

[13] G. Heiser. The sel4 microkernel an introduction, 2020.

[14] J. Huang. F9 microkernel. https://github.com/f9micro/f9-kernel.

[15] R. Kaiser and S. Wagner. Evolution of the pikeos microkernel. 2007.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. 2009.

[17] I. Kuz. *L4 User Manual API Version X.2*, Sepember 2004.

[18] J. Liedtke. Improving ipc by kernel design. dec 1993.

[19] J. Liedtke. On u-kernel construction. 1995.

[20] J. Liedtke. Toward real microkernels. sep 1996.

[21] J. Liedtke and H. Wenske. Lazy process switching. 2001.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. april 1960.

[23] D. Patterson and A. Waterman. *The RISC-V Reader: An Open Architecture Atlas*. 2017.

[24] D. A. Patterson and K. Asanovic. Instruction sets should be free: The case for risc-v. aug 2014.

[25] RISC-V authors. Clint, advanced core local interruptor specification. https://github.com/riscv/riscv-aclint/blob/main/riscv-aclint.adoc.

[26] RISC-V authors. Plic, platform-level interrupt controller specification. https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc.

[27] P. Seibel. *Introduction: Why Lisp?*, pages 1–7. Apress, Berkeley, CA, 2005.

[28] R. Sevaldson. F9 risc-v. https://github.com/rubensseva/f9-riscv.

[29] R. Sevaldson. hoppus. https://github.com/rubensseva/hoppus.

[30] R. Sevaldson. Porting f9 to risc-v. December 2021. Made during TDT4501 Specialization Project at NTNU.

[31] System Architecture Group, Dept. of Computer Science, Karlsruhe Institute of Technology (L4Ka Team). *L4 eXperimental Kernel Reference Manual, version X.2, revision 7*, October 2011.

A Setting up F9 RISC-V

A.1 gcc & gdb

A RISC-V gcc toolchain is needed, capable of compiling with

- `arch=rv32imc`
- `abi=ilp32`

Note that it might be required to compile gcc from source:

```
$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git .
$ ./configure --prefix=<install-dir> --with-arch=rv32imc --with-abi=ilp32
$ make
```

A.2 openocd

An openocd installation is required to connect to the board. ESP-IDF framework [7] comes with one that is working with the ESP32-C3.

Since RTOS support is not required it can be removed; Navigate to the espressif installation folder, and find the file named `esp32c3.cfg` (which should exist under the openocd folder). Then comment out/delete these lines from `esp32c3.cfg`:

```
if { $_RTOS == "none" } {
    target create $_TARGETNAME esp32c3 -chain-position $_TAPNAME
} else {
    target create $_TARGETNAME esp32c3 -chain-position $_TAPNAME -rtos $_RTOS
}
```

A.3 Running F9 RISC-V

1. Connect the ESP32-C3 via USB
2. Start openocd (See listing 56)
3. Start a serial terminal with a baud rate of 115200 (See listing 57)
4. Start gdb from the F9 RISC-V root folder (in order for gdb to read the `.gdbinit` file)
5. Openocd should automatically connect to the connected ESP32-C3 board, if not, reset the ESP32-C3 board and run the `rst` command to reconnect and load the kernel again.
6. If everything worked, the serial terminal should now wait for input for Hoppus
7. Check the `.gdbinit` file for how to manually load the kernel, and to find more convenience functions.

```
$ cd ~/esp/esp-idf && . ./export.sh && sudo openocd -f board/esp32c3-built-in.cfg
```

Listing 56: Bash oneliner to start openocd, installed via esp-idf. `sudo` might not be required.

```
$ picocom -b 115200 /dev/ttyUSB0
```

Listing 57: Bash oneliner to start picocom.

Ruben S. Sevaldson

Memory protection for embedded RISC-V systems