

Bachelor Thesis
Herbstsemester 2018

Inhaltsübersicht

Muen on ARM

Version: 1.00, Datum: 21. Dezember 2018

Betreuer:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rügsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, Student
BSc Informatik
HSR Rapperswil

Dokumentationsübersicht

- A. Eigenständigkeitserklärung (deutsch)
- B. Persönlicher Bericht (deutsch)
- C. Aufgabenstellung (englisch)
- D. Management Summary (englisch)
- E. Bachelor Thesis (englisch)
- F. Glossar und Abkürzungsverzeichnis (englisch)
- G. Projektorganisation inklusive Zeiterfassung und Protokolle (deutsch)
- H. Studienarbeit (englisch)
- I. USB Stick mit sämtlichen Daten der während der Bachelor Thesis erstellten Dokumente, dem gesamten Code Repository sowie der referenzierten Literatur (sofern in pdf Form frei erhältlich)

Eigenständigkeitserklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe,
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Tischinas Ruschein, 21. Dezember 2018

Unterschrift David Loosli

Bachelor Thesis
Herbstsemester 2018

Persönlicher Bericht

Muen on ARM

Version: 1.00, Datum: 21. Dezember 2018

Betreuer:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rügsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, Student
BSc Informatik
HSR Rapperswil

Ausgangslage

Das Ziel der Bachelor Arbeit war die Implementation eines minimalen Separationskernels für die ARMv8-A Architektur basierend auf den Erkenntnissen der Studienarbeit. Als Zielplattform wurde in einem separaten Evaluationsprozess das NXP LS1012A FRDM Board festgelegt.

Projektorganisation

Den Erfahrungen bezüglich des administrativen Aufwandes aus der Studienarbeit entsprechend und nach Rücksprache mit sämtlichen Beteiligten entschied ich mich dafür, die Anzahl der verwendeten Projektadministrationstools von deren vier auf zwei zu reduzieren. Neben dem auf einer von der Hochschule für Technik Rapperswil (HSR) zur Verfügung gestellten virtuellen Maschine installierten Software Entwicklungstool Jira von Atlassian konnte über den Server der beiden Betreuer zusätzlich ein Git Repository verwendet werden. Sowohl das Aufsetzen der virtuellen Maschine inklusive der Installation der Jira Software als auch die Konfiguration des Git Repositories verliefen ohne Probleme. Von beiden Instanzen wurden zusätzlich lokal weitere BackUps erstellt.

Die Betreuung der Arbeit wurde wiederum von Adrian-Ken Rüeggsegger und von Reto Bürki übernommen. Mit dieser war ich auch während der gesamten Bachelor Arbeit äusserst zufrieden. So konnte ich dank einem hervorragend präsentierten Code Walk Through durch den Muen SK Quellcode und der jederzeit vorhandenen Unterstützung bei Schwierigkeiten auch fachlich enorm von diesem Projekt profitieren. Die vollumfänglich protokollierten Besprechungen empfand ich wiederum sowohl auf fachlicher als auch persönlicher Ebene als sehr lehr- und hilfreich. Ich möchte mich deshalb an dieser Stelle nochmals für die Unterstützung bedanken!

Ebenfalls noch ganz herzlich bedanken möchte ich mich bei Prof. Dr. Andreas Steffen. Ohne seine fachliche und organisatorische Unterstützung wäre ein solches, mehrere im Rahmen des Bachelor Studiengangs an der Hochschule für Technik Rapperswil (HSR) zu absolvierende Arbeiten umfassendes Projekt schlichtweg nicht möglich gewesen.

Bachelor Thesis

Im Gegensatz zum evaluativen Ansatz der Studienarbeit war die Bachelor Arbeit aufgrund der klaren Vorgaben und Erkenntnisse aus der vorangegangenen Studie sehr viel zielgerichteter. Grundsätzlich folgte die Implementation der einzelnen Komponenten dem Aufbau der Studienarbeit.

Obwohl mir seitens der Betreuer erneut viele Freiheiten bezüglich der Herangehensweise an die Problemstellung gewährt wurden, waren die Rahmenbedingungen und Ziele einiges enger gesteckt und wurden auch eingehender überprüft als in der Studienarbeit. Allerdings fehlte es deshalb nicht an Herausforderungen bei der Implementation der einzelnen Komponenten - insbesondere das Generieren der Adresstabellen, bei denen schon die kleinsten Änderungen zu trotz vorhandenem Hardware Debugger kaum sinnvoll zu untersuchenden Fehlern führen können, stellte mich vor so einige Probleme.

Mit dem Ergebnis meiner Bachelor Thesis bin ich persönlich sehr zufrieden. Besonders stolz bin ich auf das Setup der Entwicklungsumgebung mit dem in die GPS IDE von AdaCore integrierten Debug Möglichkeiten über den J-Link Debug Adapter, die angepasste Version der OpenOCD Anwendung und den GDB Debugger Tools. Der einzige kleine Wermutstropfen findet sich darin, dass die Virtualisierungserweiterung des Generic Interrupt Controllers und damit einhergehenden der ARM Generic Timer aus zeitlichen Gründen nicht mehr implementiert werden konnten. Dies insbesondere deshalb, da ich davon überzeugt bin, dass auch diese Komponenten noch hätten realisiert werden können, wäre ich nicht 1.5 Wochen gesundheitlich angeschlagen gewesen. Trotzdem bin ich davon überzeugt, dass sich die rund 580 investierten Stunden in diese Bachelor Arbeit mehr als nur gelohnt haben.

Persönliche Erkenntnisse

Im Verlaufe des gesamten Projektes konnte ich erneut auf fachlicher Ebene in den verschiedensten Bereiche enorme Fortschritte erzielen und mich bezüglich der administrativen Arbeiten vollumfänglich auf meine bisherigen Erfahrungen aus den anwaltlichen Tätigkeiten verlassen. Ich bin immer noch davon überzeugt, dass mir eine solche Arbeit im Gegensatz zu den üblichen Modulprüfungen ein Vielfaches an konzeptionellem, breit anwendbarem Wissen vermitteln kann.

Während der Bachelor Arbeit konnte ich von den Erkenntnissen der Studienarbeit sehr profitieren. So konnte ich mich dank der Einschränkung der administrativen Arbeiten zugunsten inhaltlicher Aspekte verstärkt mit der eigentlichen Implementation auseinandersetzen. Ich bin der Ansicht, dass sich dies sehr positiv auf die Qualität meines Quellcodes ausgewirkt hat.

Die aus der Studienarbeit gewonnene Erkenntnis bezüglich der Problematik der Einzelarbeit hat sich auch während der Bachelor Thesis bestätigt. So wäre ich in den verschiedensten Situationen froh gewesen, einen Diskussions- und Entscheidungspartner mit einem ähnlichen Wissensstand zu haben. Andererseits könnte sich in Zukunft der Entscheid, eine Einzelarbeit einzureichen, doch noch als lohnenswert herausstellen.

Mitunter eine der wichtigsten Erkenntnisse der Studienarbeit waren die negativen Folgen meiner mit einem zu grossen Aufwand und Perfektionismus verbundenen Herangehensweise. Ich habe diese Erkenntnis während des gesamten Projektes zu berücksichtigen versucht und einen etwas pragmatischeren Ansatz gewählt. Allerdings zeigte sich beispielsweise bei der Implementation des Generic Interrupt Controller, dass mir dies nicht immer gelang und dass ich daran weiter arbeiten muss.

Zum Schluss möchte ich noch anmerken, dass ich das grosse Interesse an meiner Bachelor Arbeit seitens von AdaCore und den Klienten der codelabs GmbH als äusserst motivierend empfand. Ich bin sehr glücklich darüber, dass das Projekt nicht in einem Aktenschränk in Vergessenheit gerät, und freue mich bereits heute auf den Antritt der offerierten Stelle zur Weiterentwicklung des Muen SK on ARM Projektes!

Bachelor Thesis
autumn semester 2018

Definition of Task

Muen on ARM

version: 0.00, date: December 21, 2018

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rüegsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Introduction

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

Based on the findings of the former student research study „Muen On ARM - an Evaluation“ written by the author of this Bachelor Thesis, the objective of this study is to develop a minimal Separation Kernel prototype for the ARMv8-A architecture based on the Muen SK and leveraging the AArch64 Virtualization Extensions introduced with the latest ARM architecture. The target hardware platform for this Bachelor Thesis is the NXP LS1012A Freedom Evaluation Board with an ARMv8 Cortex-A53 CPU and the programming language is Ada/SPARK 2014.

Objectives

(i) Prototypical implementation of main Separation Kernel building blocks

- System initialization
- Exception & interrupt handling
- Definition and switching of AArch64 subject state
- Subject preemption mechanism
- Serial debug driver

(ii) Documentation

(iii) Optional:

- Scheduling of multiple subjects
- AArch64 pagetable generation tool

Bachelor Thesis
autumn semester 2018

Management Summary

Muen on ARM

version: 1.00, date: December 21, 2018

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rügsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Introduction

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

Background

Based on the findings of the former Student Research Study „Muen On ARM - an Evaluation“ written by the author of this bachelor thesis, the objective of this study was to develop a minimal separation kernel prototype for the ARMv8-A architecture based on the Muen SK, written in Ada/SPARK and leveraging the AArch64 Virtualization Extensions introduced with the latest ARM architecture. The target hardware platform for this bachelor thesis is the NXP LS1012A FRDM Board with an ARMv8-A Cortex-A53 single core CPU.

Results

Using a Segger J-Link hardware debug probe device, the on-chip debugger software OpenOCD and the AdaCore toolchain including their integrated development environment, essential parts of a separation kernel have been implemented in Ada in the course of the project. With this basic separation kernel prototype and its two differently configured subjects, it could be demonstrated that all requirements with respect to the porting of the Muen SK to the ARMv8-A architecture can be met applying the already during the Student Research Study examined ARMv8-A architecture design principles.

Prospects

Due to the enormous economic interest in porting the Muen SK to the ARMv8-A architecture and the current success of the Muen SK project, the author of this study is going to continue to develop the Muen on ARM project full time starting in March 2018.

Bachelor Thesis
autumn semester 2018

Bachelor Thesis

Muen on ARM

version: 1.0, date: December 21, 2018

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rügsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Change History

date	version	change	author
Dec 10, 2018	0.1	prepared template, setup basic version, glossary, bibliography and introduction	David Loosli
Dec 13, 2018	0.2	bibliography, glossary, abstract and introduction up to scope	David Loosli
Dec 14, 2018	0.3	introduction up to first part theoretical background	David Loosli
Dec 15, 2018	0.3	second part theoretical background	David Loosli
Dec 16, 2018	0.4	first part practical part	David Loosli
Dec 17, 2018	0.4	second part practical background	David Loosli
Dec 18, 2018	0.4	third part practical part	David Loosli
Dec 19, 2018	0.5	last part practical part up to epilogue	David Loosli
Dec 20, 2018	0.6	change history	David Loosli
Dec 21, 2018	1.0	final check before hand in	David Loosli

Abstract

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and separates security critical functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

This bachelor thesis implements the main building blocks of a separation kernel for the ARMv8-A architecture, leveraging in particular the recently introduced AArch64 Virtualization Extensions. This practical study builds on the findings of the Student Research Study „Muen on ARM - an Evaluation“ also written by the author of this paper that investigated the theoretical and practical aspects of porting the Muen SK to the ARMv8-A architecture. The target hardware platform chosen for this study is the NXP LS1012A FRDM Board.

Contents

Change History	2
Abstract	3
1 Introduction	6
1.1 Structure of the Thesis	7
1.2 Scope	7
1.3 Related Documents	8
1.4 Literature	8
2 Theoretical Background	10
2.1 Muen Separation Kernel	10
2.1.1 Functional Principle	10
2.1.2 Components	11
2.2 ARM Overview	12
2.2.1 ARMv8-A Architecture	13
2.2.2 ARM Cortex-A53 Implementation	15
2.2.3 ARM Peripheral Components	16
2.3 NXP LS1012A Evaluation Board	16
2.3.1 Overview	17
2.3.2 Documentation	19
2.3.3 Board Setup	19
3 Practical Part	22
3.1 Development Environment	22
3.1.1 Toolchain	22
3.1.2 Integrated Development Environment	23
3.1.3 Debugger Setup	24
3.1.4 Deployment	29
3.2 Software Architecture	30
3.2.1 MuenSK Projects	31
3.2.2 Code Structure	32
3.2.3 Code Style	33
3.2.4 License	34
3.3 Implementation Details	35
3.3.1 Startup Code	35
3.3.2 Registers	36
3.3.3 Subjects	38
3.3.4 Subject Context Switch	40

3.3.5	Memory Management	43
3.3.6	Exception Handling	46
4	Conclusion	47
4.1	Status of Development	47
4.2	Integration of the Muen SK	47
4.3	Further Development	48
5	Epilogue	49
	Appendix	51
A	Project Task Description	51
B	List of Related Documents	53
C	U-Boot Environment Setup	54
D	GDB Initialisation Script	55
E	Codelabs Contributors Agreement	57
	Bibliography	59
	List of Figures	61
	List of Tables	62

1 Introduction

The tremendous progression in the last years in the world of information technology not only led to an enormous increase of mobile devices and into network integrated components, but also to a raised awareness and alertness with respect to network security among companies and people. In particular, the latest political developments not only in Europe ¹ but throughout the world show an increased need for security and protection of one's own personality throughout the internet.

Reto Buerki and Adrian-Ken Rueegsegger recognised this need for high-assurance security very early on and designed the Muen Separation Kernel (SK) as their Master Thesis at the University of Applied Sciences Rapperswil (HSR) in the year 2013². The Muen SK basically leverages three principles. The first principle is a mathematically provable secure approach to control the access to sensitive data and was invented with the theory of the Separation Kernel published by John Rushby in a paper presented at the 8th ACM Symposium on Operating System Principles in December 1981³. In order to be able to implement such a separation kernel, the programming language SPARK as a second component was chosen, which allows to formally prove the correctness of code. Thanks to developments in the field of processor architecture, the Muen SK was able to be built on the Intel Virtualization Extension as a third component, thereby improving performance and reducing the size of the code.

In the last years, ARM Limited has expanded its primarily on low energy embedded devices focused business strategy to general purpose central processing unit (CPU) architectures with great success. Many small devices, especially mobile devices, currently use an ARM CPU or an ARM based system on chip (SoC). With the latest ARM architecture, the so called ARMv8 architecture ⁴, ARM Limited finally introduced a Virtualization Extension composed of several components that opened the market for small devices also to the Muen SK.

After the successful exploration of the ARMv8-A architecture during the Student Research Study and the confirmation that a porting of the Muen SK should be possible, this bachelor thesis as the final paper of the Bachelor of Science in Computer Science program at the University of Applied Sciences Rapperswil (HSR) now realizes the main components of a separation kernel for the ARMv8-A architecture in software. Even though the target platform for this project is the the NXP LS1012A Freedom evaluation board, the implementation attempts - whenever possible - to keep in mind the diversity of the existing ARM based system on chip (SoC).

¹ Exemplarily, the Datenschutz Grundverordnung (DSGVO) established by the European European Union (EU) and put into force since May 2018 can be mentioned with its effects for the entire continent, [https://www.nzz.ch/wirtschaft/folgen-der-neuen-datenschutz-grundverordnung-eu... \(dt.\)](https://www.nzz.ch/wirtschaft/folgen-der-neuen-datenschutz-grundverordnung-eu... (dt.)), December 21, 2018.

²[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013.

³[22] Rushby. "Design and Verification of Secure Systems". 1981.

⁴https://en.wikipedia.org/wiki/ARM_architecture#ARMv8-A, December 21, 2018

1.1 Structure of the Thesis

This paper is divided into three main parts preceded by this introduction and followed by an epilogue containing a summary of the project. In the first main part (chapter 2) an overview of the theoretical background related with the topic of porting the Muen SK to the ARMv8-A architecture is given. This includes an overview of the ARM architecture, a description of the chosen hardware platform as well as a short analysis of the Muen SK. The next chapter 3 describes the development and deployment process as well as the software architecture and design decisions that had to be taken during the construction phase of this project. The explanations of this passages of the document correspond - with regard to the content - to the software architecture document (SAD) as an important part of the software development process. The third main part of this thesis (chapter 4) is dedicated to an analysis and discussion of the current implementation and suggests some approaches for further development and the integration of the ARMv8-A project into the Muen SK.

1.2 Scope

As already mentioned, the main objective of this study is to develop a minimal separation kernel prototype for the ARMv8-A architecture based on the Muen SK and leveraging the AArch64 Virtualization Extensions introduced with the latest ARM architecture. The official task description document can be found in the appendix A. In order to be able to achieve the corresponding objectives in the time available, the following topics - even though important with respect to the porting of the Muen SK - have to be considered out of scope:

- (i) *ARM TrustZone*: The ARM TrustZone provides a hardware mechanism to isolate trusted software. With this separation, an ARMv8-A processor supports a secure (*Secure World*) and a non-secure (*Normal World*) state and allows an operating system to run in parallel with a so called trusted operating system ⁵. Since the ARMv8-A boot process requires the code in the TrustZone to be executed before entering the Normal World over the secure monitor and since the Muen SK as a hypervisor has to be executed at exception level 2 in the non-secure state, the ARM TrustZone is considered to be out of scope for this study. In future, it must be ensured that the ARM TrustZone does not execute any code or at least does not have access to the Normal World.
- (ii) *System Initialization*: At the beginning of the project it was determined that the firmware and bootloader should in particular configure and initialize the DDR RAM component and hand over execution to the hypervisor code at exception level 2. However, in the course of the project it soon became apparent that the hypervisor has to make additional configurations with regard to the physical interrupt handling ⁶ and the Generic Interrupt Controller (GIC) ⁷ at exception level 3.

⁵Further details can be found in in chapter 3, section 3.2 in the Student Research Study [4] and on the ARM homepage under <https://developer.arm.com/technologies/trustzone>, December 21, 2018.

⁶cf. chapter 3, section 3.3.1

⁷cf. chapter 3, section 3.3.6

Therefore, it is now assumed that the according evaluation board **(a)** configures and initializes the existing random access memory (RAM) components, **(b)** establishes an identity mapping as well as configures and enables the Memory Management Unit (MMU) if necessary for the use of the RAM, and **(c)** hands over the execution to the separation kernel code at exception level 3 (i.e. secure monitor mode). Since this requirements are derived from the currently used NXP LS1012A FRDM evaluation board, this assumptions will have to be extended and adjusted or as well taken to a higher level of abstraction with the integration of further target hardware platforms.

- (iii) *ARMv8-A AArch32*: As described in the following chapter 2.2, the ARMv8-A architecture retains full compatibility with the ARMv7-A AArch32 execution state. Therefore, there actually exist two sets of registers related with either the AArch64 or the AArch32 execution state. Since the extension of the kernel to the execution of 32-bit applications does not provide any additional conceptual insights, this implementation of a separation kernel only considers the ARMv8-A AArch64 execution state.
- (iv) *Multicore Environment*: This study only considers aspects of a single core system and therefore sets all eventually existing cores apart from the main processor into a waiting state.

1.3 Related Documents

As this bachelor thesis is based on the feasibility study of porting the Muen SK to the ARMv8-A architecture, the Student Research Study written by the author of this paper is declared as an **integral** part of this document. In the following it is therefore assumed that the reader is familiar with the basics of hardware related concepts and software development explained in the student research project. A list of all the related documents can be found in the appendix B.

1.4 Literature

First of all, it has to be mentioned that a detailed list of referenced literature can be found in the bibliography at the end of this document (cf. [Bibliography](#)). Due to the structure and the requirements of this bachelor thesis, the principal literature used for writing this document as well as the implementation source code can be divided into four different main topics:

- (i) *ARM*: The theoretical part is mainly based on the Student Research Study⁸ but also incorporates the book *Professional Embedded ARM Development* by James A Langbridge⁹. For the practical part with respect to the ARM related literature, both the generic architecture documents with the

⁸[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017.

⁹[3] Langbridge. *Professional Embedded ARM Development*. 2014.

ARMv8-A Programmer's Guide¹⁰ and the ARMv8-A Architecture Reference Manual¹¹ as well as the documents specifying the implementation of the processor, i.e. the ARM Cortex-A53 MP Core Technical Reference Manual¹², specific application notes on in-depth topics such as the Virtualization Extension¹³ and Generic Interrupt Controller related manuals^{14,15}, have to be considered.

- (ii) *Evaluation Board*: For all by the ARM documentation as „implementation defined“ declared issues and for all driver related topics, the NXP LS1012A processor specific (i.e. the Data Sheet¹⁶, the Processor Reference Manual¹⁷ and the Security Reference Manual¹⁸) and the NXP LS1012A FRDM board specific documents (i.e. Getting Started Guide¹⁹ and the Board Reference Manual²⁰) have to be consulted.
- (iii) *Muen SK*: Although the book Programming in Ada 2012²¹ could not be worked through completely due to the time constraints of this bachelor thesis, it has nevertheless to be mentioned at this point as a reference book together with the online available Ada Reference Manual²². The implementation also follows the SPARK principles explained in the Muen documentation²³ and the Muen SK design and implementation principles²⁴.
- (iv) *Development and Tools*: As the development environment and the corresponding tools form the basis of the entire project, a separate section is dedicated to them. The principal literature used for this topic includes the OpenOCD User's²⁵ and Developer's²⁶ Guides, the Segger J-Link documentation²⁷, the ARM Cortex JTAG interface documentation²⁸ and the NXP PBL Configuration Application Note²⁹.

¹⁰[7] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015.

¹¹[6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2018.

¹²[8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2018.

¹³[5] n.a. *AArch64 Virtualization*. 2017.

¹⁴[9] n.a. *ARM Generic Interrupt Controller, Architecture Specification*. 2013.

¹⁵[11] n.a. *CoreLink GIC-400 Generic Interrupt, Technical Reference Manual*. 2012.

¹⁶[16] n.a. *QorIQ LS1012A Data Sheet*. 2018.

¹⁷[18] n.a. *QorIQ LS1012A Reference Manual*. 2018.

¹⁸[19] n.a. *QorIQ LS1012A Security (SEC) Reference Manual*. 2017.

¹⁹[17] n.a. *QorIQ LS1012A Getting Started Guide*. 2016.

²⁰[15] n.a. *QorIQ LS1012A Board Reference Manual*. 2016.

²¹[1] Barnes. *Programming in Ada2012*. 2018.

²²Official Ada 2012 Reference Manual, <https://developer.arm.com/technologies/trustzone>, December 21, 2018

²³[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, SPARK, chapter 2, section 2.1, page 3 f.

²⁴[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Design, chapter 3, page 19 ff., and Implementation, chapter 4, page 31 ff.

²⁵[21] Oliver et al. *Open On-Chip Debugger: OpenOCD User's Guide*. 2017.

²⁶[20] Oliver et al. *Open On-Chip Debugger: OpenOCD Developers's Guide*. 2017.

²⁷[13] n.a. *J-Link / J-Trace User Guide*. 2018.

²⁸[12] n.a. *CoreSight Components, Technical Reference Manual*. 2009.

²⁹[14] n.a. *QorIQ LS1012A Application Note, PBL Configuration using QCVS*. 2016.

2 Theoretical Background

This chapter is intended to give a brief overview of the topics already developed in the Student Research Study by summarising the individual chapters of the documentation and highlighting the most important topics once again.

2.1 Muen Separation Kernel

As already mentioned, the design and implementation of the Muen SK is premised on three basic concepts. First of all, it is based on the Separation Kernel principle introduced by John Rushby¹ that basically adapts the principles of a distributed system with respect to physical isolation, communication and shared resources to a single processor and can be verified with a Proof of Separability. Secondly, the theory of the Separation Kernel principle requires that an actual implementation of this principle has to use a programming language that is amenable to formal verification. Therefore, the Muen SK is written in SPARK, a formally analysable subset of the programming language Ada². Finally, to achieve the requirement of a sufficiently small code base, the Muen SK relies on the hardware virtualization support of the Intel x86/64 architecture³.

2.1.1 Functional Principle

As defined by the separation kernel principle, the system or security policy is at the heart of the Muen SK implementation. This policy describes basically a system-wide, static allocation of resources in a way such that the guest systems are strictly isolated from each other and their communication is exclusively governed by the Muen SK type I hypervisor according to a policy. In practice, this means that supported by the Muen SK Tools an IRQ routing specification for the system's I/O APIC⁴, a vector routing specification to determine the destination subject of interrupt vectors⁵, a memory map defining the kernel stack, the page tables and the per-CPU storage memory addresses, the static scheduling plans for all CPU cores including a barrier as synchronization mechanism to avoid any interprocessor drift in the context of scheduling plans and hence to eliminate timing side channels⁶ as well as the subject specifications are generated. Finally, all object binaries created by the build process (i.e. the Muen SK kernel and all subjects) are packed into a bootable OS image⁷.

¹[22] Rushby. "Design and Verification of Secure Systems". 1981.

²cf. <https://www.adacore.com/sparkpro>, December 21, 2018

³[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Intel Virtualization Technology (VT), chapter 2, section 2.3.1, page 12 ff.

⁴[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.3, page 18 ff.

⁵[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, External Interrupts, chapter 4, section 4.4.6, page 50 f.

⁶[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.2, page 17 f.

⁷[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Image Packaging, chapter 4, section 4.5.3, page 57 f.

2.1.2 Components

The Muen SK makes use of several (hardware) components and principles to guarantee the separation of the guest systems. In this section, the most important components especially with respect to this bachelor thesis are mentioned again. Detailed descriptions of these components and principles can be found in the Student Research Study in chapter 2.

To be able to implement a small code based type I hypervisor, the target processor architecture needs to provide hardware assisted virtualisation support. The Muen SK relies on the Intel Virtualization Technology (VT) that introduces a new hypervisor execution level with an additional protection ring and simplifies the switching between a hypervisor running in VMX root operation and a guest subject executing in VMX non-root⁸.

With respect to memory and storage, it is crucial to remember the different technologies within their hierarchy as well as the varying aspects of accessing this components⁹. While the secondary (i.e. disk storage) and tertiary (i.e. input storage) are treated as a pure I/O device by the Muen SK¹⁰, all memory resources of a system running the Muen SK are allocated statically and are explicitly specified in the so called system policy¹¹. This, for example, not only implies that there is no such mechanism implemented for loading missing page contents from a storage device after a page fault or page miss, as most of the common operating system kernels would do, but also leads to the concept of generating the page tables statically during the compilation and build phase of the Muen SK. A detailed overview of the functionality, use and resulting requirements with respect to memory, caches, memory management, advanced memory virtualization and even memory in multicore environments can be found in chapter 2, section 2.2, of the Student Research Study.

The third aspect that has to be concerned in the context of separating guest systems is the interrupt handling. The Muen SK uses the Intel's Advanced Programmable Interrupt Controller (APIC) that is composed of two components - the Local APIC as a part of every physical CPU and the I/O-APIC as a part of the chipset. This combination of a local and system-wide part provides the possibility not only to forward external interrupts to specific cores but also to create a mechanism that is used for inter-subject signalization. The Muen SK implementation distinguishes between Exceptions, Software Generated Interrupts, Traps and Events. These types of interrupts are then handled differently according to the VMX mode and the subject class. As an example, VM subjects running in VMX non-root mode must implement their own exception handling and hence exceptions and software generated interrupts must not result in a subject exit, whereas an exception occurring during the regular execution of the Muen SK in VMX root mode would indicate a serious problem in the kernel code and hence halt the whole system. Further details, especially related to the handling of the different interrupt types, can, again, be found in the Student Research Study in chapter 2, section 2.3, page 18 and following pages.

⁸[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.1, page 10 ff.

⁹[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.2, page 11 ff.

¹⁰[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.5, page 24 ff.

¹¹[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Memory, chapter 3, section 3.4.2.1, page 24.

As for every software written for an integrated circuit, the Muen SK also needs a timer to first of all schedule the different subjects but also to be able to implement a system-wide synchronization barrier to avoid any interprocessor drifts¹². Therefore, the most important timer used by the Muen SK is the VMX preemption timer provided by Intel's virtualization extension that allows the kernel to hand over execution to a subject for a beforehand specified time slice and to return to the hypervisor after the timer has expired in a preemptive way, i.e. regardless of what operations a subject was performing¹³. However, a timer usable for the implementation of the Muen SK must also provide valid accessibility across all cores in order to be able to implement the required system-wide synchronization barrier.

The last important component of the Muen SK implementation is the programming language SPARK, a specialized well-defined subset of the Ada general-purpose language designed for high integrity software. This programming language was chosen by the founder of the Muen SK project due to the ability of SPARK to formally prove the correctness of the software with respect to a variety of program properties before the program is executed. An overview to the possibilities of SPARK can be found the Student Research Study¹⁴ and the book Programming in Ada 2012¹⁵.

2.2 ARM Overview

The Advanced RISC Machines ARM architecture denotes a Reduced Instruction Set Computing RISC microprocessor design by ARM Limited. Due to the focus of the processor architects at ARM Limited on a low number of transistors and hence a low power consumption and heat generation, ARM processors were in the past mainly used in the embedded area. With the ARMv8-A architecture introduced in 2011, ARM Limited has presented its first 64-bit architecture with a virtualization extension not only applicable for embedded systems but also interesting for personal computers and server systems as an alternative to Intel and AMD processors¹⁶.

To be able to understand the structure of all the reference manuals and technical documents needed to develop software for a specific ARM system, one has to bear in mind that ARM Limited - in contrast to the popular Intel processors - does not manufacture the processors itself, but grants design licenses to semiconductor manufacturing companies. Due to this licensing strategy of ARM Limited, there exists a large number of different so called ARM-based System on Chip (SoC) that correspond to a combination of an ARM specified processor as CPU together with other peripheral devices and coprocessors. Therefore, the following document structure and development hierarchy have to be considered for developing software for a specific SoC¹⁷:

¹²[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Scheduling, chapter 3, section 3.4.7, page 29 f.

¹³[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.4, page 23 f.

¹⁴[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.4, page 23 f.

¹⁵[1] Barnes. *Programming in Ada2012*. 2018, chapter 27, section 27.6, page 839 ff.

¹⁶[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, page 29.

¹⁷[3] Langbridge. *Professional Embedded ARM Development*. 2014, chapter 1, page 10 ff.

- (i) *ARM Architecture*: The highest level of abstraction in the specification of an ARM-based SoC is called the processor architecture. The according Architecture Reference Manual defines all the features common to a family of processor's core designs including the assembly instruction set, the supported processor modes and memory management components. The architecture does not specify the actual core architecture conclusively, but delegates some of properties as „implementation defined“ to the ARM core implementation specification. An ARM architecture specification can typically be recognized by the letter „v“ in the naming - for example, ARMv7 denotes an architecture, while ARM7 stands for an (older) core design specification.
- (ii) *ARM Core Implementation*: The second highest level of abstraction is the specification of the actual core design. The according Technical Reference Manual gives additional information on the options left „implementation defined“ by the ARM architecture version but does not add or explain architectural features in a more detailed way. The latest processor implementation specifications by ARM Limited normally contain the term „Cortex“ in their naming - as an example, the name Cortex-A15 denotes a core design with an ARMv7 architecture.
- (iii) *SoC Specific Components*: At the lowest level of abstraction, the SoC manufacturers provide another Reference Manual that contains not only details on the core implementation but also specifies all additional components (i.e. GPU, interrupt controller, etc.) and their usage (i.e. memory mapping of peripheral devices, controller enabling etc.) included in the same integrated circuit, as the accessibility of the processor to the peripherals and its control is not predetermined by ARM. In practice, the SoC manufacturers also often provide fully functional evaluation boards for the software development. Their architecture and implementation details are then usually explained again in a separate Board Reference Manual ¹⁸.

2.2.1 ARMv8-A Architecture

This section basically summarises the ARMv8 fundamentals¹⁹ of the Student Research Study and recapitulates the most important topics in connection with the porting of the Muen SK to the ARM architecture. As already mentioned, ARM Limited not only presented its first 64-bit processor with the introduction of the ARMv8-A architecture, but also added a Virtualization Extension to the processor's architecture. However, the naming is somewhat misleading - the Virtualization Extension is not a single component, but consists of several additional structures on top of already existing components and design principles of the architecture itself. This allows on the one hand to maintain the backward compatibility to the ARMv7-A architecture and on the other hand to support a simplified development of type I and II hypervisors²⁰.

¹⁸There are a variety of different SoC and board architectures with different accessibility strategies: from processor controlled (Odroid C2 with amlogic S905 SoC, cf. <https://www.hardkernel.com/shop/odroid-c2>, December 21, 2018) to CoProcessor controlled (Raspberry Pi 3 with Broadcom 2837, cf. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b>, December 21, 2018).

¹⁹[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.2, page 34 ff.

²⁰5, cf. for AArch64 Virtualization.

The first and most important aspect of the Virtualization Extension is the additional hypervisor layer with respect to the ARM exception level concept provided at the exception level 2. It is important to note, that, unlike on Intel x86/64 architecture, code execution at a higher Exception Level (i. e. an Exception Level EL_n with a larger value for n) has *more* privileges than code execution at a lower one²¹. In addition to a separated privilege level, the Virtualization Extension provides some hypervisor specific registers for a simplified software development as well as an additional Hypervisor Call instruction HVC as an extension to the ARM64 and ARM32 instruction sets. The additional registers allow, for example, to trap register access and instruction execution in the guest systems to the hypervisor on a per subject base. Further details on the exception level principle and the execution states of the ARMv8-A architecture can be found in the Student Research Study²².

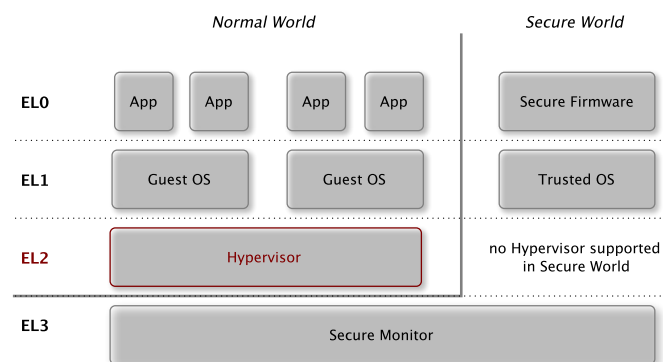


Figure 2.1: ARMv8-A Exception Levels in AArch64 with Hypervisor Level

In addition to the in the ARM A architecture standard memory components, i.e. the Memory Management Unit (MMU) as well as the various caches and cache maintenance functions, the Virtualization Extension provides specific virtualisation-related cache maintenance instructions and a second level address translation mechanism (i.e. so-called Stage 2 translation). This allows a type I hypervisor to implement an additional, for a guest system transparent address translation and to create a separate memory mapping for each subject running at the exception levels 1 and 0. For details with respect to the standard and advanced virtualisation memory mechanisms including multicore-related explanations for the ARMv8-A architecture, it is referred to section 3.4 of the Student Research Study.

In the ARM terminology, all interrupts (i.e. external and internal interrupts, system errors, aborts and software generated interrupts) are denoted as exceptions. While at this point a detailed description of the exception handling is omitted and the corresponding sections in the Student Research Study ref-

²¹ [7] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-1.

²² [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.2.1, page 34 ff., and chapter 3, section 3.2.2, page 37 ff.

erenced²³, the two different possibilities regarding the virtualisation of interrupts have to be mentioned again. Along with the additional Hypervisor Control Register, the ARMv8-A Virtualization Extension provides a simple solution for the virtualisation of exceptions. Due to the design principles used for the Muen SK with its local and system-wide separation of the interrupt handling, this mechanism, however, is not sufficient for the porting of the Muen SK to the ARM architecture. As shown in the course of the Student Research Project, the exception handling has to rely on an additional component also specified by ARM Limited, i.e. the Generic Interrupt Controller starting at version 2 (GICv2)²⁴. Since already many of the existing SoC implement an interrupt controller meeting the GICv2 specification, this only slightly restricts the usage of the Muen SK on different ARM-based SoC. Details on the GIC specified by ARM Limited can, again, be found in the Student Research Study²⁵.

The last important aspect of porting the Muen SK to the ARMv8-A architecture is the ARMv8-A Generic Timer. Due to the scope of this thesis, the following explanations are restricted to the AArch64 execution state²⁶, although the basic timer mechanism are still the same for the AArch32 execution state except for the differently mapped and named registers. Every core design that fulfills the ARMv8-A architecture specification, has to provide at least one SoC-wide system timer, which provides a uniform view of the overall system time. Also mandatory are a memory mapped system counter and a memory mapped virtual timer per core with at least one comparator each, to configure the timers to generate an interrupt when the count is greater or equal to the programmed comparator value (i.e. preemption mechanism). As explained in the Student Research Study²⁷, this three timer and counter components already fulfill the Muen SK requirements. But usually, the core specification includes some optional timers and counters for a variety of combinations of secure state, execution state and exception level. The corresponding Technical Reference Manuals (cf. section 2.2.2) give further details on their accessibility. In addition, the SoC manufacturer is free to implement even more timer and counter components (e.g. NXP Flex Timer Modul for the NXP LS1012A FRDM Board).

2.2.2 ARM Cortex-A53 Implementation

As already mentioned and described in the following chapter, the target platform of this bachelor thesis is the NXP LS1012A with an ARMv8-A Cortex-A53 core. Therefore, this section only briefly discusses the Technical Reference Manual for the corresponding core design²⁸. Since a comprehensive or even final explanation of the details of the core design compared to the architecture specification is neither possible nor useful, a simple example will be given to provide some insights into the interaction between architecture and core design specification.

²³ [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.5, page 47 ff.

²⁴ cf. section 2.2.3

²⁵ [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.5.6, page 51 ff.

²⁶ [6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2018, chapter D10, section D10.1, page D10-2645 ff.

²⁷ [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.6, page 52 ff.

²⁸ [8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2018.

When accessing a memory address, the Memory Management Unit checks (among other things), whether the corresponding process has the required access permissions. These access rights can be defined per page level by using the Memory Attribute Indirection Register (MAIR_EIn). Since this means that only eight different access definitions per exception level can be created by the developer, the architecture specification offers the option of implementing additional access permission definitions with the Auxiliary Memory Attribute Indirection Registers (AMAIR_EIn). The interpretation and accessibility of these registers are declared as „implementation defined“ by the ARMv8-A architecture and therefore left to be specified by the respective core designs²⁹. While other core designs implement and specify these registers to be used for additional memory access rights, the ARMv8-A Cortex-A53 now declares this registers as „not implemented“ and hence „reserved zero“³⁰.

2.2.3 ARM Peripheral Components

ARM Limited not only grants architecture and design licenses for processors, but also specifies further (peripheral) components that can be integrated into a System on Chip. Typically, these specifications have exactly the same structure as the documentation of the processors - i.e. at the highest level of abstraction there is an ARM architecture specification, followed by a design specification also developed by ARM Limited and at the lowest level of abstraction the manufacturer of the SoC provides further details on the implementation of the component.

For this study, only two of these peripheral component specifications have to be considered. First, the Generic Interrupt Controller Architecture Specification³¹ with the CoreLink GIC-400 Technical Reference Manual³² for the exception handling³³ and, second, the CoreSight Components Technical Reference Manual³⁴ as design specification for the official IEEE-Standard JTAG debug interface³⁵ to be able to use the hardware debugger³⁶.

2.3 NXP LS1012A Evaluation Board

In the course of the Student Research Study, it could be showed that the Raspberry Pi 3 is not suitable as a development platform for porting the Muen SK³⁷. Therefore prior to this bachelor thesis, several evaluation boards were examined with respect to various requirement criteria, whereby only boards were considered that come with an ARMv8-A processor and could be qualified as „processor con-

²⁹[6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2018, chapter D12, section D12.2.11, page D12-2705 f.

³⁰[8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2018, chapter 4, section 4.3.54, page 4-90.

³¹[9] n.a. *ARM Generic Interrupt Controller, Architecture Specification*. 2013.

³²[11] n.a. *CoreLink GIC-400 Generic Interrupt, Technical Reference Manual*. 2012.

³³cf. section 3.3.6

³⁴[12] n.a. *CoreSight Components, Technical Reference Manual*. 2009.

³⁵https://de.wikipedia.org/wiki/Joint_Test_Action_Group, December 21, 2018

³⁶cf. section 3.1

³⁷[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 4, page 56 ff.

trolled³⁸. After this evaluation the NXP LS1012A FRDM Board was finally chosen as target platform, although it does not - like all other examined development boards - meet all the technical criteria. The following table 2.3 shows a summary of the results of this evaluation:

Board	Cortex-A53	64-bit	TrustZone	GIC	SMMU	Ada Support ³⁹	DevTools ⁴⁰	Price ⁴¹	AA ⁴²
Pine64 Rock64	✓	✓	✗	✓	~	✗	7	34.95	HDR Media
Pine A64(+)	✓	✓	✗	✓	✗	✗	5	29.00	low cost pc
Odroid C2	✓	✓	✗	✓	~	✗	5	46.00	allrounder
NXP LS1012A FRDM	✓	✓	✗	✓	~	✗	8	53.40	allrounder
Espressobin	✓	✓	✗	✓	✗	✗	4	79.00	network
Macchiatobin	✗	✓	✗	✓	✓	✗	6	269.00	network
Jetson TX1	✗	✓	✓	✓	✓	~	9	582.90	VC, KI

Table 2.1: board evaluation process, final result matrix

2.3.1 Overview

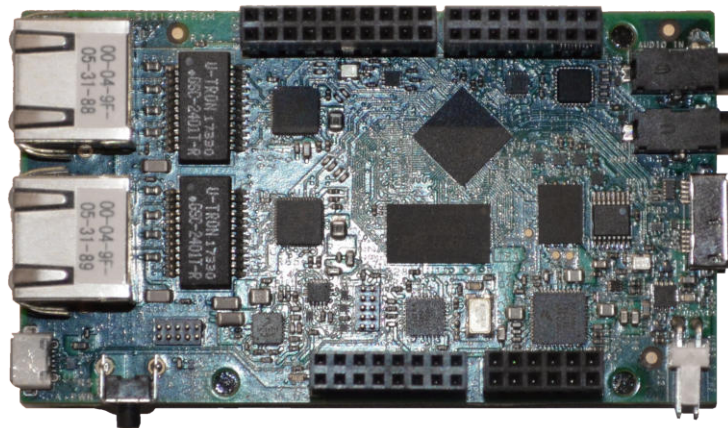


Figure 2.2: NXP LS1012A FRDM evaluation board

The QorIQ LS1012A freedom evaluation board is a low cost development platform for the NXP LS1012A processor series. It is based on a 800 MHz LS1012A ARM Cortex-A53 single processor with 512 mb DDR3L DRAM, a dual 1000Base-T ethernet interface with RJ-45 connectors, USB 2.0 and 3.0 OTG

³⁸[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, page 29, and chapter 4, section 4.1, page 56 f., esp. Figure 4.2: Raspberry Pi 3 schematic.

³⁹i.e. additional community edition support and bb-runtime examples

⁴⁰i.e. other development tool independent of the programming language including debugging, example code, documentation etc. with grades between 1 (abysmal) to 10 (brilliant)

⁴¹i.e. price in US Dollar with 2 GB RAM

⁴²i.e. the application area, the board was designed for

with Micro A/B connectors, a PCIe 2.0 and SATA3 controller, various peripheral interfaces and GPIO connectivity as well as additional debug support via the ARM Cortex 10-pin CoreSight JTAG connector or the CMSIS DAP through a K20 Cortex-M microcontroller. Even though the LS1012A SoC does not feature a System Memory Management Unit (SMMU), the evaluation board was chosen due to the NXP announcements of different i.MX 8 64-bit multi-core SoC with an integrated, the ARM specification fulfilling SMMU ⁴³. Further details on the features of this evaluation board can be found in the according data sheet ⁴⁴ as well as online ⁴⁵.

In contrast to the Raspberry Pi 3 board examined in the Student Research Study, the NXP LS1012A FRDM board is a processor controlled evaluation board ⁴⁶. This means that the ARMv8-A Cortex-A53 on the NXP evaluation board is the organising part of the SoC and has therefore full control over the initialisation of each component. Figure 3.6 shows a schematic overview for the architecture of the target platform for this study.

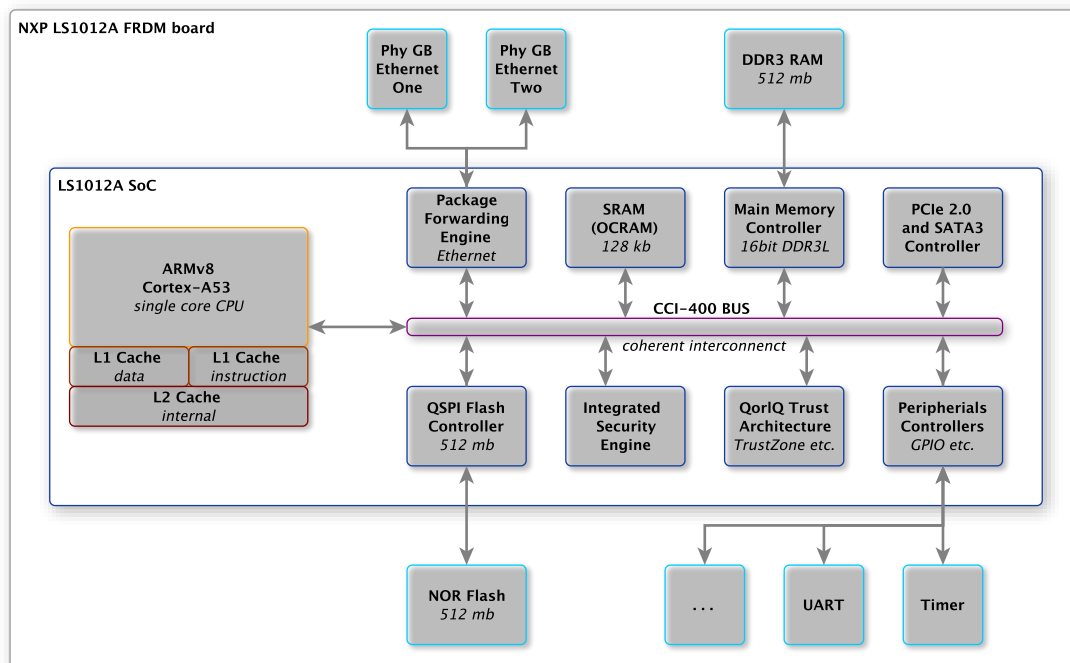


Figure 2.3: NXP LS1012A FRDM schematic

⁴³cf. NXP LS1021A i.MX 8 series <https://www.nxp.com/products/.../i.mx-8-processors:IMX8-SERIES>, December 21, 2018

⁴⁴[16] n.a. *QorIQ LS1012A Data Sheet*. 2018.

⁴⁵cf. for the board specifications <https://www.nxp.com/.../qoriq-frdm-ls1012a-board:FRDM-LS1012A>, and the SoC details <https://www.nxp.com/.../qoriq-layerscape-1012a-low-power-communication-processor:LS1012A>, December 21, 2018

⁴⁶[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 4, section 4.1, page 56 f.

2.3.2 Documentation

Another reason, why the NXP LS1012A board was chosen as the target platform for this bachelor thesis, is the detailed and comprehensive documentation as well as the support and development tools provided by NXP. It has to be mentioned that all NXP documents and tools are only accessible by creating a free NXP developer account. Apart from the literature already mentioned in section 1.4, the following tools and support options are then available via the NXP account:

- *CodeWarrior IDE*: With the as a 30 days test version available CodeWarrior package, NXP not only provides an IDE but also includes some (bare metal) code examples written in assembly and the C programming language⁴⁷. This code examples were used to get a basic understanding of the boot process of the NXP LS1012A Board.
- *QorIQ Linux SDK*: The Linux Software Development Kit (SDK) for QorIQ Processors offers a full-featured development environment consisting of the Linux kernel and device drivers source code, a Yocto Embedded Linux development environment, GNU tools (compilers, linkers, etc.), the adapted U-Boot bootloader source code as well as boot-related firmware, libraries and middleware⁴⁸. This SDK is needed to be able to build a slightly customised U-Boot bootloader that exits at exception level 3 (cf. the following section).
- *Community Forum*: The developer account also grants access to the NXP community forum that is administrated by an NXP support team. In the course of this project, the author also had to rely on this support option⁴⁹.

2.3.3 Board Setup

Like almost every development board with an ARM-based SoC, the NXP LS1012A FRDM too has quite an unexpected boot process. As soon as the board is turned on, the firmware initialises all components to their default values, executes the ARM TrustZone Secure Boot bootloader and exits on exception level 3 into the secure monitor. The only boot option available after the firmware initialisation is the QSPI flash memory device⁵⁰. The development boards are delivered with the factory settings for the QSPI Flash as shown in table 2.3.3 and hence start with the pre-boot loader (PBL) composed of the Reset Configuration Word (RCW) and the pre-boot initialisation (PBI) commands. This pre-boot loader not only configures all peripheral devices (e.g. the UART GPIO Pin Mux) but also initialises the DDR3 RAM component on the NXP LS1012A Board⁵¹. After the pre-boot loader, the U-Boot bootloader is called and initialises the DUART serial communication peripheral to be able load and execute custom ARMv8-A binaries from within the U-Boot console.

⁴⁷cf. https://www.nxp.com/.../codewarrior-development-tools:CW_HOME, December 21, 2018

⁴⁸cf. <https://www.nxp.com/.../linux-sdk-for-qorIQ-processors:SDKLINUX>, December 21, 2018

⁴⁹cf. <https://community.nxp.com/thread/487093>, December 21, 2018

⁵⁰[17] n.a. *QorIQ LS1012A Getting Started Guide*. 2016, chapter 11, page 11.

⁵¹[14] n.a. *QorIQ LS1012A Application Note, PBL Configuration using QCVS*. 2016, for more details.

Start Address	End Address	Image	Maximum Size
0x4000_0000	0x400F_FFFF	RCW and PBI	1 MB
0x4010_0000	0x400F_FFFF	U-boot boot loader and PFE binary	1 MB
0x4020_0000	0x401F_FFFF	U-boot environment	1 MB
0x4050_0000	0x409F_FFFF	PPA FIT	2 MB
0x40A0_0000	0x43FF_FFFF	Kernel ITB	59 MB

Table 2.2: NXP LS1012A QSPI Flash Layout

As already mentioned in the section 1.2, it soon became apparent in the course of the project that the hypervisor has to make additional configurations with regard to the physical interrupt handling⁵² and the Generic Interrupt Controller (GIC)⁵³ at exception level 3. Therefore, the NXP LS1012A Board with its factory settings can not be directly used for running the code developed during this bachelor thesis. Due to the absence of a QSPI Flash driver support in the OpenOCD software, the simplest way to update the evaluation board with a suitable bootloader is as follows⁵⁴:

- (1) *Pre-boot loader and U-Boot Binaries*: First, the latest Linux SDK for QorIQ Processors has to be downloaded from the NXP software page and installed according to the package installation instructions. Following the build instructions also provided within the package, the latest PBL and U-Boot binaries can then be built. Since the entire build process is relatively complex, the two binaries to be used can be found in the *scripts* folder of the *muensk* project in the directory *u-boot*. Due to the uncertain license situation, it is important to note that the binaries should only be used in the context of this bachelor thesis (i.e. educational use only).
- (2) *TFTP Server Setup*: In a second step, a TFTP server in a local network has to be set up. For this study, a Raspberry Pi 3 running the official TFTP software (i.e. `tftpd-hpa`) provided by the Debian-based Raspbian operating system was used. An installation and configuration guide can be found at <https://www.cyberciti.biz/faq/install-configure-tftp-server-ubuntu-debian-howto>. To be able to load the binaries directly from the development environment on the desktop computer to the TFTP server, an additional FTP server was configured on the Raspberry Pi 3 according to the setup guide published at <http://raspberrypi-projects.com/.../vsftpd-ftp-server>. The support scripts for the whole process can also be found in the *scripts* folder of the *muensk* project. The final setup for this project was the following:

⁵²cf. chapter 3, section 3.3.1

⁵³cf. chapter 3, section 3.3.6

⁵⁴**Attention** - even though this is the easiest way to update the bootloader, it is still possible to brick the NXP evaluation board by not exactly following the NXP Getting Started Guide or messing up the RCW or PBI binaries. If something went wrong with the update of the code in the flash memory the only option to unbrick the board is to use a JTAG hardware debugger tool with appropriate QSPI flash support.

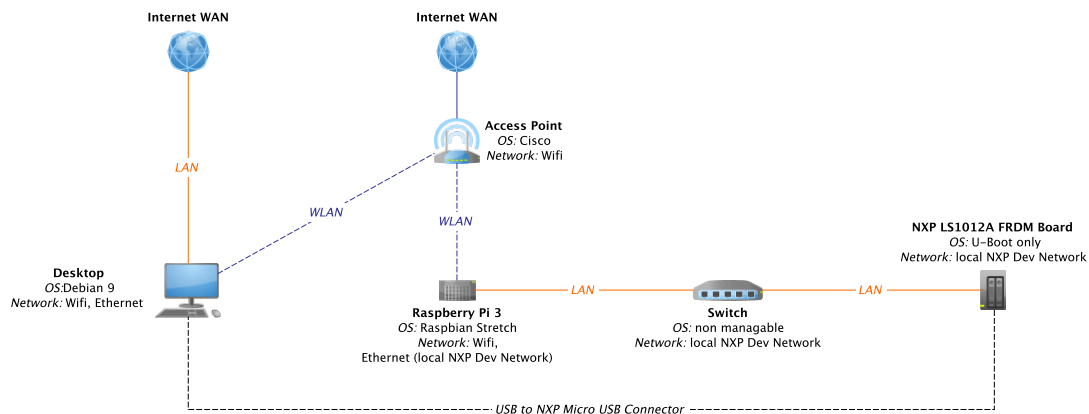


Figure 2.4: TFTP Server Setup

- (3) *QSPI Flash Update*: Finally, the pre-boot loader and U-Boot binaries have to be uploaded to the TFTP server and written to the QSPI flash memory of the NXP LS1012A FRDM Board. This can be achieved **(a)** by connecting the evaluation board with the Micro USB to USB cable to the development computer and starting the CoolTerm serial console with the script from the repository, **(b)** by resetting the board, stopping the autoboot and entering the U-Boot prompt, **(c)** by setting the U-Boot environment variables with the correct IP addresses for the NXP LS1012A FRDM Board and the TFTP server and resetting the board again; **(d)** and, last, by executing the following update commands:

```

# Environment Setup
=> printenv bootdelay
=> setenv ipaddr <board_ip>
=> setenv serverip <server_ip>
=> saveenv
=> reset
..
# PBL Update
=> tftp 0x80000000 PBL_0x33_0x05_800_250_1000_default.bin;
=> sf probe 0:0; sf erase 0 40000; sf write 0x80000000 0x0
    40000;
# U-Boot Update
=> tftp 0x80000000 u-boot.bin;
=> sf probe 0:0; sf erase 0x100000 80000; sf write 0x80000000
    0x100000 80000;
# Linux Kernel Erase
=> sf probe 0:0; sf erase $kernel_start $kernel_size;
  
```

After the QSPI flash memory is updated, the custom ARMv8-A binaries start execution at exception level 3. However, to be able to deploy the code of this project, the U-Boot environment variables have to be set according to the configuration file in the appendix [C](#).

3 Practical Part

This chapter describes those aspects of the bachelor thesis that are closely related to the development of the code written in assembly and Ada/SPARK. In the first part, a detailed explanation of the development environment is given, while as the second part is dedicated to the current project and code structure elaborated during this study.

3.1 Development Environment

identifier	description	link
<i>Method</i>	Dual Boot Desktop PC	-
<i>Operating System</i>	Debian 64-bit 9.2 64-bit	Debian Download
<i>Native Toolchain</i>	AdaCore GNAT Community Edition	AdaCore Download
<i>Cross Toolchain</i>	AdaCore GNAT AArch64 Pro Edition	-
<i>IDE</i>	AdaCore GPS Community Edition	AdaCore Download
<i>Debugger Code</i>	AdaCore GDB AArch64 Pro Edition	-
<i>Debugger Probe</i>	Segger J-Link Edu	J-Link Probe
<i>Debugger Driver</i>	Segger J-Link Driver Package	J-Link Download
<i>Debugger Connect</i>	OpenOCD 0.10.0 (customised)	OpenOCD

Table 3.1: Development Environment Overview

As the development environment has been most important for the success of this bachelor thesis, this whole section is dedicated to its setup. An overview of all the required tools is given in table 3.1. Due to the requirements of the above mentioned software, the environment setup is based on a Debian 9 operating system with a Gnome desktop. However, the installation of the base system will not be discussed further as there are enough installation guides and configuration options to be found in literature and online.

3.1.1 Toolchain

To be able to build all the source code developed during this study, a native Ada/SPARK toolchain for the test project and a Ada/SPARK ARMv8-A AArch64 cross toolchain for the main source code are needed. While the native toolchain provided by the AdaCore Community Project is freely available for almost all operating systems ¹, the GNAT Pro AArch64 ELF cross toolchain currently used for the development is

¹cf. <https://www.adacore.com/download>, December 21, 2018

part of the AdaCore GNAT Pro Edition² and therefore a valid license is needed. Owing to the request of the Muen SK developers, i.e. Adrian-Ken Rüegsegger and Reto Bürki, AdaCore provided a free license for this bachelor thesis. The according installation package is provided with the source code, but it is important to note that this toolchain should only be used in the context of this bachelor thesis (i.e. educational use only). The installation of both the native and the cross toolchain can be done without problems on any Linux derivative with the installation script included in the AdaCore packages.

An alternative way to build a GNAT ARMv8 AArch64 toolchain with the open source GCC GNU Compiler Collection and the software tool crosstool-ng can be found in the evaluation case documentation of the Student Research Study³, although this option could not be tested for the current source code during this bachelor thesis.

3.1.2 Integrated Development Environment

Due to the integration possibilities for the GNAT toolchain, the OpenOCD client and the GNU Debugger (GDB) software, the GPS Integrated Development Environment (IDE) developed by AdaCore was used during the entire project. The IDE is part of the AdaCore toolchain packages and is automatically installed during the AdaCore toolchain installation process.

Apart from the standard tools offered by a full-featured IDE, the GPS IDE also provides a test suite for native unit tests. This suite is used by the Muen SK test project that can be found in the root directory of the code repository.

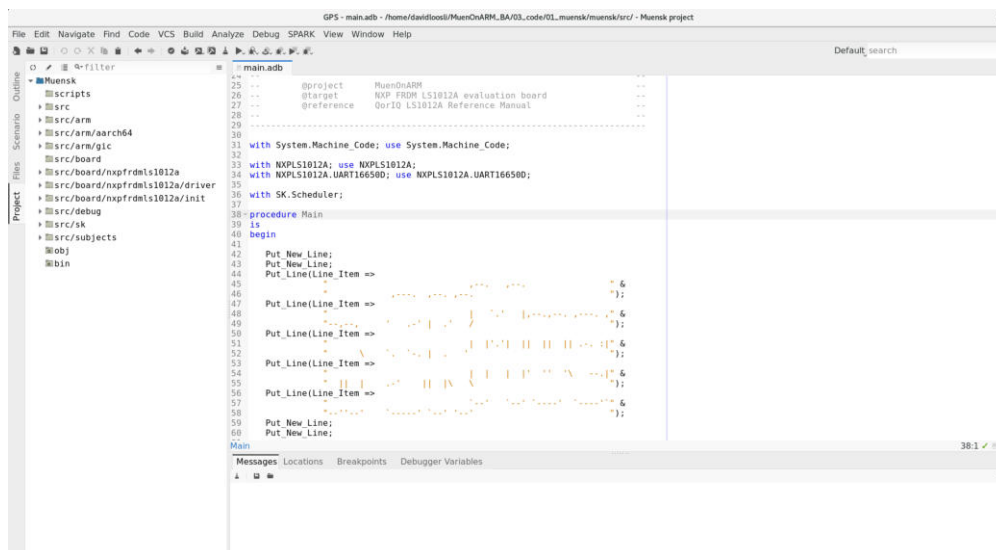


Figure 3.1: GPS IDE overview

²cf. <https://www.adacore.com/gnatpro/>, December 21, 2018

³Problem Description Toolchain - Ada toolchain ARMv8 AArch64, chapter 2, page 3 ff.

3.1.3 Debugger Setup

Even though the development of bare metal software does not necessarily require a hardware debugger⁴⁵, the usage of the J-Link Debug Probe in connection with the GNU Debugger and OpenOCD has been of key importance to accomplish the objectives within the time constraints of this bachelor thesis. However, the main problem in this context is that there are hardly any open source or free software products that support the latest ARMv8-A architecture. Therefore, a working combination of hardware and software tools had to be assembled that could provide the desired debugger functionality with as few adjustments as possible.

J-Link Debug Probe

For this project, the Segger's J-Link Edu Version 10.1⁶ debug probe was chosen as the in-circuit debugger after a thorough examination of various devices and possibilities during the Student Research Study⁷, even though the J-Link software provided with this device does not support the ARMv8-A Cortex-A53 processor yet⁸. The debug probe is used to connect the JTAG header on the NXP LS1012A board to the developer's desktop computer.

Although, this setup uses the OpenOCD software to guarantee a working communication between the J-Link Edu debug probe and the GDB server provided with the GPS IDE, the official drivers from the Segger download page may have to be installed too, depending on the operating system of the development setup. The J-Link Software bundle as well as the documentation can be downloaded from [here](#). As an example, the installation commands for the Debian 9 distribution would be:

```
$ cd /path/to/JLink_Linux_V640_x86_64.deb
$ su
# enter your root password
$ dpkg -i JLink_Linux_V640_x86_64.deb
# and follow the installation instructions
```

Finally, the NXP LS1012A FRDM Board has to be wired correctly to the Segger J-Link Edu debug probe. Because the NXP evaluation board implements the official JTAG 10-pin interface specified by ARM Limited, the according pinout description can be found in the CoreSight Components Technical

⁴Alternatives to a hardware debugger would be an according configured UART communication or a simulator, although every consulted literature, that covers this topic, recommends using a hardware debugger especially when writing boot code (cf. chapter 9, page 159 ff., [3])

⁵[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.1, page 30 ff.

⁶<https://www.segger.com/products/debug-probes/j-link/models/j-link-edu/>, December 21, 2018

⁷[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.1.2, page 31 f.

⁸At the end of the second quarter of 2018, Segger released a beta version of the J-Link software package with ARMv8-A support (cf. <https://www.segger.com/news/j-link-64-bit-support>, December 21, 2018). However, this package is currently only supported by the J-Link Ultra+ debug probe (cf. <https://forum.segger.com/.../5234-SOLVED-armv8-iMX8M-Support> and <https://www.segger.com/.../model-overview>, December 21, 2018).

Reference Manual⁹. The pinout description for the Segger J-Link Edu debug probe is explained in the Segger J-Link documentation¹⁰. The final setup is shown in figure 3.2.

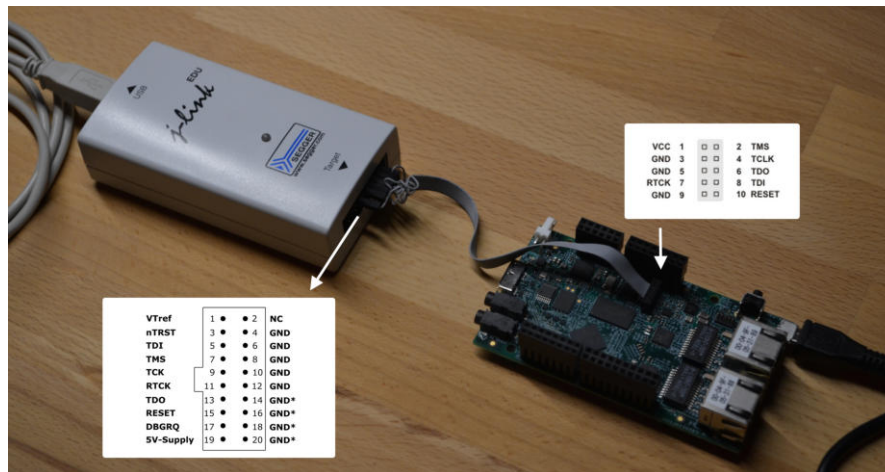


Figure 3.2: J-Link Debug Probe setup including ARM JTAG pinout

OpenOCD

For the communication between the J-Link debug probe and the GDB server, the Open On-Chip Debugger (OpenOCD) software created by Dominic Rath as part of a diploma thesis at the University of Applied Sciences Augsburg was chosen¹¹. As the ARMv8-A support is still under development, the latest source code from the master branch of the sourceforge repository¹² has to be adapted with two patches provided with the source code for this bachelor thesis¹³. After applying the patches to the downloaded source code of the OpenOCD software, the project has to be built and installed according to the instructions that can be found in the root directory of the OpenOCD master branch.

To be able to integrate the OpenOCD software into development process with the GPS IDE, an OpenOCD configuration file (i.e. muenproj.cfg) had to be developed to start the OpenOCD server correctly from within the IDE. The configuration file can be found in the bachelor thesis' code repository in the directory 01_muensk → scripts → jtag. After copying the configuration file into the GPS project folder, the GPS project settings have to be adjusted according to figure 3.3. Before starting the GDB-based debugger from within the GPS IDE, the OpenOCD server must be initialized via the menu entry Build → Bareboard → openocd as shown in figure 3.4.

⁹[12] n.a. *CoreSight Components, Technical Reference Manual*. 2009, appendix C, section C.2.2, page C-5 f.

¹⁰[13] n.a. *J-Link / J-Trace User Guide*. 2018, chapter 15, section 15.1.1, page 337.

¹¹<http://openocd.org>, December 21, 2018

¹²<https://sourceforge.net/p/openocd/code/ci/master/tree/>, December 21, 2018

¹³At the beginning of the project, three patches were needed. But with one of the changes of the OpenOCD software in the course of this study, one of the source code changes can now be omitted.

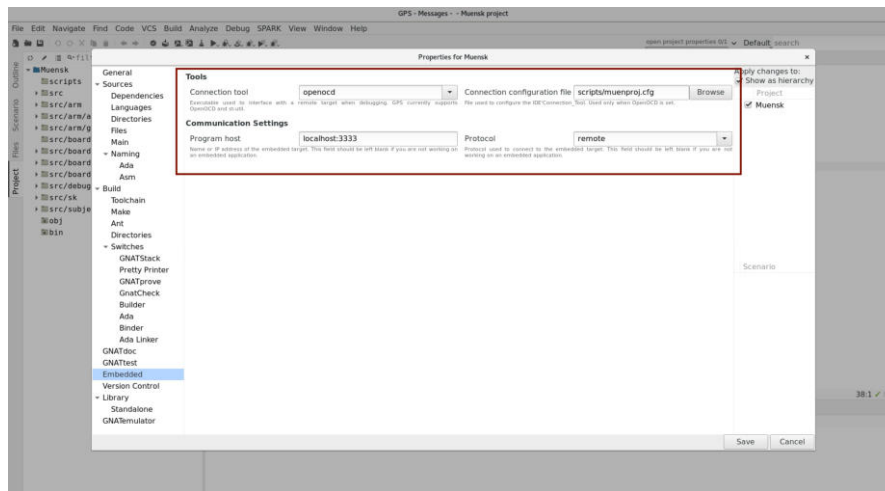


Figure 3.3: GPS IDE OpenOCD settings project

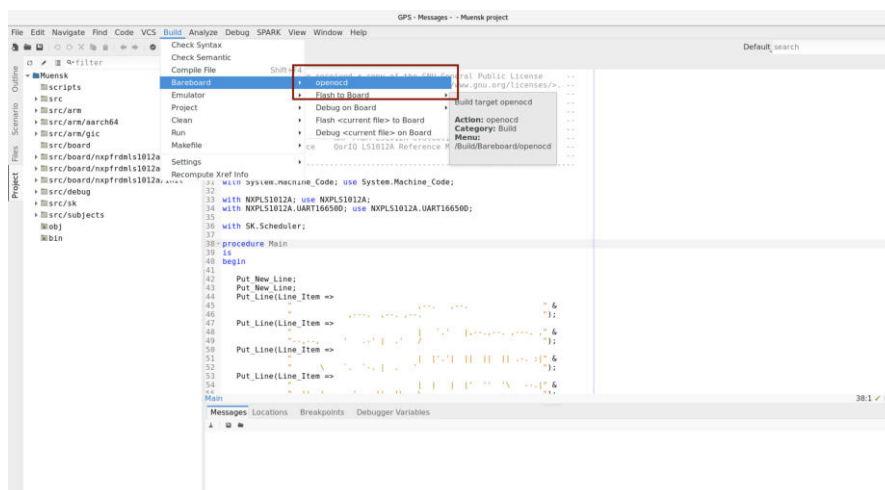


Figure 3.4: GPS IDE OpenOCD integration

GNU Debugger

The third component of the hardware debugger setup is the GNU Debugger (GDB) software ¹⁴ that allows the developer to send debugging commands over the OpenOCD server to the JTAG interface of the NXP LS1012A FRDM Board. The GBD software tools are provided with the AdaCore toolchain bundles and automatically installed during the according installation process (cf. section 3.1.1).

¹⁴<https://www.gnu.org/software/gdb/>, December 21, 2018

The standard initialisation of the GDB server by the GPS IDE is not sufficient for the Muen SK project. Therefore, a specific GDB initialisation script has to be created in the root directory of the project that not only configures the GDB server according to the setup but also adds project-specific load function and post setup hooks. An example for a Muen SK GDB initialisation file can be found in the appendix D as well as in the source code script folder delivered with this project documentation.

Per default, the GPS IDE is not allowed to load a project-specific GDB initialisation script from the root folder of the GPS project. Therefore, a user defined `.gdbinit` file with a corresponding entry has to be created in the user's home directory. It is recommended to limit the access of the GDB autoload function to the currently used project-specific GDB initialisation scripts with:

```

$ cd $HOME
$ nano .gdbinit
# enter the following line and save the file
add-auto-load-safe-path /path/to/project/.gdbinit
  
```

Finally, the hardware debugging process can be initialised by starting the debugger from within the GPS IDE without the usage of a main file ¹⁵ as shown in the following figure.

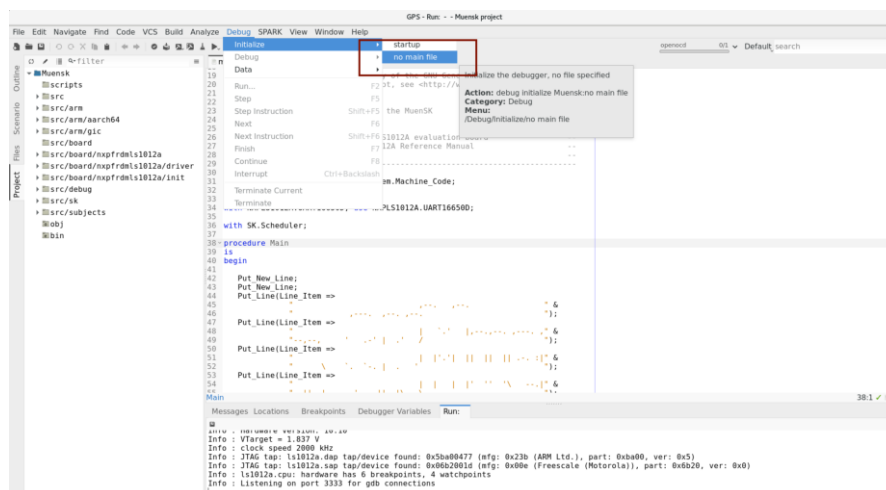


Figure 3.5: GPS IDE GDB integration

Final Debugging

By using the setup described in this section, it is now possible to compile, build, load and execute the Muen SK binary directly from within the GPS IDE. After starting the OpenOCD server and initialising

¹⁵ Calling the debugger with the current main file would automatically set a breakpoint at the first instruction executed on the NXP LS1012A FRDM Board, before the binary is loaded correctly into the DDR RAM.

the debugger without a main file, the GPS IDE switches to the debugger view. Owing the customised GDB initialisation script, the binary can now be loaded into the DDR RAM of the NXP LS1012A Board by executing the command `load_with_subjects` within the debugger console.

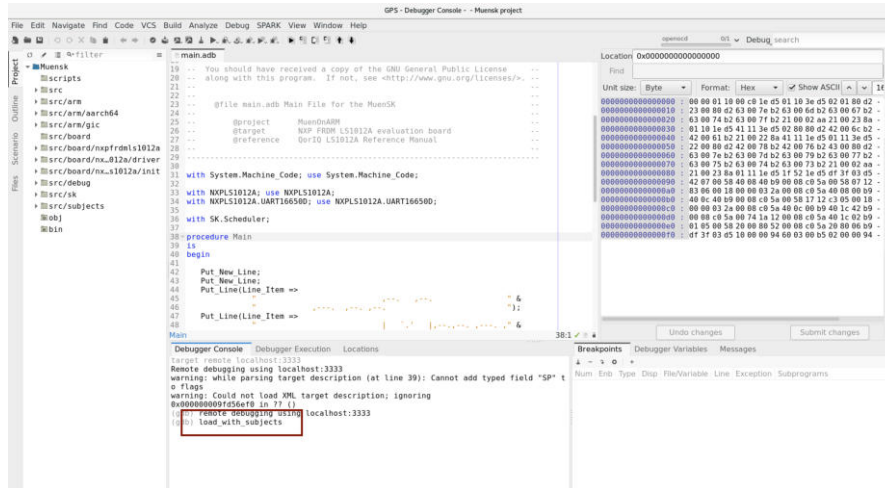


Figure 3.6: GPS IDE GDB load binary into DDR RAM

Due to the customised OpenOCD software, the debugger is now capable of giving the developer a lot of information with respect to the hardware register states (1). After loading the binary into RAM and setting some breakpoints and watchpoints in the debugger console with the appropriate GDB commands, the program can finally be executed by either running the command `continue` (2) or pressing the continue button from the GPS IDE (3).

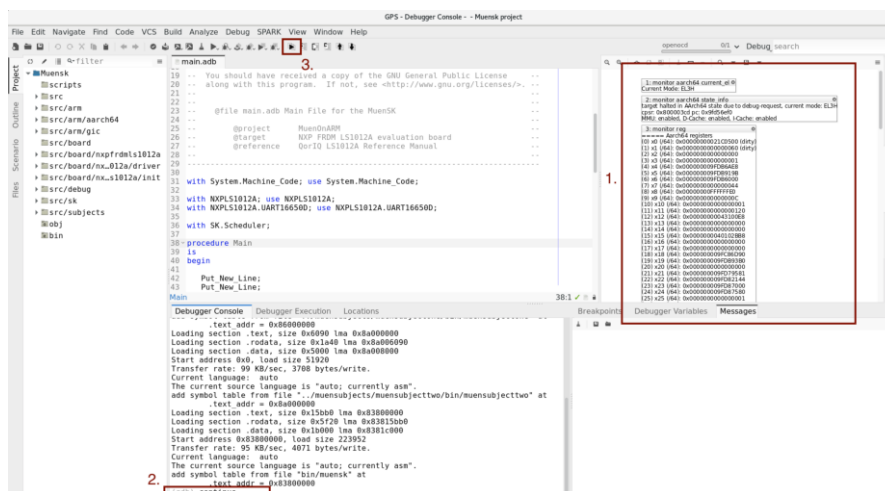
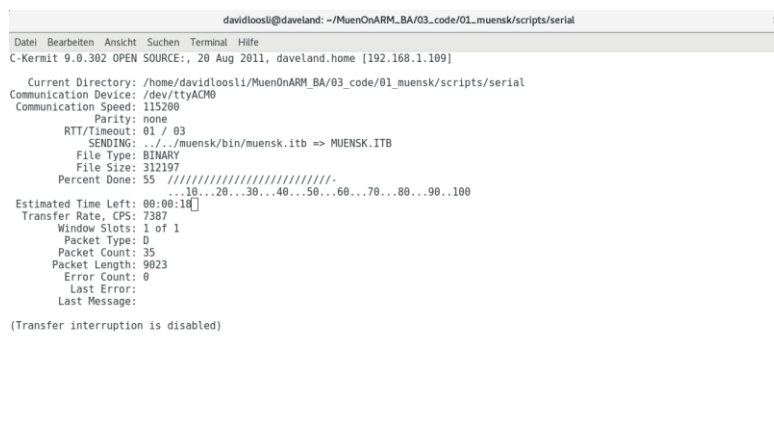


Figure 3.7: GPS IDE GDB execute binary from DDR RAM

3.1.4 Deployment

In order to offer the supervisors the possibility to load and execute the master branch of the source code during the whole project without the need of a hardware debugger setup, the code repository contains two Kermit scripts for the deployment of the Muen SK binary via the serial interface. These scripts can be found in the folder `01_muensk` → `scripts` → `serial`. For NXP LS1012A FRDM Boards, that are still equipped with the factory default bootloader versions, the Kermit script for running the binaries at exception level 2 has to be executed. However, due to the implementation of the exception handling based on the Generic Interrupt Controller (GIC), the latest source code has to be executed starting at exception level 3 and therefore the deployment has to be done by using the according Kermit script for EL3 binaries and an updated NXP evaluation board as described in section 2.3.3.

To be able to execute the scripts, the `ckermi` package has to be installed first¹⁶. After making the scripts executable with the well-known `chmod +x <filename>` command and building the Muen SK GPS project, the binaries can be loaded and executed by running the command `./serialcon_boot_el<n>` in a terminal. If everything has been set up correctly, the binary is first loaded over the serial connection onto the evaluation board and then executed by the bootloader.



```

davidloosli@daveland: ~/MuenOnARM_BA/03_code/01_muensk/scripts/serial
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
C-Kermit 9.0.302 OPEN SOURCE: 20 Aug 2011, daveland.home [192.168.1.109]

Current Directory: /home/davidloosli/MuenOnARM_BA/03_code/01_muensk/scripts/serial
Communication Device: /dev/ttyACM0
Communication Speed: 115200
Parity: none
RTT/Timeout: 01 / 03
SENDING: ./../muensk/bin/muensk.itb => MUENSK.ITB
File Type: BINARY
File Size: 312197
Percent Done: 55
Estimated Time Left: 00:00:18
Transfer Rate, CPS: 7387
Window Slots: 1 of 1
Packet Type: 0
Packet Count: 35
Packet Length: 9023
Error Count: 0
Last Error:
Last Message:

(Transfer interruption is disabled)
  
```

Figure 3.8: Kermit Script serial load process

¹⁶With the setup described in the preceding sections, this can be achieved by using the official apt package from the Debian repository


```

davidloosli@daveLand: ~/MuenOnARM_BA/03_code/01_muensk/scripts/serial
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
davidloosli@daveLand:~/MuenOnARM_BA/03_code/01_muensk/scripts/serial$ ./serialcon.boot_el3
Prepared to boot new kernel. Please reset the board now!
Connecting to /dev/ttyACM0, speed 115200
Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
-----
bootm
# Loading kernel from FIT Image at a0000000 ...
Using 'config0' configuration
Trying 'muensk01' kernel subimage
  Description:  muen sk binary
  Type:         Kernel Image
  Compression:  uncompressed
  Data Start:   0xa0000000
  Data Size:    225280 Bytes = 220 KiB
  Architecture: AArch64
  OS:           U-Boot
  Load Address: 0x83800000
  Entry Point:  0x83800000
  Loading Kernel Image ... OK

Console Initialized at EL2

GIC Controller Initialized at EL2 with
-----
-- Generic Interrupt Controller Specification --
-----
GIC Board Specification:
  NXP LS1012A FRDM
GIC Distributor Specification:
  Architecture : GICv2
  Implementation : GIC-400
  Implemented by : ARM Limited
  Number of Components : 1
GIC CPU Interface Specification:
  Architecture : GICv2
  Implementation : GIC-400
  Implemented by : ARM Limited
  Number of Components : 1
GIC Interrupt Specification:
  Number of Interrupts (Total) : 256
  Number of SGI Interrupts : 16
  Number of PPI Interrupts : 16
  Number of SPI Interrupts : 224
  Number of Lockable Interrupts : 31
  
```

Figure 3.9: Kermit Script kernel execution start

3.2 Software Architecture

The root directory of the project repository is divided into three parts - the administration, the documentation and the source code folder. The first two directories contain all the administration and documentation files prescribed by the University of Applied Sciences Rapperswil (HSR) ¹⁷. While the documents in the administration folder are written in German, the whole documentation is written in English in accordance with the general conditions discussed during the kickoff meeting. Since these documents should be self-explanatory, the following section only considers the third directory (i.e. the source code folder) and explains the most important design elements of the software architecture of the separation kernel developed in the course of this bachelor thesis. The code directory consists of three independent subprojects:

- (1) *Muens SK*: The 01_muensk directory holds the source code for the zero footprint runtime, the separation kernel and the two subjects. It can therefore be considered as the main source code folder for this study and, hence, is described more detailed in the following sections.
- (2) *Muen SK Test*: This part of the source code is dedicated to the unit test framework GNATtest provided by the AdaCore toolchain bundle. As it is currently not possible to run tests directly on the remote hardware, this subproject only contains algorithmic Unit Tests that can be executed on the development platform with the native AdaCore toolchain. Further details on the GNATtest unit test harness generator can be found at <https://www.adacore.com/gnatpro/toolsuite/gnattest>.

¹⁷ cf. Anleitung: Dokumentation Studien- und Bachelorarbeiten, HSR intern

- (3) *OpenOCD*: At first, this folder contained the fork of the master branch of the whole OpenOCD source code. In the course of the project, however, it was decided to provide only the patches needed for the latest source code from the sourceforge OpenOCD repository. Details on the installation and configuration can be found in the section 3.1 of this paper.

3.2.1 MuenSK Projects

The Muen SK project folder contains four different GPS subprojects, that in contrast to the code repository projects depend highly on each other:

- (1) *muenrts*: The Muen SK uses a Zero Footprint Runtime¹⁸. This runtime was borrowed from the Muen SK project without any substantial adjustments.
- (2) *muensk*: This GPS project implements the basic components of the separation kernel, builds all the necessary binaries (i.e. kernel, runtime and subject binaries) and makes use of the mkimage tool to create a multi binary U-Boot FIT image according to the .its configuration file in the root directory of the muensk project. This multi binary U-Boot FIT image can be loaded and executed over the serial connection (cf. section 3.1.4).
- (3) *muensubjects*: This folder contains two GPS projects of two differently configured native subjects. The source code of the two subjects can be built independently of the hypervisor in use.

For all GPS projects of this study, the project settings have to select the MuenRTS object folder as their runtime. This can be achieved in the toolchain menu entry in the project settings:

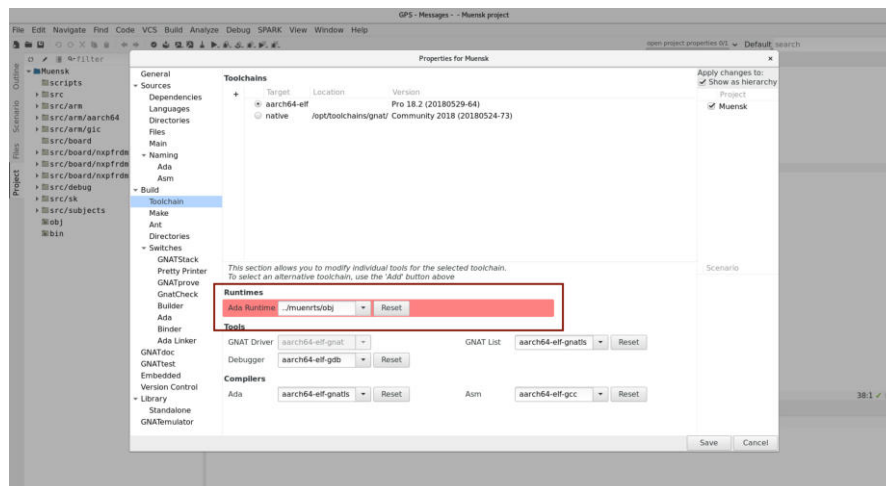


Figure 3.10: GPS IDE runtime settings project

¹⁸[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.7, page 26.

All GPS projects have to be configured as makefile projects with the Makefile located in the root directory of the according projects folder:

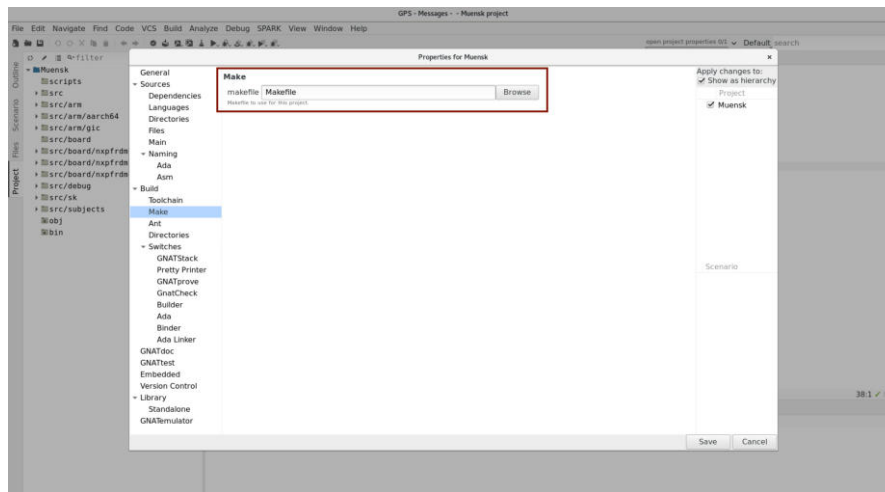


Figure 3.11: GPS IDE makefile settings project

3.2.2 Code Structure

According to the author's experiences in the field of embedded systems and following different hypervisor implementations ¹⁹, the source code for the implementation of the separation kernel as well as for both subjects is structured into two parts. The first two subdirectories (color-coded blue) contain:

- *arm*: This subfolder is structured according to the ARMv8-A architecture and core design specification. While the root directory consists of all the execution state independent ARMv8-A source code, the aarch64 folder relates to code that has to be qualified as AArch64 execution state only. In addition, all ARM components, that are described in a separate specification document, are also architecturally separated (i.e. currently the Generic Interrupt Controller GIC only).
- *board*: This source code folder holds board specific code, e.g. all driver implementations, the board initialisation related code and the board's startup assembly file. For every new target platform, a separate directory should be created.

The second type of subdirectories (color-coded orange) contains the project-specific source code files. In the example of the separation kernel shown in Figure 3.12, the files are separated into the debug specific, the kernel specific, and the subject initialisation files.

¹⁹e.g. Xen Hypervisor <https://xenbits.xen.org/gitweb/?p=xen.git;a=summary> and HASPOC Hypervisor <https://haspoc.sics.se/source.html>, December 21, 2018

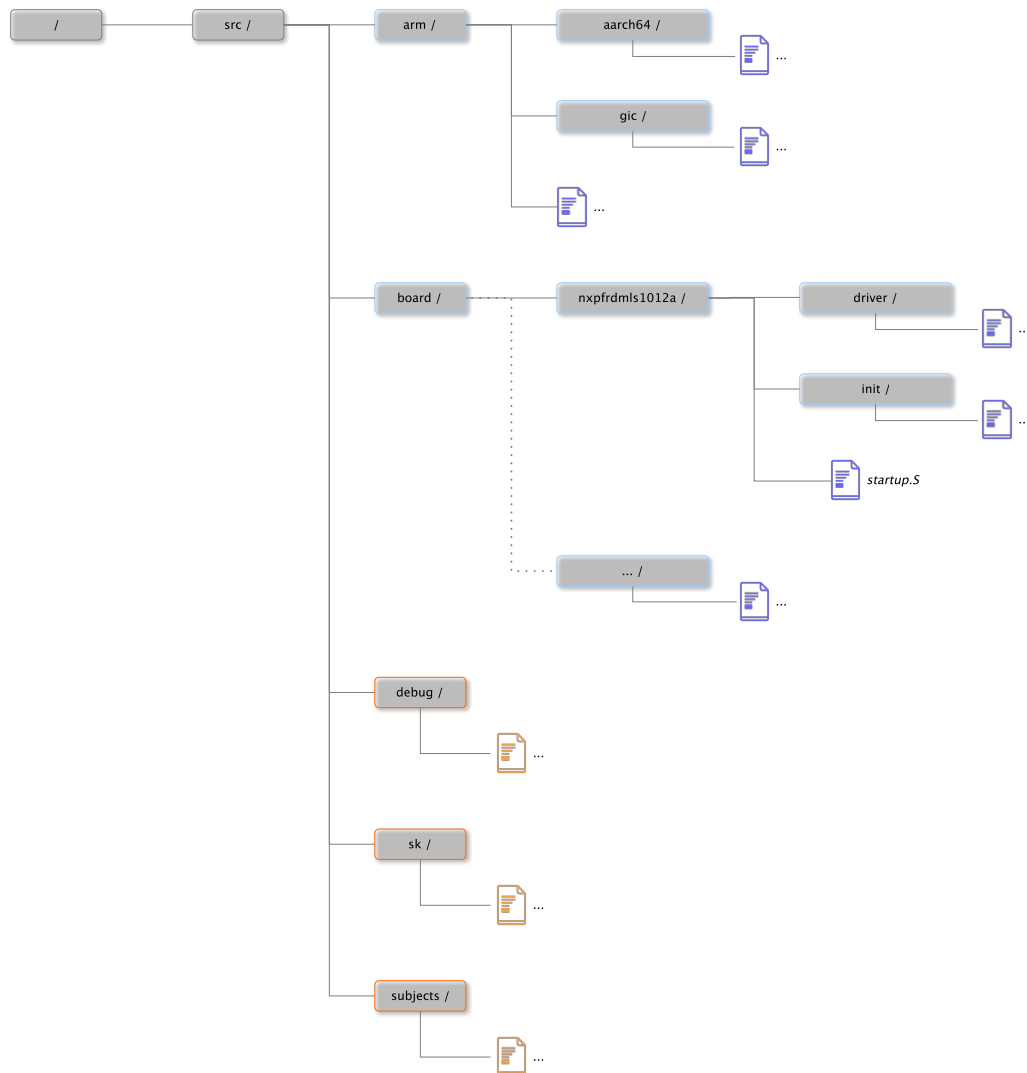


Figure 3.12: Muen SK separation kernel code structure

3.2.3 Code Style

The source code was written in accordance with the Muen Coding Guidelines ²⁰. In addition, all implemented procedures, functions and initialised registers are provided with a header that consists of the according name, the type, the visibility, a brief description, a detailed description and, if needed, other information like the accessibility or the parameter specifications.

²⁰The corresponding guidelines can be found in the literature folder on the enclosed USB data carrier.

```

— @name <name> —
— @type <procedure|function|register> —
— @visibility <private> —
— — —
— @brief <brief_description> —
— — —
— <long_description> —
— — —
— @param <param> —
— — —
— @accessibility <RW|WO|RO> —
— — —

```

3.2.4 License

As stated in every source code file, the whole project is basically licensed according to the GNU General Public License version 3 (GPLv3). In addition, the author also signed the Codelabs Contributors Agreement. Its content can be found in the appendix [E](#) of this document.

```

— — —
— Copyright (C) 2018, David Loosli <dloosli@hsr.ch>, —
— University of Applied Sciences HSR, Rapperswil —
— p.p. Reto Buerki <reet@codelabs.ch> —
— and —
— p.p. Adrian-Ken Rueegsegger <ken@codelabs.ch> —
— — —
— This program is free software: you can redistribute it and/or modify —
— it under the terms of the GNU General Public License as published by —
— the Free Software Foundation, either version 3 of the License, or —
— (at your option) any later version. —
— — —
— This program is distributed in the hope that it will be useful, —
— but WITHOUT ANY WARRANTY; without even the implied warranty of —
— MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the —
— GNU General Public License for more details. —
— — —
— You should have received a copy of the GNU General Public License —
— along with this program. If not, see <http://www.gnu.org/licenses/>. —
— — —
— @file <filename> <description> —
— — —
— @project MuenOnARM —
— @target <target> —
— @reference <reference> —
— — —

```

3.3 Implementation Details

The last section of the practical part highlights some of the most important implementation details. For a deeper insight into the implementation itself, it is referred to the source code of this bachelor thesis.

3.3.1 Startup Code

Due to the licensing strategy of ARM Limited and hence the various SoC designs, the first special case concerns the startup code. In ARM-based systems, the startup code for different SoC and evaluation boards usually slightly differs. In the source code, it is taken account of this differences by

- (i) separating the various board and SoC related source code;
- (ii) moving the startup code file to the according SoC or board root directory (cf. figure 3.12);
- (iii) implementing the code in assembly and only using the ARMv8-A ARM64 instruction set ²¹.

To simplify the integration of further ARM-based systems, an abstract specification of the hardware state before entering the actual separation kernel is required. As already mentioned in the section 1.2, it is therefore currently assumed that the according evaluation board

- (i) configures and initializes the existing random access memory (RAM) components, as the separation kernel source code does not provide any firmware capabilities apart from the UART communication driver;
- (ii) establishes an identity mapping as well as configures and enables the Memory Management Unit (MMU) if necessary for the use of the RAM component;
- (iii) hands over the execution to the separation kernel code at exception level 3 (i.e. secure monitor mode) in a legal hardware state.

Despite the above-mentioned measures, there still can be observed differences between the various SoC. For example, the common start handler would not be necessary for the current implementation, because the NXP LS1012A FRDM board is already a single core system. In contrast, the Odroid C2 development board is based on Amlogic ARMv8-A Cortex-A53 quad core SoC and hence the slave cores would have to be put into a waiting state, since the current implementation only supports single core systems.

²¹ Due to the backward compatibility of the ARMv8-A architecture with respect to the ARMv7 specification, the A32 and Thumb32 instruction sets are supported too (cf. appendix B, page 203, [3]), but must not be used in the case of porting a SoC or an evaluation board to the Muen SK project.

3.3.2 Registers

Due to the strong typing principle of the Ada/SPARK programming language, the implementation of the registers could imitate exactly the ARMv8-A architecture and Cortex-A53 design specifications. This could be achieved by first implementing the register type as a volatile record in the according Ada/SPARK specification file.

```

--- @name Translation_Table_Base_Register_Type (TTBRn) ---
--- @type type ---
---
--- @brief type definition for the 64-bit ARMv8 AArch64 ---
--- translation table base register (see ARM Architecture ---
--- Reference Manual ARMv8, section D7.2.100, p. D7-2521) ---
---
type Translation_Table_Base_Register_Type is record
  Common_Not_Private : Common_Not_Private_Flag := 2#0#;
  Base_Address       : Base_Address_Type       := 16#0000_0000_0000#;
  Address_Space_ID   : ARMv8.Halfword          := 16#0000#;
end record
with Volatile, Size => 64,
  Bit_Order => System.Low_Order_First;

for Translation_Table_Base_Register_Type use record
  Common_Not_Private at 0 range 0 .. 0;
  Base_Address       at 0 range 1 .. 47;
  Address_Space_ID   at 0 range 48 .. 63;
end record;

```

In a second step, the functions to access and set the actual register are declared according to the ARM specifications from the corresponding reference manual applying the following rules: **(a)** If according to the ARM specification the AArch64 register can be mapped to the AArch32 execution state or is accessible from different exception level in AArch64 execution state, a generic function to access and a generic procedure to set the according register has to be implemented; **(b)** for every AArch64 execution state a function to access and a procedure to set the according register with the prefix `_ELn` has to be implemented; **(c)** currently the AArch32 registers have neither to be declared nor implemented. As an example, the Translation Table Base Register (TTBR0) is accessible from all exception levels except for EL0 and can be mapped to an AArch32 TTBR register - therefore, the generic function `Translation_Table_Base_Register_0` to access the TTBR0 register and the corresponding `Set_0` procedure as well as all according functionality for the three exception levels have to be declared and implemented as shown in the following code snippet. In contrast, the Hypervisor Control Register (HCR) is only available at exception level 2, but can be mapped to an AArch32 register - hence, the declaration of a generic access function and a set procedure is required, but only the functions and procedures for exception level 2 are need to be implemented.

— Procedures and Functions Declaration —

```

function Translation_Table_Base_Register_0
  return Translation_Table_Base_Register_Type;
function Translation_Table_Base_Register_0_EL3
  return Translation_Table_Base_Register_Type;
function Translation_Table_Base_Register_0_EL2
  return Translation_Table_Base_Register_Type;
function Translation_Table_Base_Register_0_EL1
  return Translation_Table_Base_Register_Type;

procedure Set_0      (Register_Value : Translation_Table_Base_Register_Type);
procedure Set_0_EL3 (Register_Value : Translation_Table_Base_Register_Type);
procedure Set_0_EL2 (Register_Value : Translation_Table_Base_Register_Type);
procedure Set_0_EL1 (Register_Value : Translation_Table_Base_Register_Type);
  
```

— Procedures and Functions Implementation —

```

— @name Translation_Table_Base_Register      —
— @type function                            —
—                                           —
— @brief returns the translation table base register —
—                                           —
— This function returns the current value of the trans—
— lation table base register 0 at the current exception —
— level (except for EL0 [not accessible])      —

```

```

function Translation_Table_Base_Register_0
  return Translation_Table_Base_Register_Type
is
  TTBR0 : Translation_Table_Base_Register_Type;
begin
  if ARMv8.Current_EL = ARMv8.EL3 then
    return Translation_Table_Base_Register_0_EL3;
  elsif ARMv8.Current_EL = ARMv8.EL2 then
    return Translation_Table_Base_Register_0_EL2;
  elsif ARMv8.Current_EL = ARMv8.EL1 then
    return Translation_Table_Base_Register_0_EL1;
  else
    return TTBR0;
  end if;
end Translation_Table_Base_Register_0;
  
```

```

— @name Translation_Table_Base_Register_0_EL3 —
— @type function —
—
— @brief returns the vector base addr register TTBR0_EL3 —
—
— This function returns the current value of the trans—
— lation table base register TTBR0 at exception level 3 —

```

```

function Translation_Table_Base_Register_0_EL3
  return Translation_Table_Base_Register_Type
is
  TTBR0 : Translation_Table_Base_Register_Type;
begin
  if ARMv8.Current_EL = ARMv8.EL3 then
    System.Machine_Code.Asm (Template =>
      "mrs_ux0,_,_TTBR0_EL3"
      & Standard.Ascii.LF & Standard.Ascii.HT &
      "str_ux0,_,_%0",
      Outputs =>
        Translation_Table_Base_Register_Type '
        Asm_Output("=m", TTBR0),
        Volatile => True,
        Clobber => "x0");
    end if;
  return TTBR0;
end Translation_Table_Base_Register_0_EL3;

...

```

3.3.3 Subjects

The current version of the separation kernel supports two differently configured subjects:

- *Subject One*: This subject is running with a direct page table configuration that shows the possibility of mapping a native subject without second level address translation. It makes use of a hypervisor control mechanism that ensures that changes to page tables are trapped to the hypervisor (i.e. HCR_EL2 TVM bit 26 and TRVM bit 30). The page tables are not directly accessible to the subject because they are stored outside its address space and are therefore only visible for the MMU. Subject One has the following memory mapping ²²:

²²Attention - sizes of the figure do not match the sizes of the memory address spaces!

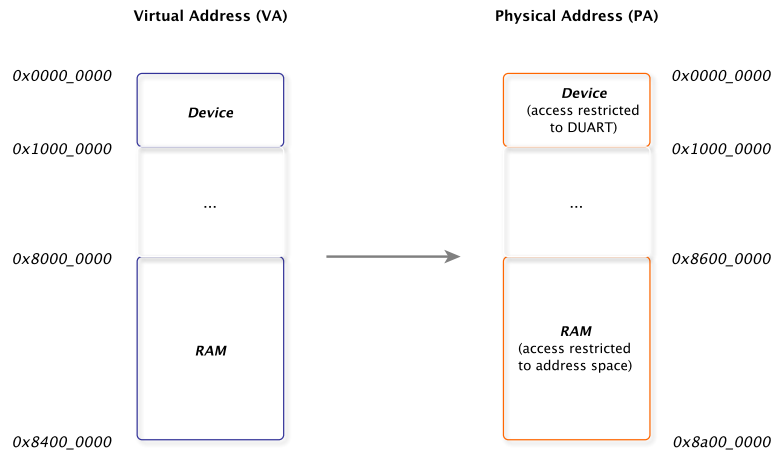


Figure 3.13: Subject One memory map

- Subject Two:** The second subject is running with the Second Level Address Translation enabled (cf. section 3.3.5). It uses the following page table configuration from the Virtual Address (VA) to the Intermediate Physical Address (IPA) defined by TTBR0 and TTBR1 and, finally, from the Intermediate Physical Address (IPA) to the Physical Address (PA) defined by the hypervisors' VTTBR page table ²³:

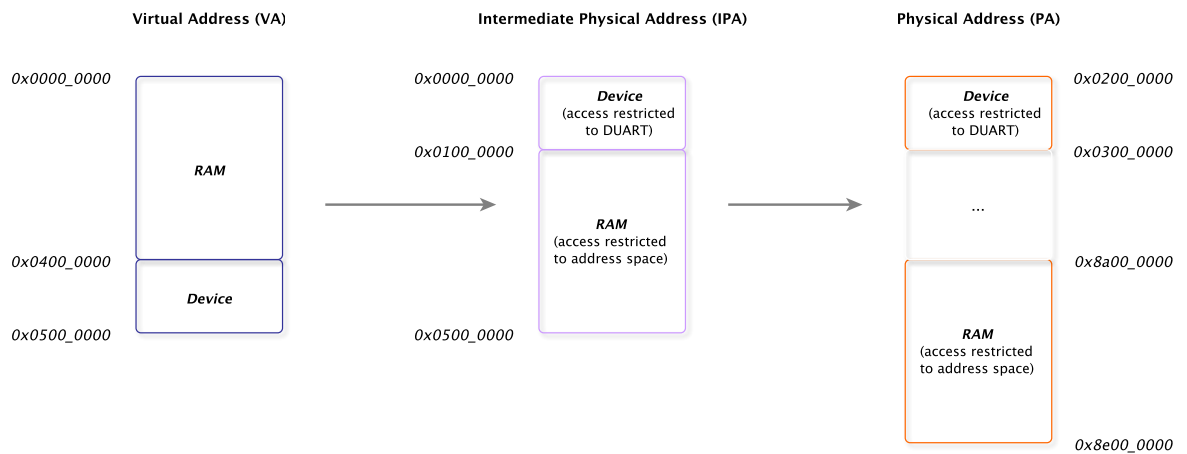


Figure 3.14: Subject Two memory map

²³ Attention - sizes of the figure do not match the sizes of the memory address spaces!

3.3.4 Subject Context Switch

One of the most important components of the separation kernel is the context switch. In contrast to Intel Virtualization Technology (VT), the ARMv8-A Virtualization Extension does not support an automatic save and restore functionality of the corresponding registers²⁴. Therefore, this section provides some information about the registers that have to be stored when switching the context (i.e. exception level 2 to 1 and back with VM entry and VM exit). The following list does not (and currently can not) guarantee completeness but should give an overview of the registers that are needed to be stored by the hypervisor²⁵:

- *Subject Related Registers:*

- (1) *Common Purpose Registers:*

- (i) *GPR*: the general purpose register from x0 to x30.
 - (ii) *SPR*: Special Purpose Register, i.e. especially the Stack Pointer Register SP, the Floating Point Register FP and the Program Counter PC - the processor state is handled as a part of the exception related registers.

- (2) *CPU Related Registers:*

- (i) *CPU Info*: depends on the implementation and could be a composed record of the MPIDR, MIDR and other registers.
 - (ii) *VM Info*: should describe a Virtual CPU Info with information that the hypervisor would like to provide to a subject, e.g. the VMPIDR register and the virtual machine id related registers for a simplified Translation Lookaside Buffer (TLB) handling.
 - (iii) *SCTLR and ACTLR*: the System and Auxiliary Control Registers used by the subject.
 - (iv) *SPSel*: Stack Pointer Select that the subject uses at exception level 1.
 - (v) *CONTEXTIDR*: the Context ID Register could be useful too.

²⁴[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.3, page 39 f.

²⁵This list is actually a compilation of the findings during the Student Research Study, the ARM reference manuals and application notes as well as some hypervisor examples like HASPOC and Xen. Furthermore, it was decided that the implementation and the list of registers will be extended and specified continuously during the project. Due to the scope limitations for this bachelor thesis, the statements to the registers only apply to subjects that use the AArch64 execution state.

(3) *Exception Related Registers:*

- (i) *SPSR*: the Saved Program Status Register for the current subject, that is automatically saved by the ARMv8-A exception handling process.
- (ii) *ELR*: the Exception Link Register that stores the return address of the subject if an exception has been taken.
- (iii) *ESR*: the Exception Syndrome Register with the description and error code of the exception used by the hypervisor to specify the state of the subject and to decide the next step in the scheduling process ²⁶.
- (iv) *VBAR*: the Exception Vector Table used by the subject.

(4) *MMU Related Registers:*

- (i) *TTBR0 and TTBR1*: the address specification for the page tables used by the subjects on **both** upper exception levels (i.e. exception level 1 and 0).
- (ii) *TCR*: the Translation Control Register used for the corresponding subject.
- (iii) *MAIR and AMAIR*: the Auxiliary and Memory Attribute Indirection Registers used for a specific subject. It is „implementation defined“ if the current core implementation supports the auxiliary register.

(5) *GIC Related Registers:*

- (i) *GIC Context Data*: depending on the current GIC version and the implementation the corresponding GIC context data registers have to be handled in the subject state.
- (ii) *IRQ Pending List*: as most GIC implementation only support a limited number of list entries in the hardware IRQ pending list, the hypervisor has to implement its own interrupt handling for all subjects running in the configuration.
- (iii) *ICH and ICC*: the Interrupt Control Registers for the subject.

(6) *Timer Related Registers:*

- (i) *Physical and Virtual Timer Context*: these context registers are closely related to the GIC implementation and therefore depend on the current GIC version.

²⁶ Even though this register is handled by the hypervisor during the scheduling process at the time of the occurrence of the exception, most other hypervisor implementation like Xen and HASPOC store it in the subject's context.

- (ii) *CNTKCTL*: independent of the chosen configuration, the Counter Timer Kernel Control Register has to be stored for each subject.

(7) *FPU Related Registers*:

- (i) *FP Context State*: contains the state of the Floating Point Unit (FPU) that has to be handled for all subject that use the FPU.
- (ii) *FPSR*: the Floating Point Status Register (cf. the SPSR register).
- (iii) *FPCR*: the Floating Point Control Register has to be stored if the hypervisor uses different FPU configurations for different subjects.

• *Hypervisor Related Registers*:

- (i) *HCR*: the hypervisor has to store the Hypervisor Control Register in the current subject's state because the Muen SK uses different trapping configurations for different subjects.
- (ii) *VMPIDR*: the Virtual Machine Processor ID Register contains some informations about a multi-core system that only has to be handled by the hypervisor if the CPU info part explained above does not cover the functionality provided with this „implementation defined“ register.
- (iii) *CNTHCTL*: the Timer Control Register has also to be stored by the hypervisor, as the Muen SK uses different timing configurations for the various subjects.
- (iv) *VTTBR*: the Virtual TTBR specifies the base address of the virtual page table used for the subject dependent Second Level Address Translation SLAT configurations.
- (v) *VTCR*: the Virtualization Table Control Register controls the behaviour of the SLAT configurations specified for each subject.
- (vi) *VMID*: the Virtual Machine ID concept is used to tag a translation as belonging to a particular virtual machine. For guest accesses, the Translation Lookaside Buffers (TLB) within the processor MMU can store a complete VA to IPA to PA translation in one entry. The VMID ensures that only the correct virtual machine can hit on a TLB entry and therefore removes the need to invalidate TLB entries when a context switch between guest operating systems is performed.

To maintain formal verification, the Muen SK implementation limits the exit and entry points of a subject to one point each in the source code. Even though, this principle has also been taken into account for the actual porting of the separation kernel to ARMv8-A architecture, the current implementation handles the subject entry and exit slightly different. While the current separation kernel first stores the subject's context and then calls the scheduler as shown in figure 3.15, the Muen SK directly exits into the scheduler and stores the subject context from within (cf. figure 3.16). To handle the subject exit and entry consistently, the current implementation has to be modified according to the Muen SK source code ²⁷

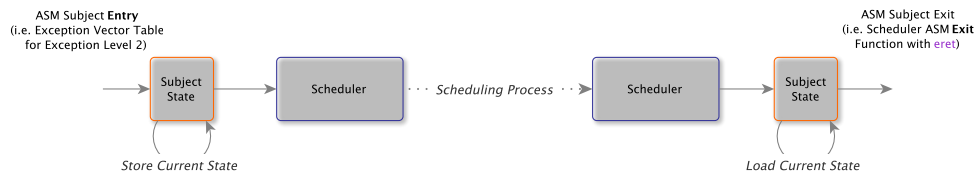


Figure 3.15: Muen SK current context switch

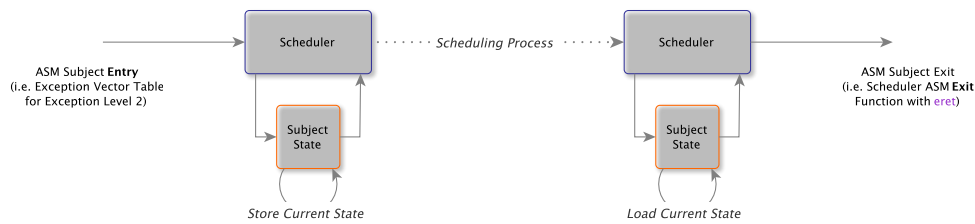


Figure 3.16: Muen SK correct context switch

3.3.5 Memory Management

Although the concept of the memory management unit and the corresponding page tables is relatively easy to understand ²⁸, it is probably the component where implementation errors occur most frequently. The ARMv8-A architecture in the AArch64 execution state supports a multi-level address translation with page granule sizes down to 4KB.

The form of the page table and the number of page table levels is basically defined by three factors, i.e. the input address size (Virtual Address Size), the output address size (Intermediate Physical Address or Physical Address) and the final page granule size. An overview for all possible combinations and the

²⁷ This could not yet be done during the project due to the time constraints of the bachelor thesis.

²⁸ [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 2, section 2.2.2, page 13 ff., and chapter 3, section 3.4.2, page 43 ff.

corresponding effects on the translation process can be found in the ARMv8-A Programmer's Guide²⁹. For a deeper insight into memory management configuration possibilities, it is referred to the theoretical part³⁰ and the examples³¹ in the ARMv8-A Architecture Reference Manual.

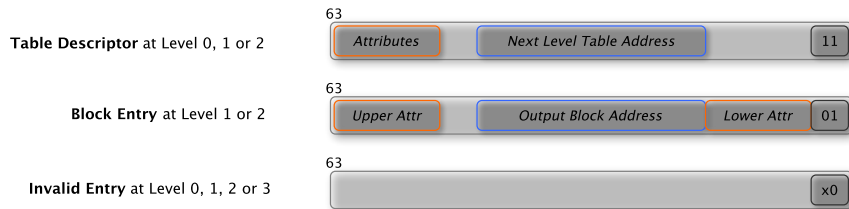


Figure 3.17: ARMv8-A address translation table entries

The number of entries in a single page table also depends on the current configuration. For the address translation configuration with 4KB page sizes according to the Muen SK requirements and 40-bit virtual and physical addresses used in this project, there are 512 entries per page table. As shown in figure 3.17, each page table entry for the AArch64 execution state can be assigned to one of the following three types:

- *Table Descriptor*: holds the address of a next level table, in which case memory can be further subdivided into smaller blocks.
- *Block Entry*: contains the address of a variable sized block of memory depending on the page table level of the entry.
- *Invalid Entry*: denote entries that are marked as fault or invalid and cause an according MMU exception.

As already mentioned, the ARMv8-A Virtualization Extension supports a Second Level Address Translation mechanism³². Although the principle of address translation is identical for both stages, it is important to note that the attributes for Stage One and Stage Two are **NOT** identical. While Stage One Translation sets the memory access attributes with the Memory Attribute Indirection Register (i.e. MAIR and AMAIR), Stage Two translations store the attribute values directly into the page tables according to a bit mask specification. Although the specifications for the two attribute values for Stage One and Stage Two look very similar, there are important differences to be considered (e.g. bit configurations are even reversed).

²⁹ [7] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.4.2, page 12-15 f.

³⁰ [6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2018, chapter D-5, page D5-2383 ff.

³¹ [6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2018, Appendix K-7, page K7-7283 ff.

³² [4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.4.3, page 45 f.

Probably the best way to explain the address translation mechanism defined by ARMv8-A architecture is the following example 3.18. It shows the address segmentation with the different level offsets for a 4KB page table granule size configuration with 48-bit virtual addresses mapped to 40-bit physical addresses. The example corresponds to the Subject One configuration, which can be found in the subjects folder of the MuenSK GPS project.

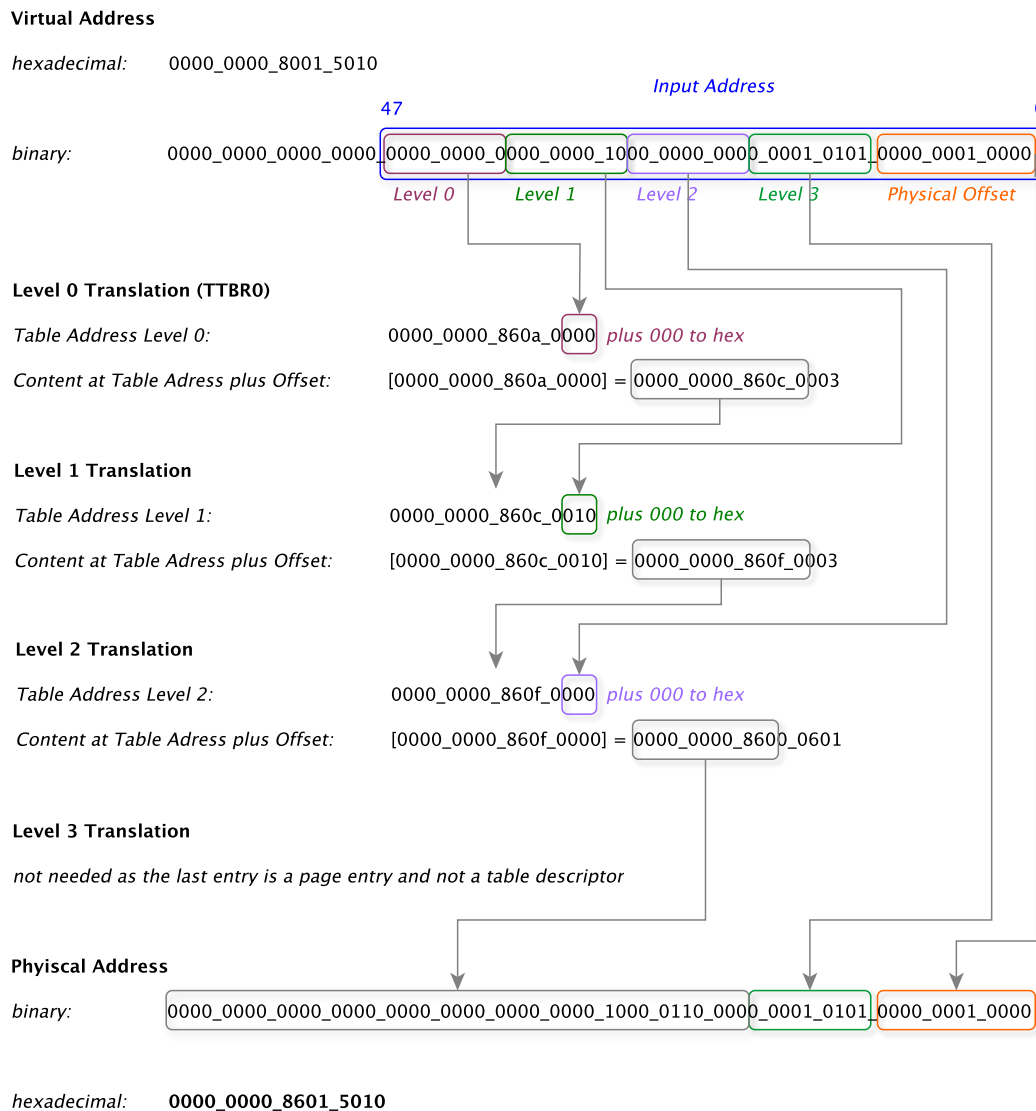


Figure 3.18: ARMv8-A address translation example with 4KB granule size, start level 0, 48 bit virtual, 40 bit physical address sizes (cf. direct mapping page tables subject one)

3.3.6 Exception Handling

Another important component in the context of the porting of the Muen SK to the ARMv8-A architecture is the Generic Interrupt Controller (GIC), which had to be qualified as a hardware requirement in the course of the Student Research Study³³. The NXP LS1012A SoC implements the GICv2 architecture, which basically consists of four components, i.e. the Distributor, the CPU Interface, the Virtual Control (corresponds to a virtualised distributor) and the Virtual CPU Interface. In order to route an exception correctly, **(a)** the according board interrupt mechanism has to be enabled, **(b)** the physical interrupt must then be handled by the distributor and forwarded to the correct CPU interface and **(c)** finally the hypervisor has to query the interrupt from the correctly configured CPU interface and process a virtual exception to the correct subject via the virtualization extension.

While the GIC interrupt mechanism without considering the virtualization extensions is very similar to the principles used in the embedded system area, the hypervisor has to implement all tasks related to the virtualization of exceptions itself. This especially means that **(a)** the Virtual Control Register has to be configured correctly for the respective subject, **(b)** the exceptions must be processed accordingly (e.g. software defined IRQ pending list) and **(c)** the hypervisor code has to ensure that the subject's page table map the CPU Interface Register correctly to the Virtual CPU Interface Register.

The current implementation of the separation kernel strictly follows the ARM GIC specifications as shown in figure 3.19. However, this design principle should not be used in the production code - the implementation rather shows the functionality of an appropriate tool for generating interrupt handling related code for different ARM GIC architectures, ARM GIC designs and SoC implementations.

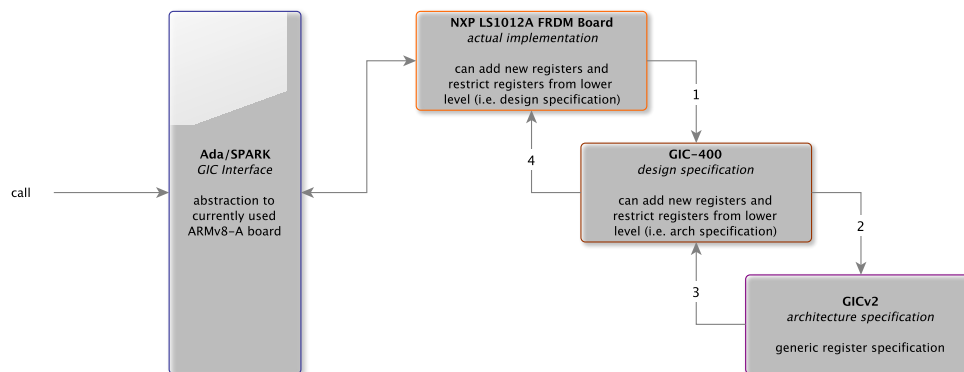


Figure 3.19: GIC implementation for NXP LS1012A FRDM Board

Due to the time constraints of this bachelor thesis, the virtualization extension of the GIC and thus also the ARM Generic Timer³⁴ could not be implemented. However, the current implementation demonstrates by an interrupt handling test with a software generated exception that both the configuration of the distributor and the CPU interface are currently working.

³³[4] Loosli. *Student Research Study, Muen on ARM - an Evaluation*. 2017, chapter 3, section 3.5.1, page 49.

³⁴[8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2018, chapter 10, page 10-1 ff.

4 Conclusion

The aim of this bachelor thesis was to implement the main building blocks of a separation kernel for the ARMv8-A architecture, leveraging in particular the recently introduced AArch64 Virtualization Extensions. This chapter is dedicated to a summarising conclusion with respect to the status of the development, the further development of the project as well as the integration of the Intel based Muen SK implementation.

4.1 Status of Development

First of all, it has to be mentioned that the findings from the Student Research Study could completely be confirmed with the experiences gained during this study. It could also be demonstrated that the porting of the Muen SK to the ARMv8-A architecture is - only restricted by a few requirements for the target platform - absolutely possible. The interest in the project shown by AdaCore and the clients of the codelabs GmbH even in the course of this bachelor thesis has been really motivating to demonstrate this capability.

With the exception of the ARM Generic Interrupt Controller Virtualization Extension and therefore also the ARM Generic Timer, the current source code of the separation kernel implements all the mechanisms required by the Student Research Study for the separation of currently two differently configured subjects. With one of this subjects running as guest system on the hypervisor, even an alternative, previously unknown to the author way to isolate a native subject without the usage of the Second Level Address Translation could be demonstrated.

4.2 Integration of the Muen SK

One of the next steps of porting the Muen SK to the ARMv8-A architecture is to integrate the hardware independent components of the Muen SK implementation into the source code of the current separation kernel. This raises a number of questions about how this can be done and what changes are involved. The two most important questions are discussed in this section.

The first of the two most fundamental questions from the author's point of view is closely related to the different processor architectures of Intel and ARM. Looking at the scheduler code of the Muen SK implementation, it can, for example, be observed that the hypervisor processes a subject exit in the procedure `Handle_Vmx_Exit` according to a `Basic_Exit_Reason` depending on the interrupt type. Due to the insights into the Intel interrupt handling mechanism gained during the Code Walk Through presented by Adrian-Ken Rüegsegger and Reto Bürki, the impression came up that the Intel architecture provides very different configuration possibilities and interrupt types compared to the ARMv8-A architecture. Therefore, it has to be thoroughly examined if a sensible common layer of abstraction can be found

and - if this can be confirmed - whether the corresponding changes with respect to time and financial constraints can be qualified as appropriate.

The second fundamental question, that arises from the discussion of the first one, is whether the two architectures should be united in a common repository or not. If one chooses the solution of a common level of abstraction, it might also make sense to combine the currently separated repositories in a common project. However, it should be considered that with regard to the formal verification and the complexity of the code it might also make sense to continue the two projects separately even if some of the tools share a common level of abstraction.

4.3 Further Development

The following list provides a chronologically ordered overview of the possible steps for the further development of the project:

- (1) *ARM GIC Virtualization Extension*: In a first step, the GIC Virtualization Extension should be configured and tested according to the requirements of the Muen SK. As the virtualisation related registers are already implemented in the current separation kernel source code, the focus has to be on the configuration and the exception handling by the hypervisor.
- (2) *ARM Generic Timer*: After a successful implementation of the GIC Virtualization Extension, a preemptive timer has to be added to the source code. With this last component required by the Muen SK, the final separation kernel can serve as the base for the integration of the hardware independent Muen SK source code.
- (3) *Muen SK Integration*: As discussed in section 4.2, there have to be taken several decisions with respect to a common layer of abstraction and the integration of the ARMv8-A related code into the current Muen SK repository. This step demands a particularly thorough examination and consideration.
- (4) *Muen SK Tools*: Especially for the generation of the page tables needed by the MMU and the interrupt handling related code, it should be considered to implement additional tools analogous to the already existent tools for the Intel based Muen SK implementation.
- (5) *Muen SK Subjects*: The Muen SK provides several native subjects and adjusted VM binaries as guest systems. Hence, the ARMv8-A implementation should follow this principle as well.

5 Epilogue

Using a Segger J-Link hardware debug probe device, the on-chip debugger software OpenOCD and the AdaCore toolchain including their integrated development environment, essential parts of a separation kernel have been implemented in Ada in the course of the project. With this basic separation kernel prototype and its two differently configured subjects, it could be demonstrated that all requirements with respect to the porting of the Muen SK to the ARMv8-A architecture can be met applying the already during the Student Research Study examined ARMv8-A architecture design principles.

Finally, I would like to thank Professor Dr. Andreas Steffen and the two supervisors Adrian-Ken Rüegsegger and Reto Bürki for their great support. This project, that covered several studies of the Bachelor of Science in Computer Science program at the University of Applied Sciences Rapperswil (HSR), would not have been possible without their technical and organisational help. I am really looking forward to the further development and porting of the Muen SK to the ARMv8-A architecture as part of the offered employment!

Appendix

A Project Task Description

Bachelor Thesis
autumn semester 2018

Definition of Task

Muen on ARM

version: 0.00, date: December 21, 2018

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rüeggsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Introduction

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

Based on the findings of the former student research study „Muen On ARM - an Evaluation“ written by the author of this Bachelor Thesis, the objective of this study is to develop a minimal Separation Kernel prototype for the ARMv8-A architecture based on the Muen SK and leveraging the AArch64 Virtualization Extensions introduced with the latest ARM architecture. The target hardware platform for this Bachelor Thesis is the NXP LS1012A Freedom Evaluation Board with an ARMv8 Cortex-A53 CPU and the programming language is Ada/SPARK 2014.

Objectives

(i) Prototypical implementation of main Separation Kernel building blocks

- System initialization
- Exception & interrupt handling
- Definition and switching of AArch64 subject state
- Subject preemption mechanism
- Serial debug driver

(ii) Documentation

(iii) Optional:

- Scheduling of multiple subjects
- AArch64 pagetable generation tool

B List of Related Documents

- Glossary
- Management Summary
- Student Research Study

C U-Boot Environment Setup

```
autoload=no
baudrate=115200
bootargs=console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0
x21c0500
bootcmd=pfe stop; sf probe 0:0; sf read $kernel_load $kernel_start
$kernel_size && bootm $kernel_load
bootdelay=-1
bootmuen=tftpboot 0x96000000 muensk.uImage; bootm 0x96000000
console=ttyAMA0,38400n8
eth1addr=00:04:9f:05:31:89
ethact=pfe_eth0
ethaddr=00:04:9f:05:31:88
fdt_high=0xffffffffffffffff
fdtcontroladdr=9fc85170
hwconfig=fsl_dds:bank_intlv=auto
initrd_high=0xffffffffffffffff
ipaddr=192.168.12.12
kernel_addr=0x100000
kernel_load=0x96000000
kernel_size=0x2800000
kernel_start=0xa00000
loadaddr=0x80100000
ramdisk_addr=0x800000
ramdisk_size=0x2000000
serverip=192.168.12.1
stderr=serial
stdin=serial
stdout=serial
verify=no
```


D GDB Initialisation Script

```

##
# @file .gdbinit GDB configuration file
#
#       @project      MuenOnARM
#       @interface    OpenOCD GDB Server
#       @target       NXP FRDM-LS1012A
#
#       @usage        First, this file has to be placed in the root directory
#                     of the project or the directory, from which the GDB
#                     Debugger is started. Then, a .gdbinit file has to be
#                     created in the home directory of the current user with
#                     the following line of code (alternatives can be found
#                     in the official GDB User Guide of the toolchain):
#
#                     add-auto-load-safe-path /path/to/project/.gdbinit
#
##

##
# @setup connect to OpenOCD
##
target remote localhost:3333

##
# @setup reset target
##
monitor reset

##
# @setup wait for target external RAM to be initialized
##
shell sleep 1

##
# @setup halt remote target
##
monitor halt

##
# @setup report breakpoints and watchpoints to gdb
##
set remote hardware-breakpoint-limit 6
set remote hardware-watchpoint-limit 4

##
# @load if the executable file is loaded manually the following defined
#       procedures can be typed in a started gdb session.
##
define load_exec
file bin/muensk
load
end

define load_with_subjects
file ../muensubjects/muensubjectone/bin/muensubjectone
load
add-symbol-file ../muensubjects/muensubjectone/bin/muensubjectone 0x86000000

```

```
file ../muensubjects/muensubjecttwo/bin/muensubjecttwo
load
add-symbol-file ../muensubjects/muensubjecttwo/bin/muensubjecttwo 0x8a000000
file bin/muensk
load
add-symbol-file bin/muensk 0x83800000
end

define load_debug_asm
file bin/muensk
load
break _start
continue
end

##
# @post_setup to display different commands as graphs in GPS
##
define hook-quit
monitor reset
end

define hook-detach
monitor reset
end

define hook-disconnect
monitor reset
end

##
# @ide to display different commands as graphs in GPS execute
# the following commands inside the GPS debugger console
#
#     graph display 'monitor aarch64 state_info'
#     graph display 'monitor aarch64 current_el'
#     graph display 'monitor reg'
#
##

## EOF #
```

E Codelabs Contributors Agreement

Codelabs Contributors Agreement

Please print this form, read it carefully, fill it out, sign it and then either

- scan and email it to legal@codelabs.ch
- send it by post to codelabs GmbH, Vadianstrasse 41, 9000 St. Gallen, Switzerland

Your contact details

Full name

Street address

Zip code, city and country

Email address

Terms and Conditions

You hereby grant codelabs GmbH, Vadianstrasse 41, 9000 St. Gallen, Switzerland, a license to use your contributions to a project managed by codelabs, under the following conditions.

The *codelabs projects*, as referred to in this agreement, means all software products that codelabs GmbH has, at the date of signing, published or will publish in the future under an open-source license, for example the GNU General Public License v3.0 (GPLv3). This includes, but is not limited to, the Muen Separation Kernel. *Contributions*, as referred to in this agreement, mean any past or future material such as source or binary code, artwork of any media, documentation, correspondence in mail, or similar material as usually found in relation to software. This agreement covers such contributions only as far as they are related to *codelabs projects*.

You hereby give codelabs GmbH a world-wide, perpetual, irrevocable, royalty-free, yet non-exclusive license to use your contributions to *codelabs projects* in any way that codelabs GmbH may see fit, including for commercial purposes. This includes, but is not limited to, the right to copy, translate, relicense, sublicense, modify, use, make available or public, sell, offer to sell, rent, lease, lend or otherwise distribute your contributions or modifications thereof, as well as any ideas contained therein that may be covered by patents under applicable law.

You formally release codelabs GmbH from its obligation to name you as the author of your contribution and otherwise respect your moral rights, as you are aware that there

are no such obligations in the GPL either and codelabs GmbH may possibly want to redistribute your contributions under that license, and you permit codelabs GmbH to act as the sole author and copyright holder of its product even with your contributions.

The codelabs GmbH will, however, make any reasonable effort to give you proper credit for your contributions in the products' documentation materials as well as websites. The codelabs GmbH will duly examine your contributions, but you understand that codelabs GmbH is under no obligation to make use of them as described above. You certify and warrant that your contributions to codelabs GmbH's products do not violate the intellectual property rights of third parties and that you are legally entitled to grant codelabs GmbH all of the rights listed above.

Place, Date Signature

Bibliography

- [1] John Barnes. *Programming in Ada2012*. Cambridge (UK): Cambridge University Press, 2018. ISBN: 978-1-107-42481-4.
- [2] Reto Buerki and Adrian-Ken Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2013. URL: <https://muen.codelabs.ch>.
- [3] James A. Langbridge. *Professional Embedded ARM Development*. Indianapolis, Indiana (USA): John Wiley & Sons Inc., 2014. ISBN: 978-1-118-78894-3.
- [4] David Loosli. *Student Research Study, Muen on ARM - an Evaluation*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2017.
- [5] n.a. *AArch64 Virtualization*. version 1.0. Cambridge (England): ARM Limited, 2017.
- [6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. version D.a. Cambridge (England): ARM Limited, 2018. URL: <http://www.arm.com>.
- [7] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. version 1.0. Cambridge (England): ARM Limited, 2015. URL: <http://www.arm.com>.
- [8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. revision r0p4 J. Cambridge (England): ARM Limited, 2018. URL: <http://www.arm.com>.
- [9] n.a. *ARM Generic Interrupt Controller, Architecture Specification*. version 2.0, B.b. Cambridge (England): ARM Limited, 2013.
- [10] n.a. *ARM System Memory Management Unit, Architecture Specification*. version 3.0 and version 3.1. Cambridge (England): ARM Limited, 2017. URL: <http://www.arm.com>.
- [11] n.a. *CoreLink GIC-400 Generic Interrupt, Technical Reference Manual*. revision r0p1, B. Cambridge (England): ARM Limited, 2012.
- [12] n.a. *CoreSight Components, Technical Reference Manual*. version H. Cambridge (England): ARM Limited, 2009. URL: <http://www.arm.com>.
- [13] n.a. *J-Link / J-Trace User Guide*. Revision 3, SW 6.34 UM08001. Hilden (Germany): SEGGER Microcontroller GmbH, 2018. URL: <https://www.segger.com/products/debug-probes/j-link>.
- [14] n.a. *QorIQ LS1012A Application Note, PBL Configuration using QCVS*. Revision 1, 2/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [15] n.a. *QorIQ LS1012A Board Reference Manual*. Revision 3, 12/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [16] n.a. *QorIQ LS1012A Data Sheet*. Revision 1, 01/2018. Eindhoven (Netherlands): NXP Semiconductors, 2018.

- [17] n.a. *QorIQ LS1012A Getting Started Guide*. Revision 3, 12/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [18] n.a. *QorIQ LS1012A Reference Manual*. Revision 1, 01/2018. Eindhoven (Netherlands): NXP Semiconductors, 2018.
- [19] n.a. *QorIQ LS1012A Security (SEC) Reference Manual*. Revision 1, 07/2017. Eindhoven (Netherlands): NXP Semiconductors, 2017.
- [20] Spencer Oliver et al. *Open On-Chip Debugger: OpenOCD Developers's Guide*. release 0.10.0. Augsburg (Germany): The OpenOCD Project, University of Applied Sciences FH-Augsburg, 2017. URL: <http://openocd.org/doc-release/doxygen/index.html>.
- [21] Spencer Oliver et al. *Open On-Chip Debugger: OpenOCD User's Guide*. release 0.10.0. Augsburg (Germany): The OpenOCD Project, University of Applied Sciences FH-Augsburg, 2017.
- [22] John Rushby. "Design and Verification of Secure Systems". In: *ACM Operating Systems Review* 15.5 (1981), pp. 12–21.

List of Figures

2.1	ARMv8-A Exception Levels in AArch64 with Hypervisor Level	14
2.2	NXP LS1012A FRDM evaluation board	17
2.3	NXP LS1012A FRDM schematic	18
2.4	TFTP Server Setup	21
3.1	GPS IDE overview	23
3.2	J-Link Debug Probe setup including ARM JTAG pinout	25
3.3	GPS IDE OpenOCD settings project	26
3.4	GPS IDE OpenOCD integration	26
3.5	GPS IDE GDB integration	27
3.6	GPS IDE GDB load binary into DDR RAM	28
3.7	GPS IDE GDB execute binary from DDR RAM	28
3.8	Kermit Script serial load process	29
3.9	Kermit Script kernel execution start	30
3.10	GPS IDE runtime settings project	31
3.11	GPS IDE makefile settings project	32
3.12	Muen SK separation kernel code structure	33
3.13	Subject One memory map	39
3.14	Subject Two memory map	39
3.15	Muen SK current context switch	43
3.16	Muen SK correct context switch	43
3.17	ARMv8-A address translation table entries	44
3.18	ARMv8-A address translation example with 4KB granule size, start level 0, 48 bit virtual, 40 bit physical address sizes (cf. direct mapping page tables subject one)	45
3.19	GIC implementation for NXP LS1012A FRDM Board	46

List of Tables

2.1	board evaluation process, final result matrix	17
2.2	NXP LS1012A QSPI Flash Layout	20
3.1	Development Environment Overview	22

Bachelor Thesis
autumn semester 2018

Glossary

Muen on ARM

version: 1.0, date: December 21, 2018

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rüegsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Change History

date	version	change	author
Dec 10, 2018	0.1	prepared template, setup basic version	David Loosli
Dec 16, 2010	0.2	added abbreviations, bibliography and glossary entries	David Loosli
Dec 21, 2018	1.0	final changes before hand in	David Loosli

Acronyms

A

ACM Association for Computing Machinery

ALU Arithmetic Logic Unit

APIC Advanced Programmable Interrupt Controller

ARM Advanced RISC Machines

ASID Address Space Identifier

AVT Study Administration Tool (dt. Arbeitsverwaltungstool)

B

BCM Broadcom

BPC Branch Prediction Caches

C

cf. confer

CISC Complex Instruction Set Computing

CPU Central Processing Unit

CU Control Unit

D

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DS Development Studio

E

e.g. for example (lat. exempli gratia)

ECTS European Credit Transfer and Accumulation System

EL Exception Level

EPT Extended Page Tables

F

FASMARM Flat Assembler ARM

FIQ Fast Interrupt Request

FPU Floating Point Unit

FVP Fixed Virtual Platform (an ARM simulation tool)

G

GB Gigabyte

GCC GNU Compiler Collection

GIC Generic Interrupt Controller

GNAT GNU NYU Ada Translator

GPIO General Purpose Input/Output

GPS GNAT Programming Studio

GPU Graphics Processing Unit

H

HDD Hard Disk Drives

HDMI High Definition Multimedia Interface

HSR University of Applied Sciences Rapperswil

I

i.e. that is (lat. id est)

I/O Input / Output

ID Identifier

IDE Integrated Development Environment

incl. including

IOMMU I/O Memory Management Unit

IPI Inter-processor Interrupt

IRQ Interrupt Request

ISR Interruption Service Routine

J

JTAG Joint Test Action Group (interface)

K

KB Kilobyte

L

LAPIC Local Advanced Programmable Interrupt Controller

LPAE Large Physical Address Extension

LPI Locality Specific Peripheral Interrupt

M

MB Megabyte

MCU Broadcom

MMU Memory Management Unit

N

NMI Non-maskable Interrupt

NYU New York University

O

OS Operating System

P

PIC Programmable Interrupt Controller

PIPT Physically Indexed Physically Tagged

PIT Programmable Interval Timer

R

RAM Random Access Memory

REQ Requirement

RISC Reduced Instruction Set Computing

ROM Read Only Memory

RPi3 Raspberry Pi 3 (hardware)

RTS Runtime System

Rx Receive (line)

S

SD Card Secure Digital Memory Card

SError System Error

SIMD Single Instruction Multiple Data

SK Separation Kernel

SLAT Second Level Address Translation

SMMU System Memory Management Unit

SMT Simultaneous Multithreading

SoC System on Chip

SRAM Static Random Access Memory

SSD Solid State Disks

T

TBL Translation Lookaside Buffer

TTL Transistor Transistor Logic

Tx Transmit (line)

U

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

V

VE ARM Virtualization Extension

VIPT Virtually Indexed Physically Tagged

VM Virtual Machine

VMCS Virtual Machine Control Structure (Intel register)

VMM Virtual Machine Monitor

VT Virtualization Technology

VT-d Intel Virtualization Technology for Directed I/O

VT-x Intel Virtualization Technology x86

W

WTFPL Do What The Fuck You Want To Public License

Z

ZFP Zero Footprint Runtime

Glossary

A

AArch32 AArch32 denotes the 32-bit execution state of the ARMv8-A architecture.

AArch64 AArch64 denotes the 64-bit execution state of the ARMv8-A architecture.

ARM On the one hand, this term is used as an abbreviation for the ARM Holdings company, a British multinational semiconductor and software design company - on the other hand ARM denotes a family of reduced instruction set computing (RISC) architectures for computer processors. In all documents of this study, the term is used in the second sense, unless explicitly otherwise specified.

ARMv8-A The Armv8-A architecture is the latest generation ARM RISC architecture targeted at the applications 'A' profile. It introduces the ability to use 64-bit and 32-bit execution states, known as AArch64 and AArch32 respectively. The architecture 'A' profile, compared against the profiles 'M' and 'R', targets high performance markets such as mobile and (industrial) enterprise. In this study, the terms ARMv8 and ARMv8-A are used interchangeably.

B

Bamboo Bamboo is a continuous integration and continuous deployment server developed by Atlassian. In this study, the software is only in use as a build server.

Bitbucket Bitbucket denotes proprietary, web-based integration software, developed by Atlassian. It is used for source code and development projects that use either Mercurial or Git version control systems.

C

Confluence Confluence is a team collaboration software, written in Java and mainly used in corporate environments. It is also developed and marketed by Atlassian.

crt0 (file) crt0 (also known as c0) contains a set of execution startup routines and is linked into a C program that performs any initialization work required before calling the program's main function. It is often written in assembly language and automatically included by the linker into every executable file it builds.

D

Do What The Fuck You Want To Public License (WTFPL) The WTFPL (Do What the Fuck You Want To Public License) is a permissive license most commonly used as a free software license. It allows redistribution and modification of the software under any terms.

F

Fixed Virtual Platforms (FVP) Fixed Virtual Platforms (FVP) is a simulation bundle for ARM processors that allows the development of software for ARM architectures without the need for actual hardware, developed by the ARM company. The functional behaviour of a model is equivalent to real hardware.

Flat Assembler ARM (FASMARM) Flat Assembler ARM (FASMARM) is a cross assembler for different ARM architectures that runs on x86 host operating systems.

G

General Purpose Input/Output (GPIO) General-purpose input/output (GPIO) is a generic pin on an integrated circuit or computer board whose behavior - including whether it is an input or output pins controllable by the user at run time.

Git Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files.

J

JIRA Jira is a proprietary issue tracking product, developed by Atlassian. It provides bug tracking, issue tracking, and project management functions. In this project, it is primarily used for providing a communication platform between the author and the supervisors (state of affairs) as well as for the mandatory time tracking.

Joint Test Action Group (JTAG) The Joint Test Action Group (JTAG) is an electronics industry association formed and is often used as a synonym for their standardized debug interface. JTAG specifies the use of a dedicated debug port implementing a serial communication interface. The interface connects to an on-chip test access port (TAP) that implements a stateful protocol to access a set of test registers that present chip logic levels and device capabilities of various parts.

L

LaTeX LaTeX is a document preparation system, widely used in academia for the communication and publication of scientific documents.

R

Raspberry Pi 3 (RPi3) The Raspberry Pi 3 is the latest version of a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries.

S

System on Chip (SoC) A system on a chip (SoC) is an integrated circuit (IC) that integrates a microcontroller or microprocessor with advanced peripherals like graphics processing unit (GPU), WiFi module or coprocessors. Typical applications can be found in the area of embedded systems. *Analogy:* A SoC corresponds to a desktop motherboard (also known as mainboard) with a separately bought and installed CPU.

U

Universal Asynchronous Receiver-Transmitter (UART) A universal asynchronous receiver-transmitter (UART) is a computer hardware interface for asynchronous serial communication in which the data format and transmission speeds are configurable. UART communicates over the two lines Rx for Receive and Tx for Transmit.

V

VT-d VT-d represents Intel's technology for I/O MMU virtualization on the x86 platform. An input/output memory management unit (IOMMU) allows guest virtual machines to directly use peripheral devices, such as Ethernet, accelerated graphics cards, and hard-drive controllers, through DMA and interrupt remapping. This is sometimes called PCI passthrough.

VT-x VT-x represents Intel's technology for virtualization on the x86 platform. In this context, the term virtualization encompasses different concepts for allowing a hypervisor to use hardware functionality to fulfil its tasks - i.e. entering and exiting a virtual execution mode, using virtual page tables and many other concepts realised in hardware.

W

WorklogPRO WorklogPRO is a time tracking and reporting plugin for the JIRA issue tracking software.

Y

yEd yEd is a free general-purpose diagramming program with a multi-document interface. It is a cross-platform application written in Java that runs on Windows, Linux, Mac OS, and other platforms that support the Java Virtual Machine. yEd can be used to draw many different types of diagrams, including flowcharts, network diagrams, UMLs, BPMN, mind maps, organization charts, and entity-relationship diagrams.

Bibliography

- [1] John Barnes. *Programming in Ada2012*. Cambridge (UK): Cambridge University Press, 2018. ISBN: 978-1-107-42481-4.
- [2] Reto Buerki and Adrian-Ken Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2013. URL: <https://muen.codelabs.ch>.
- [3] James A. Langbridge. *Professional Embedded ARM Development*. Indianapolis, Indiana (USA): John Wiley & Sons Inc., 2014. ISBN: 978-1-118-78894-3.
- [4] David Loosli. *Student Research Study, Muen on ARM - an Evaluation*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2017.
- [5] n.a. *AArch64 Virtualization*. version 1.0. Cambridge (England): ARM Limited, 2017.
- [6] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. version D.a. Cambridge (England): ARM Limited, 2018. URL: <http://www.arm.com>.
- [7] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. version 1.0. Cambridge (England): ARM Limited, 2015. URL: <http://www.arm.com>.
- [8] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. revision r0p4 J. Cambridge (England): ARM Limited, 2018. URL: <http://www.arm.com>.
- [9] n.a. *ARM Generic Interrupt Controller, Architecture Specification*. version 2.0, B.b. Cambridge (England): ARM Limited, 2013.
- [10] n.a. *ARM System Memory Management Unit, Architecture Specification*. version 3.0 and version 3.1. Cambridge (England): ARM Limited, 2017. URL: <http://www.arm.com>.
- [11] n.a. *CoreLink GIC-400 Generic Interrupt, Technical Reference Manual*. revision r0p1, B. Cambridge (England): ARM Limited, 2012.
- [12] n.a. *CoreSight Components, Technical Reference Manual*. version H. Cambridge (England): ARM Limited, 2009. URL: <http://www.arm.com>.
- [13] n.a. *J-Link / J-Trace User Guide*. Revision 3, SW 6.34 UM08001. Hilden (Germany): SEGGER Microcontroller GmbH, 2018. URL: <https://www.segger.com/products/debug-probes/j-link>.
- [14] n.a. *QorIQ LS1012A Application Note, PBL Configuration using QCVS*. Revision 1, 2/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [15] n.a. *QorIQ LS1012A Board Reference Manual*. Revision 3, 12/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [16] n.a. *QorIQ LS1012A Data Sheet*. Revision 1, 01/2018. Eindhoven (Netherlands): NXP Semiconductors, 2018.

- [17] n.a. *QorIQ LS1012A Getting Started Guide*. Revision 3, 12/2016. Eindhoven (Netherlands): NXP Semiconductors, 2016.
- [18] n.a. *QorIQ LS1012A Reference Manual*. Revision 1, 01/2018. Eindhoven (Netherlands): NXP Semiconductors, 2018.
- [19] n.a. *QorIQ LS1012A Security (SEC) Reference Manual*. Revision 1, 07/2017. Eindhoven (Netherlands): NXP Semiconductors, 2017.
- [20] Spencer Oliver et al. *Open On-Chip Debugger: OpenOCD Developers's Guide*. release 0.10.0. Augsburg (Germany): The OpenOCD Project, University of Applied Sciences FH-Augsburg, 2017. URL: <http://openocd.org/doc-release/doxygen/index.html>.
- [21] Spencer Oliver et al. *Open On-Chip Debugger: OpenOCD User's Guide*. release 0.10.0. Augsburg (Germany): The OpenOCD Project, University of Applied Sciences FH-Augsburg, 2017.
- [22] John Rushby. "Design and Verification of Secure Systems". In: *ACM Operating Systems Review* 15.5 (1981), pp. 12–21.

Bachelor Thesis
Herbstsemester 2018

Projektorganisation

Muen on ARM

Version: 1.00, Datum: 21. Dezember 2018

Betreuer:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rügsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, Student
BSc Informatik
HSR Rapperswil

Änderungshistorie

Datum	Version	Bezeichnung	Autor
18. Dezember 2018	0.1	Erstellen Vorlage und Struktur	David Loosli
20. Dezember 2018	0.2	Erarbeiten Einleitung bis Anhang	David Loosli
21. Dezember 2018	1.0	letzter Nachtrag vor Abgabe	David Loosli

Inhaltsverzeichnis

Änderungshistorie	2
1 Einleitung	4
2 Organisation	5
2.1 Zeiterfassung	5
2.2 Projektplanung	6
2.3 Besprechungen	6
3 Administrationstools	7
3.1 Jira	7
3.2 Codelabs Git Repository	9
3.3 Backup	9
Anhang	10
A Wochenauszüge Worklog Pro (Jira)	10
B Protokolle	15

1 Einleitung

Dieses Dokument soll einen Überblick über die während der Bachelor Arbeit berücksichtigten organisatorischen Richtlinien und die verwendeten Administrationstools verschaffen. Im Gegensatz zur Studienarbeit wurden die Projektorganisationstools jedoch stark eingeschränkt, um den Administrationsaufwand möglichst zugunsten der inhaltlichen Tätigkeiten zu verringern. Im ersten Teil in Kapitel 2 werden die im während des Projektes festgehaltenen organisatorischen und zeitlichen Zielsetzungen dem tatsächlichen Verlauf des Projektes gegenübergestellt. Die für das Projekt verwendeten Administrationstools werden in Kapitel 3 vorgestellt.

2 Organisation

2.1 Zeiterfassung

Für das Projekt standen insgesamt 360 Arbeitsstunden zur Verfügung mit einem Budget von rund 26 Stunden pro Woche.

Auf die Bachelor Arbeit wurden während 14 Wochen 586.5 Stunden verbucht und somit die eigentliche Kostenschätzung um rund zwei Drittel überschritten. Diese Mehrkosten entstanden einerseits aufgrund des Umfangs des Projektes und andererseits aufgrund der Einreichung der Bachelor Thesis als Einzelarbeit. Die detaillierten Wochenauszüge des Jira Plugins Worklog Pro finden sich im Anhang [A](#) dieses Dokumentes und ein kompletter CSV Export auf dem ebenfalls beiliegenden USB Datenträger.

Semesterwoche	Referenz	Geleistete Stunden in [h]
Vorbereitung	Anhang A , Abbildung 3.4	13.25
Woche 1	Anhang A , Abbildung 3.5	33.00
Woche 2	Anhang A , Abbildung 3.6	61.25
Woche 3	Anhang A , Abbildung 3.7	38.25
Woche 4	Anhang A , Abbildung 3.8	46.50
Woche 5	Anhang A , Abbildung 3.9	58.75
Woche 6	Anhang A , Abbildung 3.10	27.75
Woche 7	Anhang A , Abbildung 3.11	40.25
Woche 8	Anhang A , Abbildung 3.12	40.25
Woche 9	Anhang A , Abbildung 3.13	54.00
Woche 10	Anhang A , Abbildung 3.14	45.75
Woche 11	Anhang A , Abbildung 3.15	29.25
Woche 12	Anhang A , Abbildung 3.16	17.75
Woche 13	Anhang A , Abbildung 3.17	33.50
Woche 14	Anhang A , Abbildung 3.18	47.00
Total		586.50

Tabelle 2.1: Übersicht geleistete Arbeitsstunden pro Semesterwoche

2.2 Projektplanung

Den Erfahrungen aus der Studienarbeit entsprechend wurde die Planung nach Rücksprache mit allen Beteiligten aufgrund der nur sehr schwer abschätzbaren Entwicklungszeiten zu den einzelnen Arbeitspaketen jeweils von Sprint zu Sprint angepasst. Ein Sprint dauerte jeweils zwei Wochen - der erste diente der Evaluation und der Vorbereitung, die nächsten fünf Sprints können der Construction zugeordnet werden und im letzten Sprint wurden die Dokumente zur Bachelor Thesis erarbeitet.

Während die Zeiten zu den Arbeitspaketen zu Beginn des Projektes relativ schlecht geschätzt wurden, so verbesserte sich dies bis zum Schluss augenscheinlich. Mit Blick auf die Organisation der Arbeitspakete hat sich das Erstellen von sog. Epics wieder sehr gelohnt.

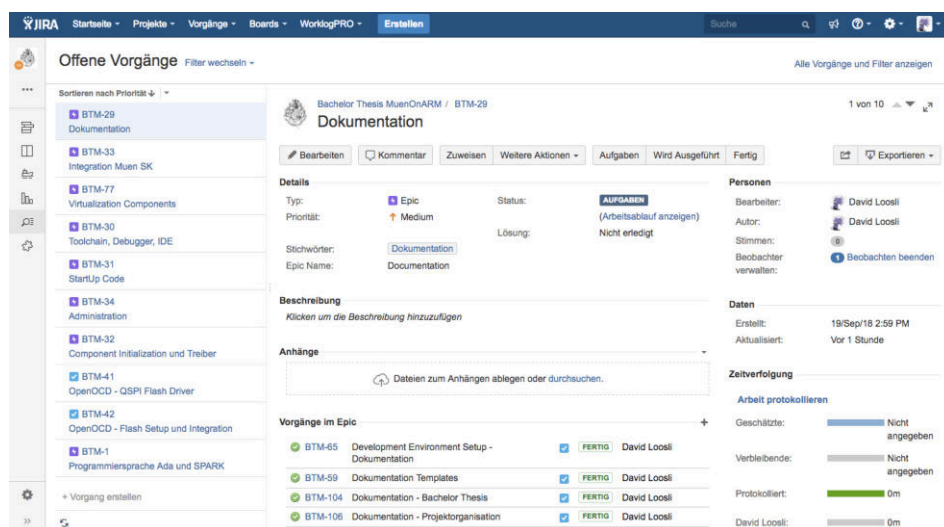


Abbildung 2.1: Jira Epic Auszug

2.3 Besprechungen

Die wöchentlichen Besprechungen zwischen dem Betreuer respektive den Betreuern und dem Autor der Bachelor Thesis fanden bis auf die letzte Woche ordnungsgemäss statt und wurden entsprechend der Projektvorgaben protokolliert. Sämtliche Protokolle finden sich im Anhang B oder können alternativ auf dem USB Datenträger eingesehen werden.

3 Administrationstools

Im Gegensatz zur Studienarbeit konnten die Administrationstools aus Infrastrukturgründen nicht auf einem eigenen Server aufgesetzt und verwaltet werden. Neben dem auf einer von der Hochschule für Technik Rapperswil (HSR) zur Verfügung gestellten virtuellen Maschine installierten Software Entwicklungstool Jira von Atlassian konnte über den codelabs Server der beiden Betreuer zusätzlich ein Git Repository verwendet werden. Sowohl das Aufsetzen der virtuellen Maschine inklusive der Installation der sich schon während der Studienarbeit bewährten Jira Software als auch die Konfiguration des Git Repositories verliefen ohne Probleme. Die Administrationstools sowie die für die Betreuung erstellten Zugänge werden noch bis zu Beginn des kommenden Semesters aufgeschaltet bleiben.

3.1 Jira

Das Software Planungstool Jira mit dem Plugin Worklog Pro wurde einerseits zur Organisation der einzelnen Arbeiten und andererseits zur Schätzung und Nachverfolgung der benötigten Arbeitszeit verwendet. Entsprechend der Erfahrung wurde auch für die Bachelor Thesis ein erweitertes Dashboard erstellt.

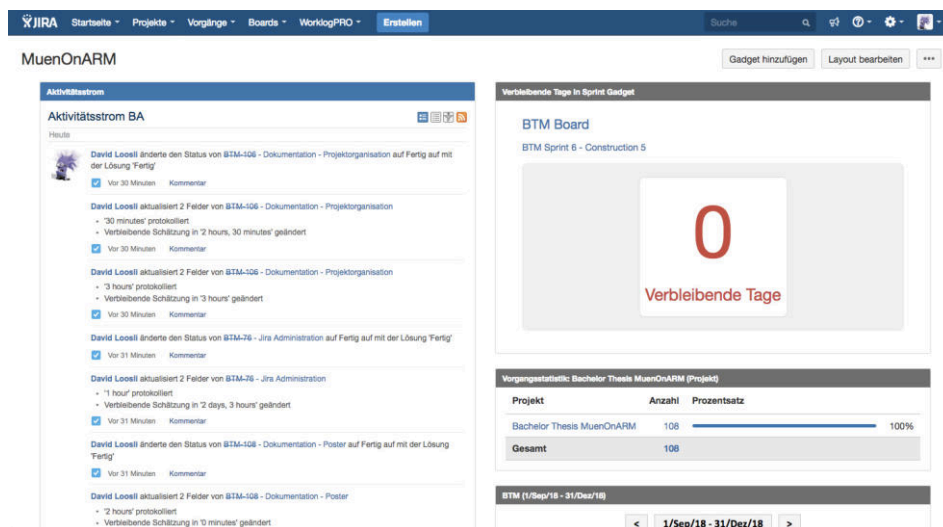


Abbildung 3.1: Jira Dashboard Bachelor Thesis

Den Projektplanungen entsprechend wurden jeweils zweiwöchige Sprints im Jira Softwareentwicklungstool definiert und die jeweiligen Arbeitspakete dazu vollständig erfasst und geschätzt. Der letzte Sprint wurde mit heutigem Datum ein Tag vor der Abgabe der Bachelor Arbeit abgeschlossen.

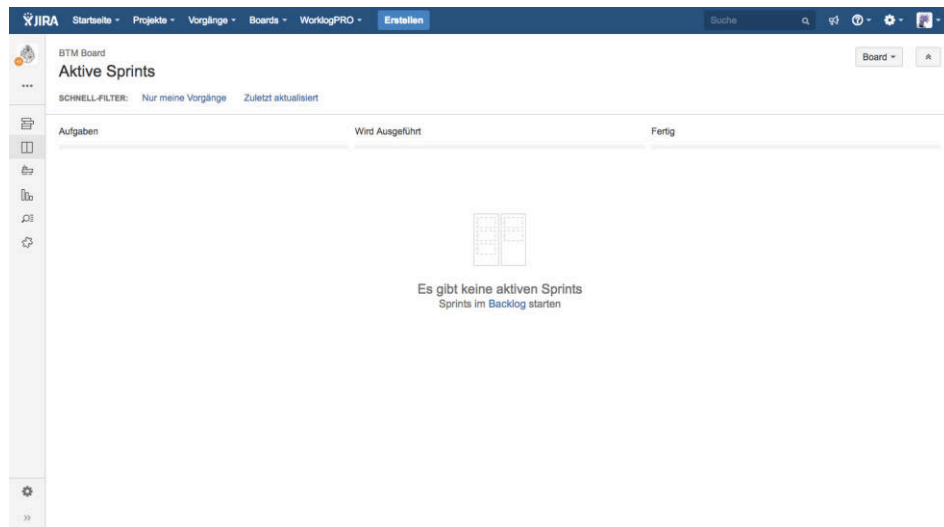


Abbildung 3.2: Jira Sprint Ansicht zum Ende der Bachelor Arbeit

Der Backlog des Projektes konnte aus zeitlichen Gründen nicht vollständig abgearbeitet werden. Bei den nicht erledigten Arbeiten handelt es sich um die beiden OpenOCD Treiber für den QSPI Flash des NXP LS1012A FRDM Board. Die Implementation dieser beiden Komponenten würde das Beschreiben des Flash Speichers ermöglichen, war für den Entwicklungsprozess dieses Projekts aufgrund des direkten Ladens in den RAM Speicher des Evaluationsboards über den GDB Debugger nicht von Bedeutung.

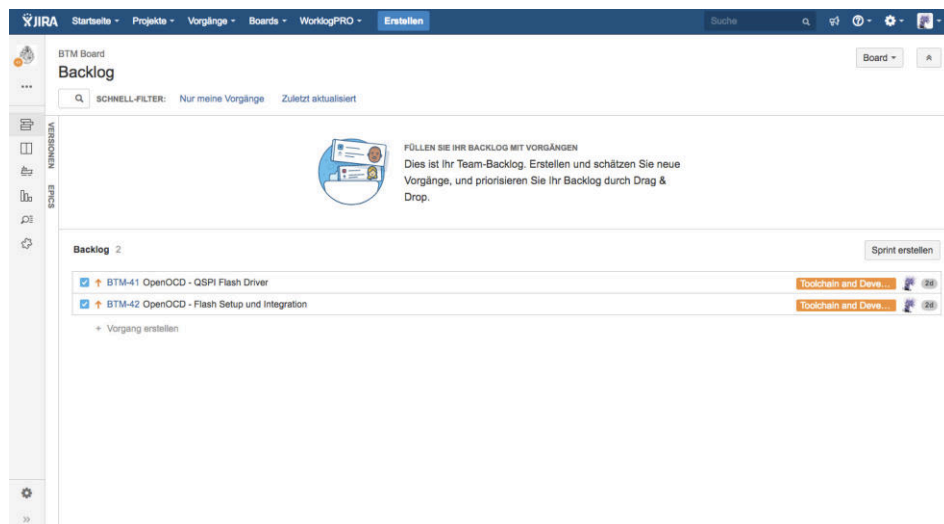


Abbildung 3.3: Jira Backlog zum Ende der Bachelor Arbeit

3.2 Codelabs Git Repository

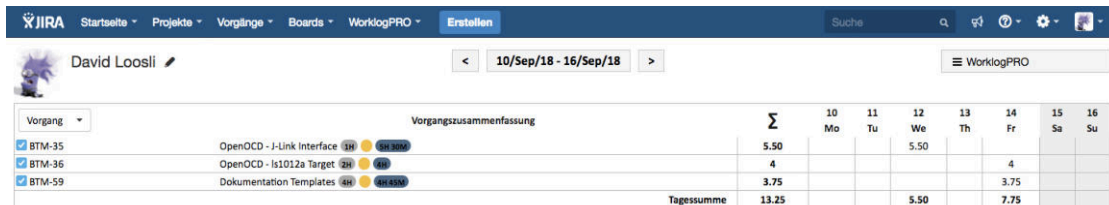
Im Gegensatz zur Studienarbeit wurde bei der Bachelor Arbeit auf einen graphischen Git Client verzichtet. Das entsprechende Repository wurde von codelabs zur Verfügung gestellt und über das Terminal des jeweiligen Betriebssystems genutzt.

3.3 Backup

Aufgrund der eingeschränkten Systembackup Möglichkeiten und des fehlenden Hardware Zuganges zu den beiden verwendeten Servern wurden wöchentlich die jeweiligen Daten manuell exportiert und auf einer externen Harddisk abgelegt. Auch wenn die Programminstallation und -konfigurationen auf diese Art nicht gesichert werden konnte, so minimierte sich die Wahrscheinlichkeit eines Verlustes der erarbeiteten Dokumente zumindest in einem gewissen Umfang.

Appendix

A Wochenauszüge Worklog Pro (Jira)



Vorgang	Vorgangszusammenfassung	Σ	10 Mo	11 Tu	12 We	13 Th	14 Fr	15 Sa	16 Su
BTM-35	OpenOCD - J-Link Interface	5.50							
BTM-36	OpenOCD - Is1012a Target	4			5.50		4		
BTM-59	Dokumentation Templates	3.75					3.75		
Tagessumme		13.25			5.50		7.75		

Abbildung 3.4: Arbeitsaufwand Vorbereitungswochen



Vorgang	Vorgangszusammenfassung	Σ	17 Mo	18 Tu	19 We	20 Th	21 Fr	22 Sa	23 Su
BTM-37	OpenOCD - NXP FRDM Is1012a Board	9		4.50				4.50	
BTM-38	GDB Debugger - OpenOCD Integration	9		4			5		
BTM-40	Ada Toolchain	3.25					3.25		
BTM-45	Meeting One (KickOff)	3	3						
BTM-59	Dokumentation Templates	1	1						
BTM-60	Aufsetzen Administrationstools	7.75			7.75				
Tagessumme		33	4	8.50	7.75		8.25	4.50	

Abbildung 3.5: Arbeitsaufwand Woche 1



Vorgang	Vorgangszusammenfassung	Σ	24 Mo	25 Tu	26 We	27 Th	28 Fr	29 Sa	30 Su
BTM-39	GPS IDE	14.50	10.25	4.25					
BTM-41	OpenOCD - QSPI Flash Driver	1.50			1.50				
BTM-43	Code Debugger - UART Driver	22		5.75		6.50	9.75		
BTM-44	Code Debugger - State Dump	9.50						9.50	
BTM-46	Meeting Two	3.75			3.75				
BTM-61	Ada GPRBuild - Project und GPS Integration	10							10
Tagessumme		61.25	10.25	10	5.25	6.50	9.75	9.50	10

Abbildung 3.6: Arbeitsaufwand Woche 2

JIRA Startseite Projekte Vorgänge Boards WorklogPRO Erstellen Suche

David Loosli < 1/Okt/18 - 7/Okt/18 > WorklogPRO

Vorgang	Vorgangszusammenfassung	Σ	1 Mo	2 Tu	3 We	4 Th	5 Fr	6 Sa	7 Su
BTM-44	Code Debugger - State Dump 1W 3D 3H 1W 2D 3H	21.75							
BTM-47	Meeting Three 3H 30M 2H 30M	2.50		1.50	1				
BTM-66	Code Debugger - String Conversion 1D 4D 30M	8.50	8.50						
BTM-67	Aufgabenstellung - Header 4H 3H 3H	3							3
BTM-68	Code Style - Muen 4H 2H 30M 1H 30M	1.50				1.50			
BTM-69	Code Style - Spaces 2H 1H 4H	1				1			
	Tagessumme	38.25	8.50	1.50	1		11	8.75	7.50

Abbildung 3.7: Arbeitsaufwand Woche 3

JIRA Startseite Projekte Vorgänge Boards WorklogPRO Erstellen Suche

David Loosli < 8/Okt/18 - 14/Okt/18 > WorklogPRO

Vorgang	Vorgangszusammenfassung	Σ	8 Mo	9 Tu	10 We	11 Th	12 Fr	13 Sa	14 Su
BTM-44	Code Debugger - State Dump 1W 3D 3H 1W 2D 3H	25.75							
BTM-48	Meeting Four 3H 4H 3H	4	2.50	1.50					
BTM-62	SK Components - Startup ASM 1D 4D 1H 15M	2.50						2.50	
BTM-63	SK Components - Context Switch 2D 2D 1H	10						4	6
BTM-64	SK Components - Exception Handling 2D 1H 1D 3H	3.50						3.50	
BTM-65	Development Environment Setup - Dokumentation 2D 1D 7H 15M 45M	0.75							0.75
	Tagessumme	46.50	11.50	1.50	6	10.75	10	6.75	

Abbildung 3.8: Arbeitsaufwand Woche 4

JIRA Startseite Projekte Vorgänge Boards WorklogPRO Erstellen Suche

David Loosli < 15/Okt/18 - 21/Okt/18 > WorklogPRO

Vorgang	Vorgangszusammenfassung	Σ	15 Mo	16 Tu	17 We	18 Th	19 Fr	20 Sa	21 Su
BTM-44	Code Debugger - State Dump 1W 3D 3H 1W 2D 3H	4	4						
BTM-62	SK Components - Startup ASM 1D 4D 1H 15M	14.75			3.25	6	3.50		2
BTM-63	SK Components - Context Switch 2D 2D 1H	7				5	2		
BTM-64	SK Components - Exception Handling 2D 1H 1D 3H	7.50	7.50						
BTM-70	SK Components - SPSR Register 1D 2H 30M 1H 30M	5.50			5.50				
BTM-72	SK Components - MMU Initialisierung 1D 4H 2D 4H	20						11	9
	Tagessumme	58.75	11.50		8.75	11	5.50	11	11

Abbildung 3.9: Arbeitsaufwand Woche 5

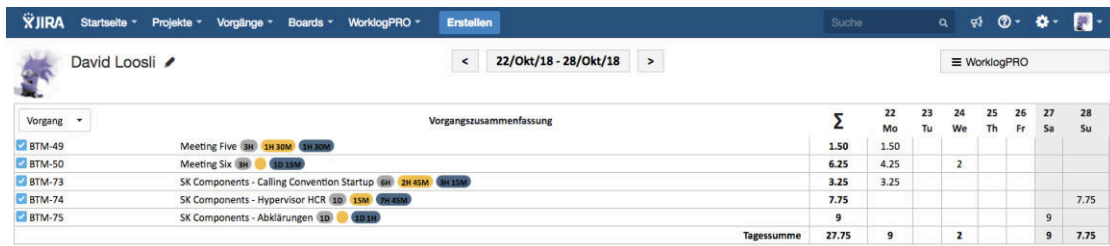


Abbildung 3.10: Arbeitsaufwand Woche 6



Abbildung 3.11: Arbeitsaufwand Woche 7

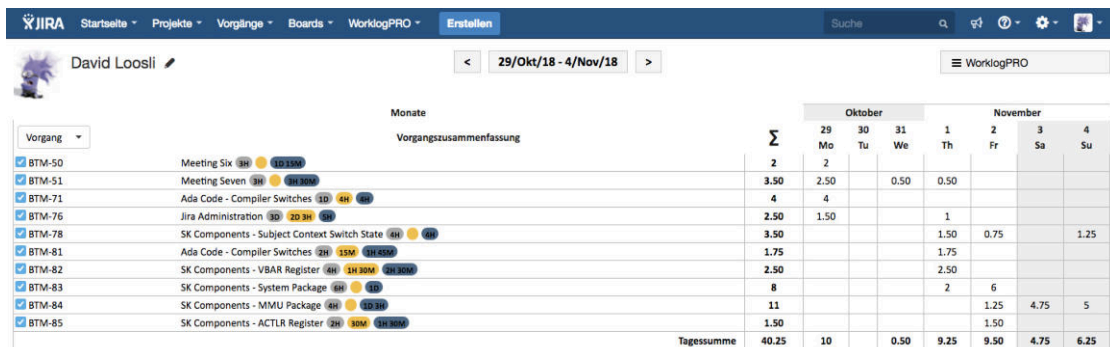


Abbildung 3.12: Arbeitsaufwand Woche 8

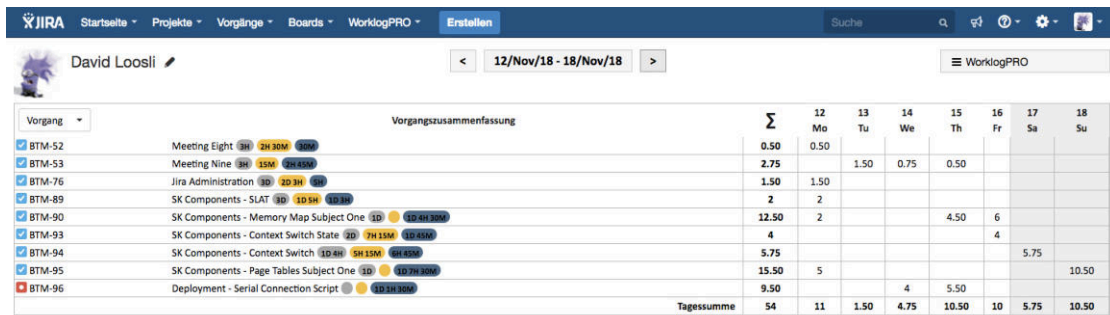


Abbildung 3.13: Arbeitsaufwand Woche 9

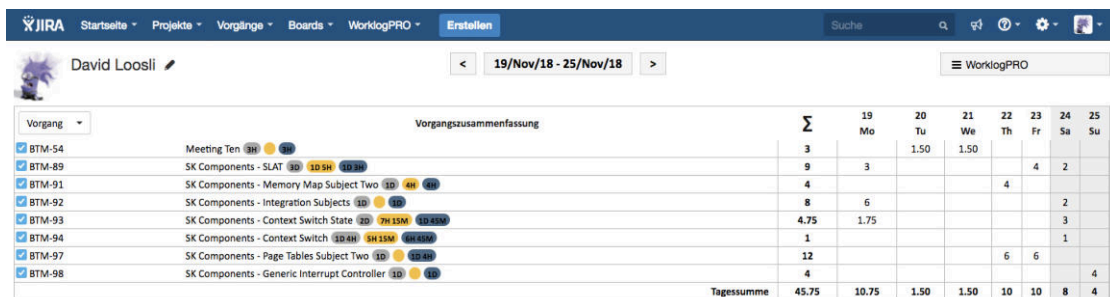
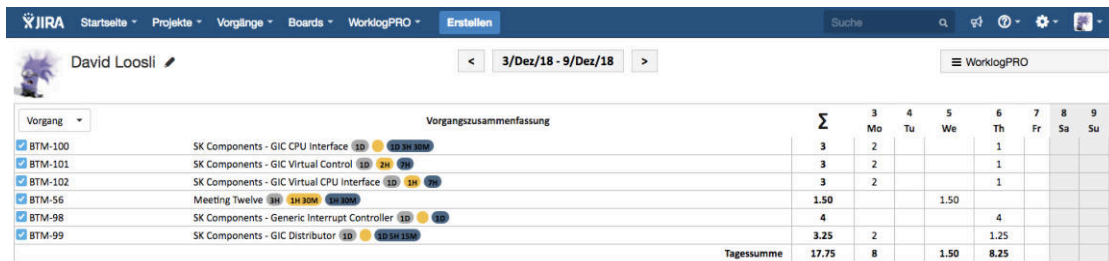


Abbildung 3.14: Arbeitsaufwand Woche 10



Abbildung 3.15: Arbeitsaufwand Woche 11



David Loosli

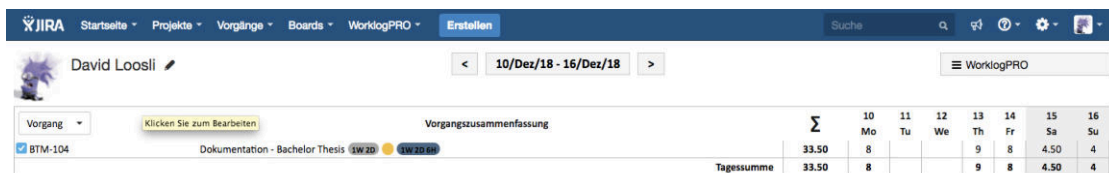
3/Dec/18 - 9/Dec/18

Vorgang

Vorgangszusammenfassung

	Σ	3 Mo	4 Tu	5 We	6 Th	7 Fr	8 Sa	9 Su
BTM-100 SK Components - GIC CPU Interface	3	2			1			
BTM-101 SK Components - GIC Virtual Control	3	2			1			
BTM-102 SK Components - GIC Virtual CPU Interface	3	2			1			
BTM-56 Meeting Twelve	1.50			1.50				
BTM-98 SK Components - Generic Interrupt Controller	4				4			
BTM-99 SK Components - GIC Distributor	3.25	2			1.25			
Tagessumme	17.75	8		1.50	8.25			

Abbildung 3.16: Arbeitsaufwand Woche 12



David Loosli


10/Dec/18 - 16/Dec/18

Vorgang

Vorgangszusammenfassung

	Σ	10 Mo	11 Tu	12 We	13 Th	14 Fr	15 Sa	16 Su
BTM-104 Dokumentation - Bachelor Thesis	33.50	8			9	8	4.50	4
Tagessumme	33.50	8			9	8	4.50	4

Abbildung 3.17: Arbeitsaufwand Woche 13



David Loosli

17/Dec/18 - 23/Dec/18

Vorgang

Vorgangszusammenfassung

	Σ	17 Mo	18 Tu	19 We	20 Th	21 Fr	22 Sa	23 Su
BTM-104 Dokumentation - Bachelor Thesis	28.50	8	9	10	0.50	1		
BTM-105 Dokumentation - Persönlicher Bericht	3				2.50	0.50		
BTM-106 Dokumentation - Projektorganisation	3.50				3.50			
BTM-107 Dokumentation - Management Summary	5				4	1		
BTM-108 Dokumentation - Poster	6				4	2		
BTM-76 Jira Administration	1					1		
Tagessumme	47	8	9	10	14.50	5.50		

Abbildung 3.18: Arbeitsaufwand Woche 14

B Protokolle

*Es wurde darauf verzichtet, in der Druckversion sämtliche Protokolle der Besprechungen mit auszu-
drucken. Diese können jedoch entweder im pdf Format von der ebenfalls beiliegenden CD eingesehen
oder im Original vom Autor der Studie verlangt werden.*

Student Research Project
autumn semester 2017

Student Research Study

Muen on ARM - an Evaluation

version: 1.0, date: December 21, 2017

supervisors:

Prof. Dr. Andreas Steffen
MSc Adrian-Ken Rüegsegger
MSc Reto Bürki

HSR, Rapperswil



David Loosli, student
BSc in Computer Science
HSR Rapperswil

Change History

date	version	change	author
Oct 15, 2017	0.1	prepared template, setup basic version	David Loosli
Nov 5, 2017	0.1	bibliography; changed structure according to previous findings	David Loosli
Nov 11, 2017	0.2	introduction incl. bibliography and glossary	David Loosli
Nov 12, 2017	0.2	first part of chapter 2 (overview, SPARK requirements)	David Loosli
Nov 14, 2017	0.2	second part of chapter 2 (SPARK requirements, virtualization basics)	David Loosli
Nov 14, 2017	0.2	third part of chapter 2 (memory)	David Loosli
Nov 17, 2017	0.2	third part of chapter 2 (memory)	David Loosli
Nov 18, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 23, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 24, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 25, 2017	0.2	fourth and fifth part of chapter 2 (interruptions, device handling)	David Loosli
Nov 26, 2017	0.3	fifth part (interruptions) and summary / last check of chapter 2	David Loosli
Nov 28, 2017	0.3	structure chapter 3 (incl. introduction)	David Loosli
Dec 2, 2017	0.3	corrections of chapter 2 according to meeting	David Loosli
Dec 3, 2017	0.4	first part of chapter 3 (overview, coding)	David Loosli
Dec 4, 2017	0.4	first part of chapter 3 (coding, startup)	David Loosli
Dec 5, 2017	0.4	second part of chapter 3 (fundamentals)	David Loosli
Dec 7, 2017	0.4	second part of chapter 3 (fundamentals, virtualization basics)	David Loosli
Dec 8, 2017	0.4	third part of chapter 3 (virtualization basics, caching)	David Loosli
Dec 9, 2017	0.4	third part of chapter 3 (caching, memory)	David Loosli
Dec 10, 2017	0.4	third part of chapter 3 (memory)	David Loosli
Dec 12, 2017	0.4	fourth part of chapter 3 (exception handling, timer)	David Loosli

date	version	change	author
Dec 13, 2017	0.4	fifth part of chapter 3 (spark and requirement comparison)	David Loosli
Dec 15, 2017	0.5	corrections up to chapter 2 (incl. rewriting)	David Loosli
Dec 16, 2017	0.5	corrections of chapter 3 (incl. rewriting)	David Loosli
Dec 17, 2017	0.5	first part of chapter 4 (overview, and boot process)	David Loosli
Dec 19, 2017	0.5	second part of chapter 4 (exception and device handling)	David Loosli
Dec 19, 2017	0.6	chapter 5 and abstract	David Loosli
Dec 21, 2017	1.0	corrections of chapter 4 and 5 as well as final review	David Loosli

Abstract

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

This feasibility study investigates the ARMv8-A architecture and in particular the AArch64 Virtualization Extensions introduced with the latest ARM architecture and evaluates how this technology could be used for porting the Muen SK to ARM. In order to be able to achieve this, the mechanisms used by Muen SK are first examined in detail. Based on this investigation, the requirements for a target processor architecture are derived and compared with the features provided by the ARMv8-A architecture. Since the target hardware platform for this study is the Raspberry Pi 3, the requirements declared as „implementation defined“ by the ARM documentation are finally assessed with respect to this System on Chip designed by the Raspberry Pi Foundation.

Contents

Change History	2
Abstract	4
1 Introduction	7
1.1 Structure of the Study	8
1.2 Related Documents	8
1.3 Literature	8
2 Muen Separation Kernel	9
2.1 Virtualization Basics	10
2.2 Memory	11
2.2.1 Caches	13
2.2.2 Memory Management	13
2.2.3 Advanced Memory Virtualization	16
2.2.4 Multicore Environment	17
2.3 Interruption Handling	18
2.3.1 Programmable Interrupt Controller	20
2.3.2 Interrupts	21
2.3.3 Exceptions and Software Generated Interrupts	21
2.3.4 Traps	22
2.3.5 Events	23
2.4 Timers	23
2.5 Device Handling	24
2.6 Floating Point	25
2.7 SPARK	25
2.8 Derived Requirements	26
3 ARMv8 Architecture	29
3.1 Code Examples	30
3.1.1 Code Compilation	30
3.1.2 Code Execution and Debugging	31
3.2 Fundamentals	34
3.2.1 Exception Levels	34
3.2.2 Execution States	37
3.2.3 Startup and Reset	38
3.3 Virtualization Basics	39
3.4 Memory	41
3.4.1 Caches	41

3.4.2	Memory Management	43
3.4.3	Advanced Memory Virtualization	45
3.4.4	Multicore Environment	46
3.5	Exception Handling	47
3.5.1	Interrupts	49
3.5.2	SErrors	49
3.5.3	Aborts	49
3.5.4	Exception Generating Instructions	50
3.5.5	Resets	51
3.5.6	Generic Interrupt Controller	51
3.6	Timers	52
3.7	Device Handling	53
3.8	SPARK	53
3.9	Requirement Comparison	55
4	Raspberry Pi 3	56
4.1	Overview	56
4.1.1	Documentation	57
4.1.2	Bare Metal Development	59
4.2	Boot Process	60
4.3	Exception Handling	61
4.4	Device Handling	62
4.5	SPARK	62
5	Conclusion	63
5.1	ARMv8 Architecture	63
5.2	Raspberry Pi 3	63
5.3	Further Investigations	64
	Appendix	66
A	List of Related Documents	66
B	Project Assignment AVT (german)	67
	Bibliography	68

1 Introduction

The evolution within the last years in the world of information technology not only led to a tremendous increase of mobile devices and networking, but also let the world economy dream of a new technological era, the Industry 4.0¹. This fourth industrial revolution is characterized in particular by the interconnection of objects and people within a so called information network. In this context, the most frequently mentioned keywords are Internet-of-Things, Cloud Computing and Bioengineering.

One of the consequences of the integration of autonomously communicating devices into our daily life is that a lot of sensitive data is collected and stored that needs best possible access control. A mathematically provable secure approach to control the access to sensitive data is the theory of the Separation Kernel published by John Rushby in a paper presented at the 8th ACM Symposium on Operating System Principles in December 1981². Based on this theoretical foundations and the Intel hardware virtualization extension, Reto Buerki and Adrian-Ken Rueegsegger designed the Muen Separation Kernel (SK) as their Master Thesis at the University of Applied Sciences Rapperswil (HSR)³. The Muen SK ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system.

A second consequence of the fourth industrial revolution is the need for small devices with low energy consumption and low production costs that still meet the state of the art with respect to processor architecture and peripheral device integration. Since many of these small devices, especially mobile devices, use an ARM central processing unit (CPU) or an ARM based system on chip (SoC) one could also determine enormous improvements up to the latest ARM architecture, the so called ARMv8 architecture⁴.

This Student Research Study, which is part of the Bachelor of Science in Computer Science program at the University of Applied Sciences Rapperswil (HSR), investigates the possibility of porting the Muen SK to the ARMv8 architecture. As the Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components, the aim of this feasibility study is to take a closer look at the ARMv8 architecture and in particular the AArch64 Virtualization Extension (VE) introduced with the latest ARM processors. The target hardware for this study is the Raspberry Pi 3⁵.

¹[3] Devezas, Leitão, and Sarygulov. *Industry 4.0 - Entrepreneurship and Structural Change in the New Digital Landscape*. 2017, Chapter 1, page 2 f.

²[17] Rushby. "Design and Verification of Secure Systems". 1981.

³[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013.

⁴https://en.wikipedia.org/wiki/ARM_architecture#ARMv8-A, December 21, 2017

⁵cf. <https://www.raspberrypi.org/>, December 21, 2017

1.1 Structure of the Study

The study is divided into three main parts followed by a summarizing conclusion including a risk assessment for the planned Bachelor Thesis to port the Muen SK to the ARMv8 architecture. In the first part (chapter 2), an overview of the Muen SK is given and the most important hardware dependent features are described, from which the general hardware requirements are derived. In the next chapter 3, an introduction to the ARMv8 architecture is presented with focus on the AArch64 architecture and the Virtualization Extension (VE) followed by a qualification of these features with respect to the derived hardware requirements from the first part. As the target hardware for this study is the Raspberry Pi 3, the third part of this document (chapter 4) is dedicated to a detailed description of this single board computer considering hardware related features used by the Muen SK.

1.2 Related Documents

As the focus of this study lies on the feasibility of porting the Muen SK to the ARMv8 architecture, many related documents apart from this document were elaborated. As examples, there can be mentioned the Raspberry Pi 3 Beginner's Guide and all the Evaluation Cases illustrated with small coding examples. All this documents are an integral part of the Student Research Project. A list can be found in the appendix of this document.

1.3 Literature

Due to the task description of the Student Research Project ⁶, the Muen Report ⁷ with the related documents and the official ARM documentation, i.e. the ARMv8 Architecture Reference Manual ⁸ and the ARM Cortex-A Series Programmer's Guide ⁹, were used as the principal literature. A detailed list of referenced literature can be found in the bibliography at the end of this document (cf. [Bibliography](#)).

Because a detailed and with respect to the AArch64 architecture complete Raspberry Pi 3 hardware reference manual did not exist at the time of writing, chapter 4 of this study had to be based on the VideoCore Reference Manual ¹⁰ and the BCM2835 ARM Peripherals documentation ¹¹ for the Raspberry Pi 1 as well as different online sources mentioned in the corresponding section 4.1.1 of this document.

⁶cf. assignment from the AVT platform, Appendix B

⁷[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013.

⁸[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017.

⁹[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015.

¹⁰[13] n.a. *VideoCore IV 3D Architecture Reference Guide*. 2013.

¹¹[11] n.a. *BCM2835 ARM Peripherals*. 2012.

2 Muen Separation Kernel

The design and implementation of the Muen SK is premised on three basic concepts: first of all the Separation Kernel principle, formal verification and hardware supported virtualization.

The concept of a Separation Kernel was introduced by John Rushby in a paper presented at the 8th ACM Symposium on Operating System Principles in December 1981 as a solution to the problem with the development and verification of large, complex security kernels¹. His proposition was to basically adapt the principles of a distributed system to a single processor to avoid the aforementioned problem. As a consequence, such a system has to physically isolate all the subjects that are part of the security policy. The communication and the access to shared resources of all these subjects must be handled only through likewise isolated, so called trusted components that can be verified². Finally, Rushby verified the outlined proposition with a Proof of Separability³.

As the verification is a compulsory consequence of the Separation Kernel principle, an implementation of a Separation Kernel has to use a programming language that is amenable to formal verification. Therefore, the SPARK programming language was chosen to write the Muen SK. SPARK is a formally analysable subset of the programming language Ada and used for implementing high integrity systems⁴. A introduction to the programming language SPARK and the related derived requirements can be found in section 2.7.

Another deducible consequence of the Separation Kernel principle is the requirement of a sufficiently small code base for the implementation of such a kernel⁵. To achieve this, the Muen SK relies on the hardware virtualization support of the Intel x86 architecture⁶. To get the full virtualization support for a desktop environment, the Intel IA-32e/64-bit architecture was chosen as the target platform of the Muen SK⁷. Therefore, a first basic requirement for a processor architecture, to be able to run the Muen SK on, can be derived as:

REQ-0: *The processor architecture has to support 64 bit datapath widths, integer size and memory address widths as well as to be able to execute 32 bit applications.*

¹[17] Rushby. "Design and Verification of Secure Systems". 1981, Section 1, page 3 f.

²[17] Rushby. "Design and Verification of Secure Systems". 1981, Section 2 f., page 5 ff.

³[17] Rushby. "Design and Verification of Secure Systems". 1981, Section 4, page 11 ff.

⁴cf. <https://www.adacore.com/sparkpro>, December 21, 2017

⁵[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 2.4, page 14.

⁶[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 2.3, page 11 ff.

⁷[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 3.2, page 20.

2.1 Virtualization Basics

A hypervisor or virtual machine monitor (VMM)⁸ is special software that emulates computer hardware. In general, two different types of hypervisors are classified⁹: *Type I* native or bare-metal hypervisors and *Type II* hosted hypervisors. A Type I hypervisor directly runs on the target hardware to control and manage the guest operating system, whereas a Type II hypervisor makes use of a conventional operating system. As a Type I hypervisor has comprehensive control over the processor(s) and other platform hardware as well as over the guest software (e.g. memory access, communication etc.), it can also be used as a mechanism for separation purpose¹⁰. Therefore, the Muen SK can be classified as a Type I hypervisor.

A hypervisor multiplexes the hardware by the usage of different virtualization techniques to provide a virtual environment to the guest software in a way that lets the guest software gain the impression of running directly on the hardware. One approach to achieve this, is to add another privilege level or protection ring to a processor architecture. A protection ring is one of two or more hierarchical layers of privilege within the architecture of a computer system. Normally, the processor architecture enforces this layering by providing different execution modes on hardware level. As an example - in standard protected mode on an Intel x86 architecture there exist four privilege levels or protection rings with ring 0 as the most privileged one whereas ring 3 having the least privileges¹¹.

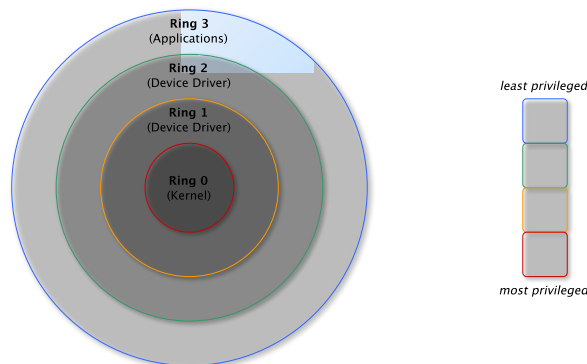


Figure 2.1: Intel x86 protection mode, protection rings hierarchy

As already mentioned, the Muen SK makes use of the Intel Virtualization Technology (VT) to fulfil the requirement of a small code base. One of the basic features of the Intel VT is the so called Intel VT-x.

⁸Because the ARMv8 architecture uses the terms secure monitor and monitor mode for a separate exception level, the expression hypervisor is used instead of VMM throughout this document.

⁹[16] Popek and Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". 1974, the first classification approach.

¹⁰[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3, page 11.

¹¹It is absolutely important to note that ring 0 has the **most** privileges - because the ARM Exception Levels define the privileges exactly the other way round by giving the Exception Level 0 the **least** privileges; cf. https://en.wikipedia.org/wiki/Protection_ring, December 21, 2017

This feature introduces a new hypervisor execution level with an additional protection ring „-1“ as well as some new VMX instructions that simplify the switching between a hypervisor running in VMX root operation and guest software executing in VMX non-root operation¹². Hence, to be able to execute the Muen SK in hypervisor mode, a target processor architecture has to meet the following requirement:

REQ-1: *The target processor architecture must provide a virtualization extension that is capable of running a Type I hypervisor. This requirement includes the hardware assisted support for an additional privilege level and instructions for a simplified switch between this additional and other privilege level.*

Another important feature of the VT-x virtualization technology is that VM exits and entries are handled automatically while the exact behaviour still stays configurable¹³. To do so, a logical processor uses virtual machine control data structures (VMCS) to manage transitions into and out of the VMX non-root operation as well as the processor behaviour in VMX non-root operation¹⁴. An illustrating example is a VM exit that automatically stores the guest processor state into the guest state area of the VMCS. But one has to be aware that registers, which can be saved and loaded by the hypervisor itself (e.g. general purpose registers), are not stored automatically¹⁵. Therefore:

REQ-2: *The target processor architecture must provide a virtualization extension that supports an automatic handling of guest exits (i.e. traps) and entries. At least, the target processor architecture must provide a support mechanism to completely save and load all the relevant guest state structures.*

2.2 Memory

In modern computer systems, usually different memory and storage technologies are used as an attempt to find the best possible compromise between access time, cost and persistence properties. The first two criteria are interrelated by the fact that the shorter the access times of a specific type of memory is, the more expensive they are. The third criterion not only considers the persistence in the proper sense, i.e. volatile or persistent, but also other properties like the degree of hardware supported manageability. Therefore, memory and storage are normally organized in a so called memory hierarchy to use the advantages of the various components while, at the same time, circumventing their disadvantages¹⁶. A standard modern memory hierarchy is composed of^{17 18}:

¹²more details can be found in [2], section 2.3.1, and [12], volume 3C, chapter 23 f., page 1083 ff.

¹³[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3, page 12.

¹⁴[12] n.a. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*. 2017, volume 3C, section 24.4, page 1090 ff.

¹⁵[14] Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". 2006, page 170.

¹⁶cf. <https://de.wikipedia.org/wiki/Speicherverwaltung>, December 21, 2017

¹⁷[4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 7.1.3, page 393 f.

¹⁸https://en.wikipedia.org/wiki/Memory_hierarchy, December 21, 2017

- *CPU Registers*: The fastest (typically one clock cycle) and most expensive type of memory that locates in the processor itself.
- *Caches*: A state of the art processor has numerous internal and shared caches (Static Random Access Memory SRAM), organized in up to four levels with increasing access times from a few tens of clock cycles down to a few hundreds, and additional hardware caching structures, e.g. Translation Lookaside Buffers (TBL) and Branch Prediction Caches (BPC).
- *Primary Storage (main memory)*: This type of memory is also referred to as Dynamic Random Access Memory (DRAM). Its speed is moderate with up to 10 GB per second but still relatively affordable. With respect to primary storage, two different applications are distinguished - physical RAM and Virtual Memory (cf. Memory Management Unit 2.2.2).
- *Secondary Storage (disk storage)*: On Secondary Storage, data can be permanently stored. It is much cheaper than primary storage but about 10'000 times slower. The most known representatives are Hard Disk Drives (HDD) or Solid State Disks (SSD).
- *Tertiary Storage (input storage)*: This category includes various types of removable media devices such as USB devices or SD cards as well as remote storage and peripherals. It is the slowest and cheapest kind of storage.

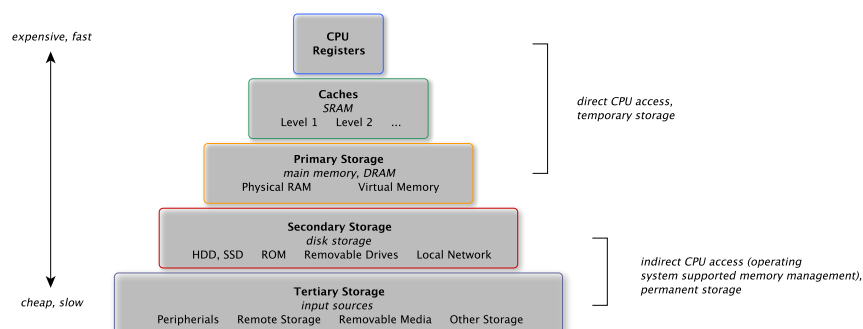


Figure 2.2: example of a memory hierarchy

At this point, one has to remember the strict distinction between memory and storage. While the CPU has direct access to the memory - whether through the processor's hardware structures or over the memory bus - storage is only available as an I/O device. Pointing out this difference is important because all memory resources of a system running the Muen SK are static and explicitly specified in the so called system policy¹⁹. This, for example, implies that there is no such mechanism implemented for loading missing page contents from a storage device after a page fault or page miss, as most of the common operating system kernels would do, and that no considerations about side and covert

¹⁹[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4.2.1, page 24.

channels with respect to disk caches or other storage structures have to be made. Since a subject²⁰ cannot change its own address space, also the page tables are static and therefore can be generated in advance according to the relevant information in the system policy. As for the storage, it is treated by the Muen SK as a pure I/O device (cf. Device Handling 2.5) .

2.2.1 Caches

As already mentioned, only cache and caching structures, that are directly accessible to the CPU, have to be considered with respect to the fundamental requirement of the Muen SK to completely separate the subjects and thus to eliminate side and covert channels. The main problem with caches is that they are shared and can normally only be controlled to a limited degree²¹. Due to performance aspects, the Muen SK has to enable the caches and caching structures. But as the Muen SK uses the Intel Virtualization Extension at least the Translation Lookaside Buffer (cf. section 2.2.2) is cleared automatically. Therefore, a processor architecture has to fulfil the following requirement:

REQ-3: *The target processor architecture shall provide a minimal set of cache management features and an automatic cache clearing feature in the context of virtualization. At least, the target processor architecture must provide a support mechanism to clear caches manually.*

Even though out of scope for this study, the cache colouring mechanism has to be mentioned here. This technique first divides the cache into disjoint units and assigns a „color“ to each of these partitions. Every process then is assigned a certain color to. A cache area of certain color can only be accessed by processes with the corresponding color. This technique is not only used for performance optimizations but can also serve as a mechanism to prevent processor caches from being used as high-bandwidth side channels²². The developers of the Muen SK mentioned this mechanism as one of possible future enhancements²³.

2.2.2 Memory Management

In modern computer systems, the management of the main memory is taken over by a hardware component called Memory Management Unit (MMU) that is usually integrated into the processor. The MMU handles all the access of the CPU to the main memory. In general, it has two main functions: on the one hand it allows the implementation of virtual memory and on the other hand it can manage memory protection and cache control²⁴.

²⁰In the context of the Muen SK, a subject is defined as one of multiple, isolated and through well-defined interfaces interacting components. More informations can be found in section 3.3 and 4.3 in [2]

²¹[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.2.1.2, page 7.

²²[1] Braun, Jana, and Boneh. "Robust and Efficient Elimination of Cache and Timing Side Channels". 2015, section 4, page 3, with references to other literature.

²³[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 6.2.1.1, page 76.

²⁴https://en.wikipedia.org/wiki/Memory_management_unit, December 21, 2017

Virtual Memory is a technique that abstracts the available memory and storage resources on a computer system in such a way that a process is given the illusion of running alone on that system and having unrestricted access to the systems main memory²⁵. To be able to provide a linear²⁶ but virtual logical address space to a process, a modern MMU uses a mechanism called paging. With paging the physical as well as the virtual address space are divided into units with a fixed size. In the context of physical memory, these units are called page frames whereas in the context of virtual memory they are denoted as pages. The mapping between a physical page frame and a virtual page is done by a so called page table that uses the two-part virtual address to calculate the physical address of the page frame.

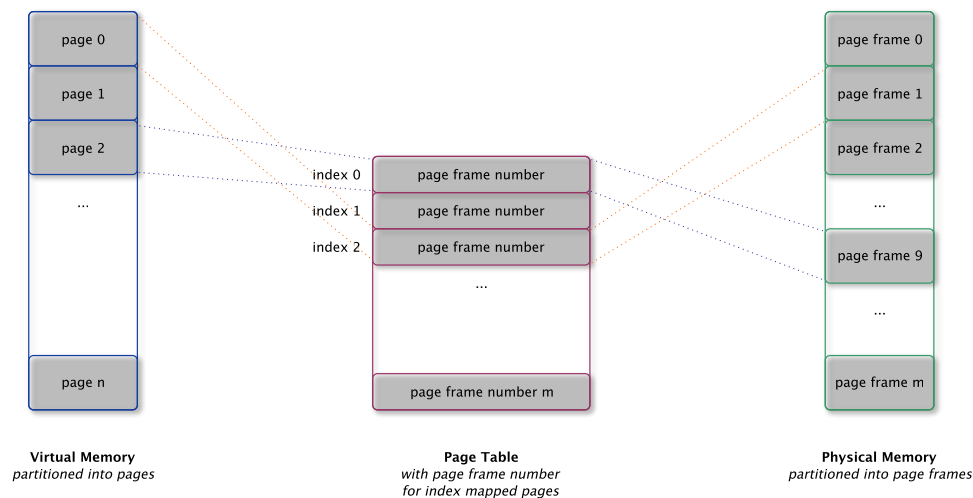


Figure 2.3: example of a one level paging with partitioning

As shown in figure 2.4, a virtual address is divided into two parts - a page number and an address offset. The page number serves as an index into the page table to read out the content at this specified address, namely the page frame number. Then the page frame number is multiplied by the predefined page size to get the base address of the corresponding page frame in the physical memory. The physical address for the requested virtual address can then be obtained by adding the offset of the virtual address to this base address²⁷.

To prevent illegal accesses, the page table must be initialized completely and undefined page table entries have to be invalidated by at least setting an invalidation bit. In combination with a multiprocessor environment, this requirement can lead to large page tables. One way to address this problem, is to implement so called multi-level page tables. As an example, a two level page table hierarchy is presented: In such a case, the virtual address is divided into three parts. The first part contains the

²⁵ https://en.wikipedia.org/wiki/Virtual_memory, December 21, 2017

²⁶ https://en.wikipedia.org/wiki/Flat_memory_model, December 21, 2017

²⁷ [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 7.5.1, paragraph Seitenbasierte Adressumsetzung, page 450 ff.

directory index to the first page table. The entry in this first page table does not directly return the page frame number, but contains the address of a second page table. The second part of the virtual address is now used as an index into this second page table. The entry at the corresponding index then contains the page frame number, which serves as the basis for the actual address resolution according to the principle described above²⁸.

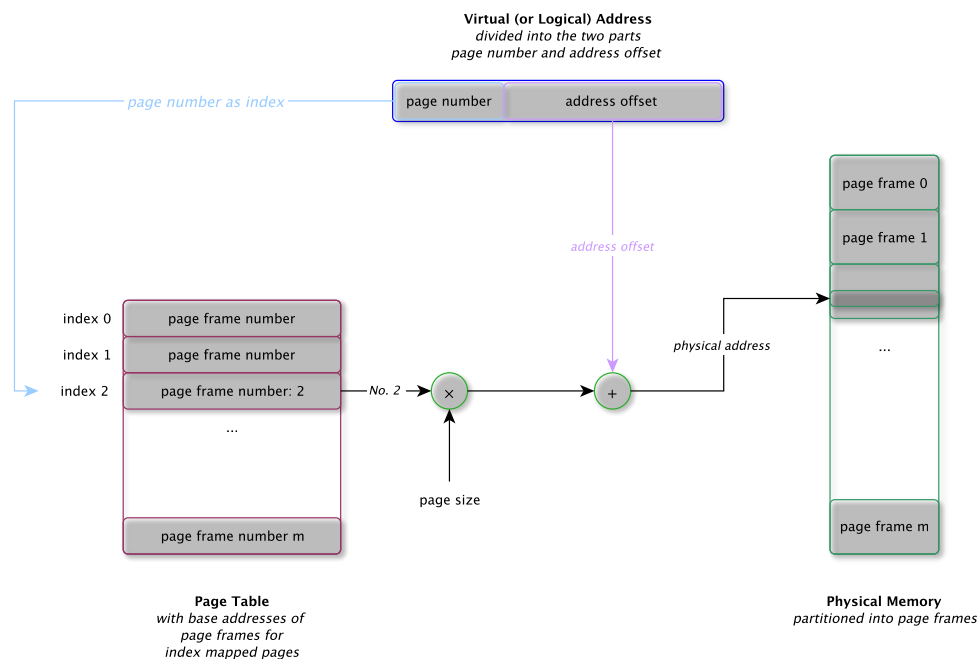


Figure 2.4: example of a one level address translation

To improve the performance of the MMU's address translation, modern processor architectures rely on the implementation of an associative cache structure, the Translation Lookaside Buffer (TLB). The address translation principle described above remains the same, but instead of a direct lookup of the first part of the virtual address (i.e. the page number) in a page table the MMU first takes a look at the TLB. If the corresponding page entry can be found in the TLB it loads the physical base address directly from there - else the corresponding page descriptor gets first loaded into the TLB from the according page table before returning it to the address translation process²⁹. In the context of the Muen SK, the separation concerns described in the section Caches 2.2.1 have to be considered accordingly (i.e. side and convert channels).

²⁸[4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 7.5.1, paragraph Seiten-basierte Adressumsetzung, page 455 ff.

²⁹[4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 7.5.1, page 446 f.

The Muen Separation Kernel uses the virtualization functions of the Intel IA-32e mode on the one hand for the implementation of the Type I hypervisor and on the other hand for the partitioning and the separation of the different subjects. Since the 80286 processor, the Intel x86 processor architecture provides an integrated MMU, that is capable of handling the paging mechanism for address virtualization. The corresponding page tables can be defined and used on a per process basis and also serve to define properties and permissions (i.e. memory protection). In addition, the MMU provided by Intel validates and enforces compliance with these additional memory protection features. A hierarchical arrangement of the page tables also enables multi-level paging - the Intel IA-32e mode supports up to 4 such levels and allows page sizes of 4 KB, 2 MB and 1 GB³⁰. Therefore, the target architecture has to support the following features:

REQ-4: *The target processor architecture has to provide a Memory Management Unit that supports:*

- (i) memory virtualization on a per subject basis (one page table per subject),*
- (ii) definition of properties and permissions per page table (read/write access, execute disable, caching behaviour),*
- (iii) checking and enforcement of defined properties and permissions,*
- (iv) different page sizes (i.e. large page support³¹), but at least a 4 KB page size³².*

2.2.3 Advanced Memory Virtualization

When using a hypervisor with different guest operating systems (i.e. virtual machines), the address virtualization technologie described above has to be extended with a second layer. The hypervisor assigns a first layer virtual memory area to the guest system, which is interpreted by the guest system as its own physical memory. If the guest system is running a modern operating system, it will use the address translation mechanism again for its applications, creating a complete second address translation layer. In order to be able to cope with the associated performance issues as well as the complexity of the hypervisor implementation, Intel's x86 virtualization technology „Extended Page Tables (EPT)“ provides a hardware assisted Second Level Address Translation (SLAT, also known as nested paging) mechanism.

³⁰[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.2.2, page 8 f.

³¹Most current CPU architectures support bigger pages, but name it differently: huge pages, super pages or large pages are only the most often used terms.

³²At the time of writing, the Muen SK only relies on 4KB pages. But as discussed in the meeting of November 20, 2017, the question about large page support by the ARM architecture should be answered in this study too.

The Muen architecture supports native subjects as well as complex Virtual Machines (VM) running their own operating system³³. To be able to run such complex VM's without having an enormous adaptation effort, the Muen SK makes use of the Intel's x86 SLAT virtualization technology EPT³⁴. Therefore, the following must apply:

REQ-5: *The target processor architecture must support hardware assisted second level address translation (SLAT).*

2.2.4 Multicore Environment

Even though out of scope for this study, the multicore environment topic has to be mentioned here. A processor architecture that implements more than one core is called a multicore processor³⁵. Another feature often implemented by modern processor architectures is the hardware assisted multithreading ability. A processor architecture, that is capable of multithreading, subdivides a central processing unit (CPU) or a single core in a multicore processor into logical cores to execute multiple processes or threads concurrently³⁶. While in a multicore environment the CPU itself as well as core specific resources (e.g. MMU, TLB and Caches) are multiplied, logical cores have to share these resources.

First of all, the Muen SK does not concern itself with memory management. All the page table structures needed in a computing system are created by the Muen policy tools and statically initialized at the system startup by the initialization code. In an initialized multicore or multithreading environment, all logical cores execute exactly the same (i.e. binary identical) Muen kernel code. Although each kernel has its own stack page and a page to store per core data, this is fully transparent to the kernels due to the usage of different page table structures per kernel³⁷. In the current version of the Muen SK, the multithreading features of the Intel x86 architecture are switched off³⁸. Therefore, to be able to port the Muen SK to another multicore or multithreading processor architecture, a target architecture has to provide the following feature:

REQ-6: *A multicore target processor architecture has to provide a mechanism to switch off the multithreading mechanism on a per core basis, if multithreading is supported.*

The Muen SK uses a barrier as synchronization mechanism to avoid any interprocessor drift in the context of scheduling plans and hence to eliminate timing side channels. This barrier guarantees that all logical cores have arrived at a specific execution point, i.e. on major frame transition, and are synchronized by waiting for the release. A sense-reversing barrier implemented in SPARK is used as

³³[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4, page 22 f.

³⁴[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.1.2, page 13, and section 3.3.3, page 22.

³⁵https://en.wikipedia.org/wiki/Multi-core_processor, December 21, 2017

³⁶[https://en.wikipedia.org/wiki/Multithreading_\(computer_architecture\)](https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)), December 21, 2017

³⁷[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4.6, page 28, and section 4.4.2, page 46.

³⁸cf. Besprechungsnotiz November 27, 2017 - section 2, page 2

the barrier mechanism³⁹. On assembly level, the barrier is realized with a spinlock using the atomic XCHG processor swapping instruction⁴⁰.

REQ-7: *A multicore target processor architecture shall provide a barrier synchronization mechanism. At least it must offer an atomic swapping instruction to support the according spinlock implementation.*

2.3 Interruption Handling

The various processor architectures and the corresponding literature use different terms (e.g. exception, interrupt, signal, event) for the temporary interruption of a running process by an interruption cause. For this study, the term interruption is used as a generic term for all types of temporary interruptions. The terms for the different types of interruptions are then described in detail according to the usage and the definitions in the respective topic. For example, in this chapter the interruption types are defined as used in the Muen report.

In the literature one can find various criteria to distinguish between interruptions and hence quite a few different categorisations of interruptions^{41,42}. For this study only the following criteria are relevant:

- *internal vs. external:* An interruption caused by a device outside the processor is referred to as external while interruptions caused by the processor itself are considered as internal. For example, a keyboard device signalling an input has to be qualified as external - in contrast, interruptions, that occur in response to a processing error, such as referencing an invalid address in memory, division by zero or similar error condition, have to be looked upon as internal.
- *hardware vs. software:* While a hardware interruption is routed to the processor via a channel that is effectively implemented in hardware, the software interruption originates from a program command. In the case of software interruptions, a distinction can also be made between *intentional* and *defective* interruptions. Applying these criteria, a keyboard interruption reflects a hardware interruption, a division by zero would be a defective software interruption and the execution of a trapping instruction could be qualified as intentional software interruption.

Nearly every processor architecture uses a different naming and separation of the components that are involved in an interruption processing. Therefore, the following explanation of a typical device interruption process is simplified with respect to the components (esp. the CPU) as well as to the architecture.

³⁹This type of barrier is described in the book The Art of Multiprocessor Programming by Maurice Herlihy and Nir Shavit

⁴⁰[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4.6, page 28, and section 4.4.2.2, page 47.

⁴¹[4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2.2, page 300 ff.

⁴²[18] Tanenbaum and Bos. *Moderne Betriebssysteme*. 2016, section 5.1.5, page 427 ff.

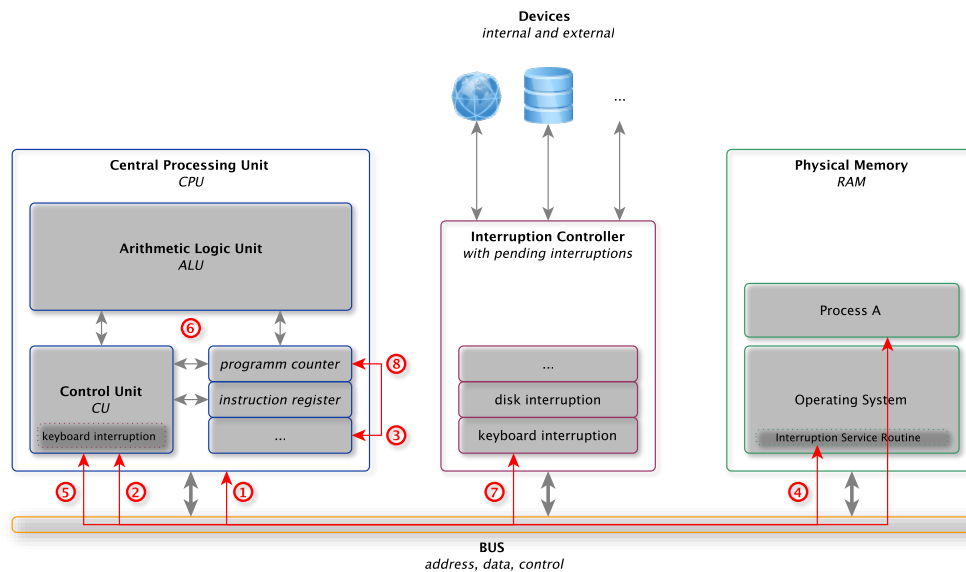


Figure 2.5: simplified interruption process

- (1) The starting situation is illustrated in the figure 2.5 - a process A running on top of an operating system, both loaded into RAM, is executed by the CPU.
- (2) As soon as an interruption (i.c. caused by a keyboard) occurs, the Interruption Controller informs the Control Unit (CU) about it. The CU then stops the execution of the process A.
- (3) To be able to restore the state of process A, the CU saves (on some processor architectures automatically) the programm counter and other registers used by process A.
- (4) Then, the CU checks the cause number of the interruption and retrieves the base address for the according Interruption Service Routine (ISR) defined in the operation systems code.
- (5) After that, the CU loads the instructions of the ISR
- (6) and executes its code until the end of the ISR.
- (7) When the execution of the ISR is finished, the CU informs the Interrupt Controller with an acknowledgement about the processed interruption.
- (8) Last, the CU restores the registers of the process A and continues executing the corresponding instructions. Process A does not even realise the interruption.

2.3.1 Programmable Interrupt Controller

Even modern processor architectures often implement only a few input lines for interruption signals and only support a simple interruption logic. In such cases, an external device, the Programmable Interrupt Controller (PIC), can be attached to the associated processor line(s) to first of all combine different interrupt sources onto one CPU interruption line, but also to allow the assignment of priorities to different kind or groups of interruption causes or to mask different types of interruptions⁴³. A Programmable Interrupt Controller normally features the following registers: (a) an Interruption Request Register (IRR) that specifies the pending interruptions, (b) an In Service Register (ISR) that records the acknowledged but still waiting for an End of Interrupt (EOI) interruptions and (c) an Interrupt Mask Register (IMR) that defines which interrupts are to be ignored and not acknowledged.

The Muen SK makes use of Intel's Advanced Programmable Interrupt Controller (APIC) that is composed of two components - the Local APIC as a part of every physical CPU and the I/O-APIC as a part of the chipset⁴⁴. The most important features of this interruption architecture are⁴⁵:

- local interruption management on a per CPU basis and therefore better performance
- support for inter-processor interrupts (IPI) between Local APICs
- Local APICs provide a high-resolution timer for interval and one-off mode usage
- flexible interruption configuration on a per interruption type basis
- support for Message Signaled Interrupts (MSI)⁴⁶
- priority definition on a per interruption type basis
- interrupt and NMI window exiting feature associated with virtualization⁴⁷
- I/O APIC support multiple interruption input lines
- I/O APIC redirection table to route interruptions to one or more Local APIC(s)

The exact determination of the APIC features required by the Muen SK and the therefore resulting requisites for a target architecture are elaborated in the following sections. But in this context, it can already be stated that:

REQ-8: *A target processor architecture has to provide a mechanism to programmatically handle interruptions.*

⁴³[4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2.2, page 306 ff.

⁴⁴[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.2.4, page 9 f.

⁴⁵https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller, December 21, 2017

⁴⁶https://en.wikipedia.org/wiki/Message_Signaled_Interrupts, December 21, 2017

⁴⁷[14] Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". 2006, page 171 ff.

2.3.2 Interrupts

In the context of the Muen SK, interrupts are defined as external hardware interruptions. As an example, the Muen report mentions a network card that generates an interrupt whenever a data packet is received⁴⁸.

The Muen SK uses the Intel VT-x technology to inform a subject about an external interrupt. An external interrupt request (IRQ) is routed to the in the system policy statically defined subject through the Muen SK that provides a per subject array with up to 32 pending interrupts for delivery. To achieve this routing mechanism, the Muen SK has to enable the I/O APIC and rely on the LAPIC feature to be able to specify not only the physical CPU, that the subject is allocated to, but also the subject itself⁴⁹. To improve the interrupt delivery with respect to performance, the Muen SK also makes use of Intel's virtualization mechanism called interrupt window exiting⁵⁰. Therefore, a target processor architecture has to meet the following requirements:

REQ-9: *A target processor architecture has to provide an interruption handling that guarantees the **exclusive** treatment of interrupts by the separation kernel.*

Another important aspect of Intel's x86 architecture is that it allows to enable or disable interrupts for the VMX root mode. This is done by not setting the IF interrupt flag in the host's FLAGS register. The Muen SK uses this mechanism to simplify the the kernel code and to assure that the Muen SK is not disrupted by external interrupts⁵¹. Therefore, a target processor architecture has to manifest a similar feature:

REQ-10: *A target processor architecture has to provide an enabling and disabling mechanism for (external) interrupts, at least for the execution of the hypervisor code.*

2.3.3 Exceptions and Software Generated Interrupts

In the context of the Muen SK, exceptions are defined as defected software interruptions. This means, that an exception is an interruption generated by the processor itself detecting an error condition during the execution of an instruction. As an example, the division by zero is given. While exceptions denote defected software interruptions, software generated interrupts have to be qualified as intentional software interruptions⁵². As both interruption types are treated similarly, they are subsumed in this section.

⁴⁸[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.3, page 9.

⁴⁹[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.6, page 50 f.

⁵⁰[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.4, page 49.

⁵¹[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.6, page 51.

⁵²[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.3, page 9, section 3.4.4, page 27 f., and section 4.4.7, page 51 f.

First of all, the Muen report distinguishes between exceptions and software generated interrupts that occur in VMX non-root mode (i.e. while executing a subject) and in VMX root mode (i.e. while the Muen SK is executed). As a basic requirement for the Muen SK, the ability to prove the absence of runtime errors is stated. Hence, if an exception (or even less likely a software generated interruption as well as a non maskable interrupt) occurs during the regular execution of the Muen SK in VMX root mode, it would indicate a serious problem in the kernel code and therefore the whole system would be halted.

In VMX non-root mode, there has to be differentiated between native and VM subjects. While VM subjects must implement their own exception handling and hence exceptions and software generated interrupts must not result in a subject exit, native subjects do not react on exceptions but handover the execution to the kernel⁵³ (cf. trap in section 2.3.4).

REQ-11: *A target processor architecture must support a mechanism to enable and disable exceptions and software generated interrupts resulting in an exit of the guest subject.*

In the context of exceptions and software generated interrupts, also system management exceptions (e.g. non maskable interrupts) have to be mentioned. The Muen SK makes sure that this type of interrupts are not handled by the subject itself but result in a subject exit by all means. Therefore:

REQ-12: *A target processor architecture shall provide a mechanism to force system management exceptions to lead to an exit of a guest subject.*

2.3.4 Traps

The term trap, as used by the Muen report, subsumes different kind of interruptions and virtualization techniques that lead to a VM exit. As examples for VM exits, the documentation mentions the execution of a privileged operation or a constrained instruction. The Muen SK uses the VT-x technology to provide the possibility of specifying a per subject trap table in the system policy, whereby all of the VMX basic exit reasons defined by Intel can be configured according to the subjects needs except the following, by the Muen SK internally reserved traps⁵⁴:

- external interrupt (cf. section 2.3.2)
- interrupt window (cf. section 2.3.2)
- VMCALL (cf. section 2.3.5)
- VMX preemption timer expired (cf. section 2.4)

⁵³[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.7, page 51 f.

⁵⁴[2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.5, page 49 f.

Therefore, a virtualization extension or interrupt handling mechanism for a target processor architecture has to meet the following requirement:

REQ-13: *A target processor architecture must be able to differentiate between exit reasons of a guest system and to handle them as specified per subject.*

2.3.5 Events

The Muen SK implements an event mechanism that is used for inter-subject signalization. This means, that a subject is allowed to send an event to another subject as long as this has been granted by an entry in the subject's policy event table.

The implementation of this event mechanism is based on the VMCALL VMX instruction. Hence, when a subject sends an event to a destination subject, it results in a trap into the Muen SK that handles the event according to the system policy. Additionally, an optional inter-processor interrupt (IPI) can be emitted to speed up the inter-core interrupt delivery. If this option is enabled for an interrupt event, an inter-processor interrupt is delivered to the CPU of the destination subject. Finally, this results in the preemption⁵⁵ of the subject, that is executed at the moment on the destination CPU, and therefore the immediate delivery of the event. A target architecture should therefore have the ability to provide a similar mechanism:

REQ-14: *A target processor architecture should provide a technique to fast process interruptions between cores.*

2.4 Timers

In the context of timers, the clock generator has to be mentioned first. In a system, the clock generator is responsible for producing a constant timing signal. This so called clock signal normally corresponds to a frequency generated by a quartz piezo-electric oscillator⁵⁶. This signal is then used by all components of the system to synchronize a circuit's operation, including the timer components. In this documentation, the term „clock“ refers only to this initial output signal. All other periodic signals, that depend on this initial signal and that are mentioned in the context of synchronization, are termed „timer“ (even though in most literature this terms are used interchangeably).

A timer is an integrated circuit that normally signals an interruption after a configurable amount of „time“ (Programmable Interval Timer PIT⁵⁷) or after an overflow of a counter register. There exist many different types of and definitions for timers realized in hardware according to their usage, e.g. pause function timers, one-shot timers, periodic timers, time-slicing timers and watchdog timers. For this study, only

⁵⁵ [https://en.wikipedia.org/wiki/Preemption_\(computing\)](https://en.wikipedia.org/wiki/Preemption_(computing)), December 21, 2017

⁵⁶ https://en.wikipedia.org/wiki/Clock_generator, December 21, 2017

⁵⁷ https://en.wikipedia.org/wiki/Programmable_interval_timer, December 21, 2017

the system timers are important: A system timer is a timer integrated into a hardware component that is responsible for producing a periodic signal used by the whole component. Regardless of the designation and application of a timer, its functionality can be described as a device that uses a high-speed clock input to provide a series of time or count-related interruption signals. As a single counter can only generate short time intervals due to the high-speed frequency of the clock, a technique called cascading can be used with some additional programmable scaling registers to multiply this short time intervals and thereby generating longer time intervals⁵⁸. An alternative to programmable scaling registers is the cascading of multiple timer components. A simple unscaled programmable timer can be described as follows:

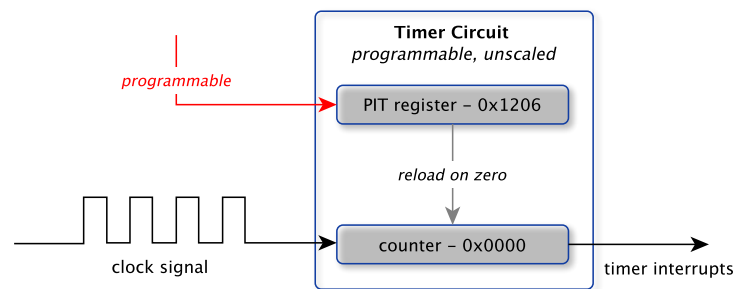


Figure 2.6: timer component

The most important timer used by the Muen SK is the VMX preemption timer in the context of the statically defined scheduling mechanism for subjects running on the same core⁵⁹. This timer provided by Intel's virtualization extension can be set to a specific value according to the time slice definition for the corresponding subject. The subject is then automatically preempted by the processor when the time slice defined in the scheduling plan is over. After that, the Muen SK hands over the execution to the next subject according to the scheduling plan.

REQ-15: *A target processor architecture shall provide a preemptive mechanism on a per subject basis. At least it must provide a timer per core.*

2.5 Device Handling

Basically, there are three possibilities to handle devices⁶⁰. The first possibility (and least reasonable one) is the code or software based device handling. It makes use of a polling mechanism by continuously checking the status register of the desired device. The second possibility is an interrupt based approach explained in the previous sections. The third one is called Direct Memory Access (DMA).

⁵⁸ [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2.2, page 309.

⁵⁹ [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.3, page 47 f.

⁶⁰ [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2, page 300.

This last technique allows attached peripheral devices to directly interact with the main memory over a usually external hardware controller (i.e. DMA controller). The CPU only has to configure the DMA controller at initialization time - after that the controller acts without the usage of the CPU⁶¹.

At the time of its writing, the Muen report declared the device virtualization out of scope⁶². But in the past years the implementation of the Muen SK has been extended and now uses Intel's VT-d Virtualization Technology for Directed I/O to virtualize I/O devices through an IOMMU. The virtualization extension VT-d simplifies the direct assignment of devices to virtual machines in two ways - first, by providing secure direct memory access (DMA) and second, by extending device interrupt remapping functionality. Even though a further evaluation of this topic is out of scope for this study, at least the following requirement can be stated:

REQ-16: *A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for associated interruption and memory features.*

2.6 Floating Point

Modern processor architectures usually implement a so called Floating Point Unit (FPU), a specialized integrated circuit used for floating point calculations. As these floating point calculations often make use of the single instruction multiple data⁶³ technique, the SIMD engine has to be mentioned in this context too. Since the Muen SK does not use either component⁶⁴, there can't be derived any further requirements in this topic area.

2.7 SPARK

While Ada is a general-purpose language supporting the usual features of modern programming languages including built-in support for the design-by-contract paradigm, SPARK is a specialized well-defined subset of Ada designed for the development of high integrity software. Due to these restrictions of the Ada programming language, SPARK has the ability to simplify the application of formal mathematical methods, so that the correctness of the software or other program properties can be guaranteed with mathematics-based assurance.

At the beginning of the Muen project, the development of SPARK 2014 was still ongoing, so that the Muen SK was initially written in SPARK 2005⁶⁵. Within the last years, the Muen developers have

⁶¹ [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2.3, page 309 ff.

⁶² [2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.1.3, page 14.

⁶³ <https://en.wikipedia.org/wiki/SIMD>, December 21, 2017

⁶⁴ cf. Besprechungsnotiz October 23, 2017, and [2] page 41

⁶⁵ [2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, chapter 2, section 2.1.3, page 5.

changed the underlying programming language and are now using SPARK 2014⁶⁶. Since SPARK is a true subset of Ada and compilers ignore the SPARK inherent annotations, every correct SPARK program is a valid Ada program and can therefore be compiled with an existing Ada compiler such as GNAT (part of the GNU compiler collection GCC). Hence, to be able to build the Muen SK, the following requirement has to be fulfilled:

REQ-17: *There must exist a native or cross compiler for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a native or cross compiler with freely available software.*

To fulfil the requirement of a small code base, the Muen SK uses the Ada Zero Footprint Runtime⁶⁷. A Zero Footprint Runtime (ZFP) is a downscaled runtime system (RTS) where only a minimum of supporting code is required. As no unnecessary libraries are introduced into the system, this setup is ideal for critical low level programming. Therefore, to be able to run the Muen SK on an processor architecture other than Intel x86, a ZFP for the targeted architecture has to be available.

REQ-18: *There must exist a Zero Footprint Runtime for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a Zero Footprint Runtime with freely available software.*

2.8 Derived Requirements

The following table summarizes the above derived requirements for a target architecture to be able to run the Muen SK:

number	requirement	topic
REQ-0	The processor architecture has to support 64 bit data-path widths, integer size and memory address widths as well as to be able to execute 32 bit applications.	basics
REQ-1	The target processor architecture must provide a virtualization extension that is capable of running a Type I hypervisor. This requirement includes the hardware assisted support for an additional privilege level and instructions for a simplified switch between this additional and other privilege level.	basics

Table 2.1: requirement summary part one

⁶⁶cf. section kernel, first statement in <https://muen.codelabs.ch/#kernel>, December 21, 2017

⁶⁷[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 4.2, page 41.

number	requirement	topic
REQ-2	The target processor architecture must provide a virtualization extension that supports an automatic handling of guest exits (i.e. traps) and entries. At least, the target processor architecture must provide a support mechanism to completely save and load all the relevant guest state structures.	basics
REQ-3	The target processor architecture shall provide a minimal set of cache management features and an automatic cache clearing feature in the context of virtualization. At least, the target processor architecture must provide a support mechanism to clear caches manually.	memory
REQ-4	The target processor architecture has to provide a Memory Management Unit that supports: (i) memory virtualization on a per subject basis (one page table per subject), (ii) definition of properties and permissions per page table (read/write access, execute disable, caching behaviour), (iii) checking and enforcement of defined properties and permissions, (iv) different page sizes (i.e. large page support), but at least a 4 KB page size.	memory
REQ-5	The target processor architecture must support hardware assisted second level address translation (SLAT).	memory
REQ-6	A multicore target processor architecture has to provide a mechanism to switch off the multithreading mechanism on a per core basis, if multithreading is supported.	memory
REQ-7	A multicore target processor architecture shall provide a barrier synchronization mechanism. At least it must offer an atomic swapping instruction to support the according spinlock implementation.	memory
REQ-8	A target processor architecture has to provide a mechanism to programmatically handle interruptions.	interruption handling
REQ-9	A target processor architecture has to provide an interruption handling that guarantees the exclusive treatment of interrupts by the separation kernel.	interruption handling

Table 2.2: requirement summary part two

number	requirement	topic
REQ-10	A target processor architecture has to provide an enabling and disabling mechanism for (external) interrupts, at least for the execution of the hypervisor code.	interruption handling
REQ-11	A target processor architecture must support a mechanism to enable and disable exceptions and software generated interrupts resulting in an exit of the guest subject.	interruption handling
REQ-12	A target processor architecture shall provide a mechanism to force system management exceptions to lead to an exit of a guest subject.	interruption handling
REQ-13	A target processor architecture must be able to differentiate between exit reasons of a guest system and to handle them as specified per subject.	interruption handling
REQ-14	A target processor architecture should provide a technique to fast process interruptions between cores.	interruption handling
REQ-15	A target processor architecture shall provide a preemptive mechanism on a per subject basis. At least it must provide a timer per core.	timer
REQ-16	A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for associated interruption and memory features.	device handling
REQ-17	There must exist a native or cross compiler for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a native or cross compiler with freely available software.	SPARK
REQ-18	There must exist a Zero Footprint Runtime for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a Zero Footprint Runtime with freely available software.	SPARK

Table 2.3: requirement summary part three

3 ARMv8 Architecture

The Advanced RISC Machines ARM architecture denotes a Reduced Instruction Set Computing RISC¹ microprocessor design from ARM Limited. Unlike the popular Intel processors, ARM Limited does not manufacture the processors itself, but grants design licenses to semiconductor manufacturing companies. Compared to Complex Instruction Set Computing CISC² architectures, the ARM architecture is characterized by a lower number of transistors and as a result lower costs, improved power consumption and less heat generation. Due to the large number of manufacturers and the advantages of this architecture, ARM processors are the most widely used processors in the embedded area. Almost all smartphones, tablets and industrial controllers today use licensed ARM processors³.

The success of ARM-based processors has led to a steady development of the architecture. With the ARMv8-A architecture introduced in 2011, ARM Limited has presented the first 64-bit architecture with a virtualization extension applicable for embedded systems. In the following years, the ARMv8-A architecture was continuously improved with the versions ARMv8.1-A, ARMv8.2-A and ARMv8.3-A⁴. These enhancements to the ARM architecture now allow software developers to port the latest applications implemented for Intel and AMD processors to the ARM architecture as well as to meet the requirements in the progress of the Industry 4.0 context by developing more secure software.

Due to the application field of ARM processors and the licensing strategy of ARM Limited, a large number of so called ARM-based System on Chip (SoC) was developed. An ARM-based SoC corresponds to the combination of an ARM processor as CPU together with the GPU and other peripheral devices on a single chip⁵. The distinction between the processor and the other devices on such a chip is essential for software development - while the architecture of the processor is defined and very well documented by the ARM company, the accessibility of the processor to the peripherals and its control is not predetermined by ARM. Hence, there are a variety of different SoC architectures with different accessibility strategies: from processor controlled (Odroid C2 with amlogic S905 SoC⁶) to VideoCore controlled (Raspberry Pi 3 with Broadcom 2837⁷).

This feasibility study follows a general approach to evaluate the portability of the Muen SK to the ARM architecture. Therefore, this chapter only covers the ARM processor architecture and its capabilities. However, some of the derived requirements from the last chapter are SoC specific and can therefore in this context only be qualified as *IMPLEMENTATION DEFINED*. In the next chapter 4, the Raspberry Pi 3 as the target hardware platform of this study is examined in more detail.

¹cf. https://en.wikipedia.org/wiki/Reduced_instruction_set_computer, December 21, 2017

²cf. https://en.wikipedia.org/wiki/Complex_instruction_set_computer, December 21, 2017

³cf. <https://www.arm.com> and https://en.wikipedia.org/wiki/ARM_architecture, December 21, 2017

⁴cf. <https://developer.arm.com/products/architecture/a-profile>, December 21, 2017

⁵cf. https://en.wikipedia.org/wiki/System_on_a_chip, December 21, 2017

⁶<http://www.hardkernel.com/main/products>, December 21, 2017

⁷<https://www.raspberrypi.org/products/raspberry-pi-3-model-b>, December 21, 2017

3.1 Code Examples

After a thorough review of all the available options to run and verify ARMv8 assembly, it was decided to use the following configuration to test the code snippets mentioned in this chapter:

identifier	description	link
<i>Method</i>	installation as Virtual Machine	VM Ware
<i>Host Operating System</i>	Debian 64-bit 9.2	Debian Download
<i>Toolchain (Cross)</i>	Linaro aarch64-elf cross compiler	Linaro Release
<i>IDE</i>	DS-5 Community Edition Linux64 28rel0	DS-5 IDE
<i>Debugger</i>	DS-5 Community Edition Debugger (integrated into the DS-5 IDE)	DS-5 Debugger
<i>Simulation</i>	ARMv8-A Foundation Model (integrated into DS-5 IDE)	Fast Models

Details on the installation and configuration of the corresponding tools can be found in the respective evaluation case documentation. However, it should be noted that the version of the DS-5 IDE has changed during this project - therefore, the installation process slightly changed compared to this documents. Due to the limitations of the Community Edition of the DS-5 IDE, the code snippets were tested in a minimal environment derived from the official startup example code and only on one processor.

3.1.1 Code Compilation

Principally, three useful compilers are available for compiling assembly code, i.e. the FASMARM Assembler ⁸, the ARM Compiler 6 ⁹ and the assembler of the GCC GNU Compiler Collection ¹⁰.

The FASMARM v1.42 assembler is a free and Open Source cross assembler add-on for the FASM flat assembler. At the beginning of this project, this assembler was used exclusively because it is easy to install, to configure and to use. However, the main disadvantage of the assembler is that it does not support the 64-bit ELF DWARF debugging format ¹¹ and therefore the assembled code cannot be executed on the Fast Model Simulation Debugger provided by the ARM DS-5 Community Edition.

The ARM Compiler 6 is the latest C/C++ Compiler toolchain provided by ARM Limited. It can be used as a standalone tool but it also supports the integration of the Compiler toolchain into the DS-5 Development Studio Professional and Ultimate Edition. As this compiler is not freely available, it was not tested during this project.

⁸cf. <https://arm.flatassembler.net>, December 21, 2017

⁹cf. <https://developer.arm.com/products/software-development-tools/compilers/arm-compiler>, December 21, 2017

¹⁰cf. <https://gcc.gnu.org>, December 21, 2017

¹¹ReadMe section 5, first paragraph - „... For 64-bit code only the binary format is currently supported. ELF64 and PE64 formats have not yet been updated.“; cf. <https://arm.flatassembler.net/ReadMe.txt>, December 21, 2017

The third compiler tested for compiling assembly code is the compiler of the GCC Gnu Compiler Collection. The Gnu Compiler Collection is a freely available compiler suit for the programming languages C, C++, Objective-C, Fortran, Ada and Go published under the Gnu Public License GPL. As part of the C compiler suite, an assembler for the ARMv8 AArch64 architecture is delivered too. The advantages and therefore the decisive reason to work with this compiler are the supported languages (including the Ada GNAT toolchain), the excellent documentation, the ability to generate ARM ADS AXD 64-bit compatible formats for code simulation on the ARM Fast Model and the large number of existing cross compiler binaries. The preferred cross compiler for this project is the Linaro AArch64 ELF cross compiler ¹².

3.1.2 Code Execution and Debugging

There are basically two possibilities available for an informative debugging: on the one hand, one can debug the code over the JTAG interface directly on the target hardware and, on the other hand, the debugger integrated in the DS-5 IDE on a simulated ARMv8 hardware model, the so called Foundation Model, can be used.

To be able to debug the code under consideration directly on the target platform, a JTAG hardware adapter is needed. The JTAG setup was tested with a Segger J-Link Edu Version 10.1 adapter ¹³ and the Raspberry Pi 3. Detailed instructions for such a setup can be found in the corresponding Development Environment Setup evaluation case for the programming language C/C++. The disadvantages of a JTAG debugging in the context of this chapter are the complicated and time consuming wiring as well as the exclusive view of the processor as one always has to test the peculiarities of the hardware too.

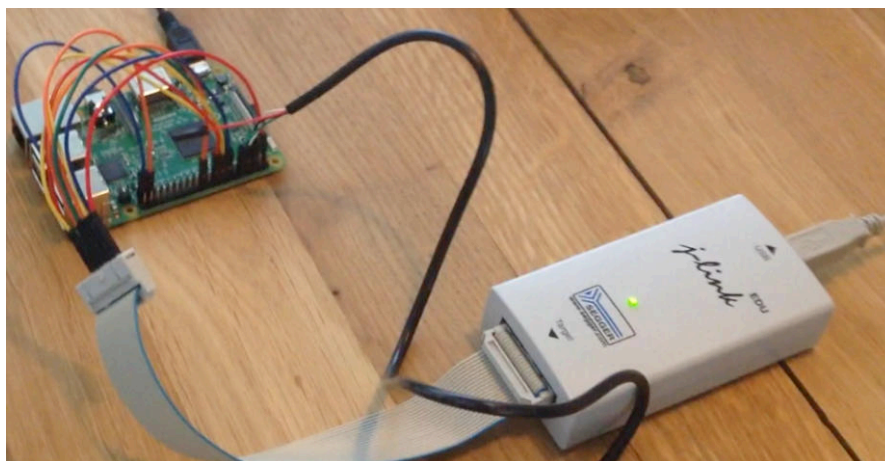


Figure 3.1: JTAG adapter with Raspberry Pi 3

¹²<https://www.linaro.org>, December 21, 2017

¹³<https://www.segger.com/products/debug-probes/j-link/models/j-link-edu>, December 21, 2017

The second alternative using the debugger integrated into the DS-5 IDE was really persuasive. Not only the good documentation provided by ARM but also the clear, informative presentation in the IDE as well as the easy handling of the tools convinced to choose this setup. The only disadvantages are the limitations for the freely available community edition - the code can only be debugged on one core, the implementation defined aspects of a SoC cannot be emulated and some restrictions for peripheral devices have to be accepted ¹⁴.

Feature	Community	Professional	Ultimate
IDE			
DS-5 Eclipse IDE	✓	✓	✓
Processor Support more»			
Arm7	✗	✓	✓
Arm9	✗	✓	✓
Arm11	✗	✓	✓
Cortex-M (Armv6, Armv7, Armv8)	✗	✓	✓
Cortex-R (Armv7)	✗	✓	✓
Cortex-A (Armv7)	Limited to Single-core Cortex-A9 Model	✓	✓
Cortex-A (Armv8), Cortex-R (Armv8)	Limited to Armv8-A Foundation Model	Limited to Armv8-A Foundation Model	✓
Support for cross triggering	✗	✓	✓

Figure 3.2: DS-5 Community Edition restrictions

A first good insight into the ARM developer tools can be gained in the videos published on Youtube ¹⁵. In addition to the standard project view (cf. figure 3.3), the debugger view (cf. figure 3.4) is automatically presented during debugging. In the upper left-hand window one can find the debug controls, that show the limitation to only one core. The command window in the middle of the upper half shows the current exception levels and executed commands including line numbers. The most interesting tab „Registers“ in the upper right window shows the general purpose and other registers whereby the currently used registers are shaded in yellow. In addition to the executed code in the lower left corner, the window on the right-hand side contains information about the memory and the stack(s), if defined.

An important note in this context: In order to be able to execute self-written code in the DS-5 debugger, the compiler command line option

```
aarch64-elf-gcc --specs=aem-ve.specs ...
```

has to be added to load the specification file for the AArch64 baremetal newlib and libgloss appropriate for the foundation model. A good tutorial can be found on the ARM developer page for the DS-5 Community Edition on the tab page „Resources“ ¹⁶.

¹⁴<https://developer.arm.com/products/software-development-tools/ds-5-development-studio>, December 21, 2017

¹⁵https://www.youtube.com/watch?v=_tXWrHD8shs, December 21, 2017

¹⁶<https://developer.arm.com/products/software-development-tools/ds-5-development-studio/resources/tutorials/getting-started-with-ds-5-ce-and-armv8-foundation-platform>, December 21, 2017

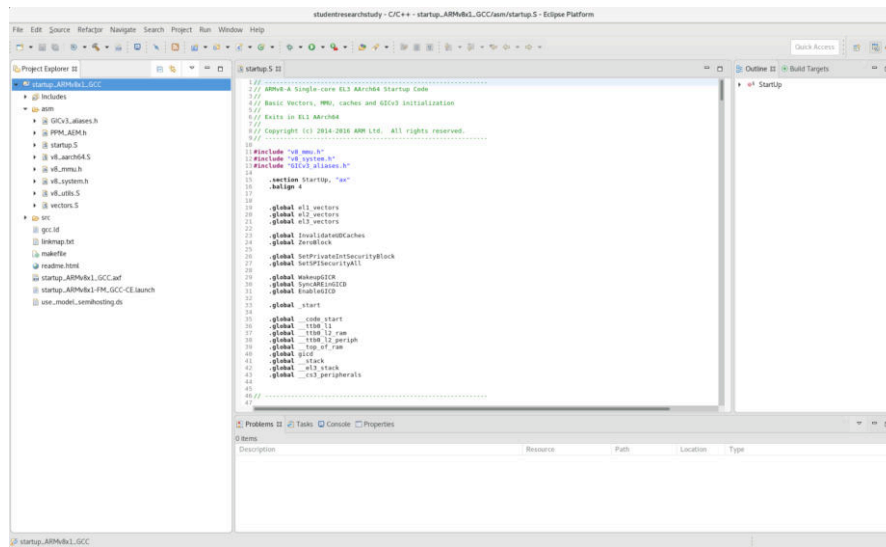


Figure 3.3: DS-5 Community Edition project view

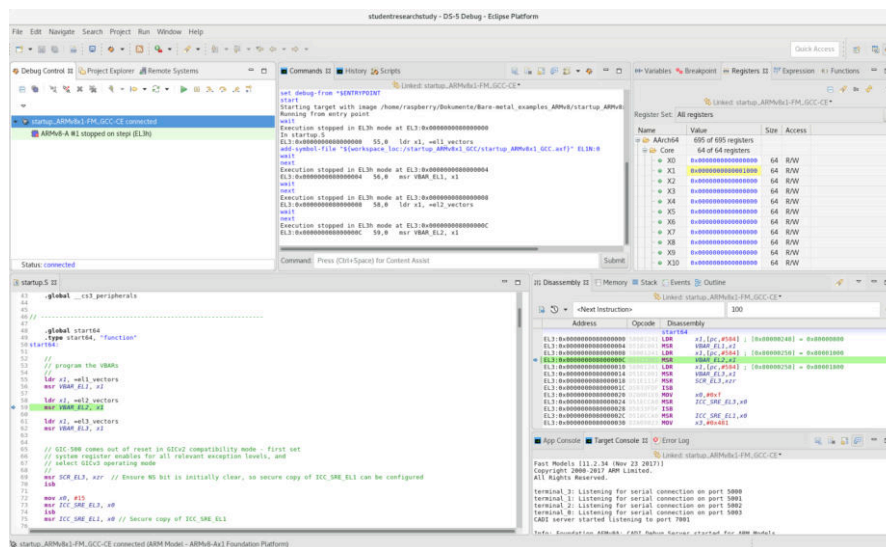


Figure 3.4: DS-5 Community Edition debug view

Last but not least, ARM Limited provides a fully functional code example for a single-core AArch64 Startup sequence with basic vectors, MMU, caches and GICv3 (cf. section 3.5.1) initialization based on the GCC C/C++ Compiler suite including all the necessary page tables and memory layout definitions. This code example is provided with the installation of the DS-5 Community Edition ¹⁷.

¹⁷<https://developer.arm.com/products/.../community-edition>, December 21, 2017

3.2 Fundamentals

3.2.1 Exception Levels

Instead of rings used by the Intel architecture (cf. section 2.1), the ARMv8-A architecture refers to privilege levels as Exception Levels. It is important to note, that, unlike on Intel x86 architecture, code execution at a higher Exception Level (i. e. an Exception Level EL_n with a larger value for n) has *more* privileges than code execution at a lower one¹⁸. With these Exception Levels, the ARMv8 architecture provides a logical separation for software execution privileges. Typically, the Exception Levels can be assigned to the following software examples:

- **EL0** - normal user applications
- **EL1** - operating system kernel (usually described as privileged level execution)
- **EL2** - hypervisor software
- **EL3** - low-level firmware and secure monitor¹⁹

In addition to the horizontal subdivision into Exception Levels, the ARMv8-A architecture also physically partitions the upper three Exception Levels into the *Normal World* and the *Secure World*. With this separation, an ARMv8-A processor supports a secure and a non-secure state and allows an operating system to run in parallel with a so called trusted operating system²⁰. A trusted OS denotes the operating system running in the Secure World and is responsible to provide secure services to the Normal World. Further details on the TrustedZone technology can be found on the official ARM homepage²¹. The following diagram shows the subdivisions as well as the partitions for the AArch64 execution state:

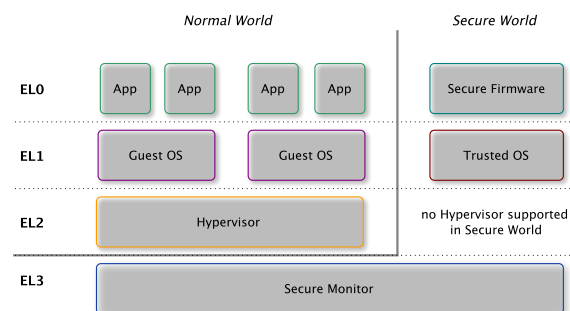


Figure 3.5: ARMv8-A Exception Levels in AArch64

¹⁸[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-1.

¹⁹ARM Trusted Firmware that takes care of the switching between the non-secure and the secure worlds. The code is available as open source on Github, cf. <https://github.com/ARM-software/arm-trusted-firmware>, December 21, 2017

²⁰The interaction (e.g. access rights) between the secure and non-secure world can be defined using the system monitor and corresponding registers (e.g. for physical address spaces in chapter 12, section 12.9 in [8])

²¹<https://www.deepl.com/translator>, December 21, 2017

The only differences between the Exception Levels in AArch64 and the AArch32 execution state are, that in the AArch32 execution state there does not exist an Exception Level 2 in the Secure World and that the privilege levels defined for the ARMv7 architecture are mapped to the Exception Levels accordingly. But as the Muen SK needs to be executed in a 64-bit environment, the details of the Exception Level organisation in AArch32 execution state can be omitted.

Even though, the details for changing the Exception Level depend on the execution state of the processor, it can be generally stated that such a change can only take place during the occurrence of an exception (cf. section 3.5), the returning from an exception (i.e. the `ERET` instruction), a supervisor call or hypervisor call. While changing the Exception Level in AArch32 execution state remains the same as with the ARMv7 architecture²², for the AArch64 execution state the following rules apply²³²⁴:

- (i) *Rule 1:* An exception causes a change of program flow by executing an exception handler function from a predefined vector. Exceptions flow from lower Exception Level to higher ones. That means, that an exception cannot be taken to a lower Exception Level (e.g. EL2 to EL1).
- (ii) *Rule 2:* Exception Handling at EL0 is not possible, i.e. exceptions must be handled at a higher Exception Level than EL0.
- (iii) *Rule 3:* To end an exception handling and return to the previous Exception Level is performed by executing the `ERET`.
- (iv) *Rule 4:* Returning from an exception handler cannot move to higher Exception Levels. Therefore, returning from an exception can stay at the same Exception Level or enter a lower one.
- (v) *Rule 5:* The security state changes according to the rules in the section D1.4 of the ARM Architecture Reference Manual [7].

As a practical example, the procedure for changing Exception Level from EL3 to EL1 by using the `ERET` instruction (returning from an exception) is described in this paragraph²⁵. According to the rules mentioned above, the only possibility to switch to a lower Exception Level is to execute the `ERET` instruction. When performing such an exception return (for this example at EL3), the processor restores the state using the system registers `ELR_EL3` (i.e. the address to return to) and `SPSR_EL3` (i.e. the state to be restored including the targeted Exception Level). These two registers are writeable, thus allowing the desired entry point and state (the Exception Level EL1 for this practical example) to be programmed manually.

²² [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-5 ff.

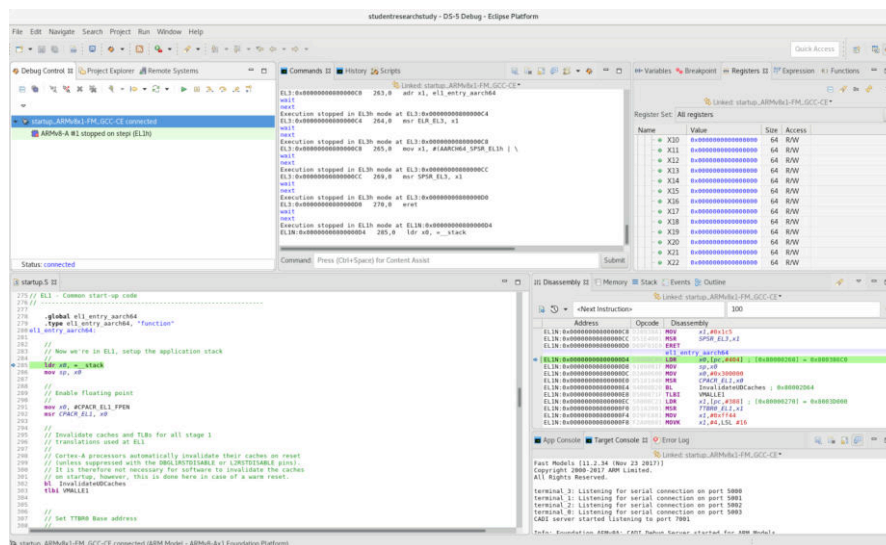
²³ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-7 f.

²⁴ [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.1, page D1-1776.

²⁵ Another example for changing the Exception Level from EL3 to EL2 hypervisor mode can be found on the ARM developer pages in [this](#) discussion, December 21, 2017

- (i) *Step 1:* In this first step, the entry point, i.e. the start address of the code to be executed at EL1, has to be loaded into a general purpose register.
- (ii) *Step 2:* The address from step (i) is then stored in the Exception Link Register `ELR_ELn` of the current Exception Level (i.e. EL3).
- (iii) *Step 3:* After that, the Program Status Register at EL3 `SPSR_EL3` has to be set accordingly²⁶. In the context of the ARMv8 startup code example it has to be noticed, that only a dummy return state for EL1 is loaded. Note, that usually the `SPSR_ELn` register holds the value of the Programm State `PSTATE` before taking the exception²⁷.
- (iv) *Step 4:* Finally, the Exception Return instruction `ERET` has to be executed, using the two registers set in the previous steps for the current Exception Level. When executed, the processor core restores the `PSTATE` from the `SPSR_EL3` register (in this case the Execution Level is set to EL1) and branches to the address held in the `ELR_EL3` register²⁸.

The example can be reproduced with the official startup code of ARM Limited (line 261 to 270) with the debugger included in the DS-5 Community Edition. The result in the debugger view can be found in figure 3.6.



3.2.2 Execution States

The ARMv8 architecture defines two Execution States, i.e. the AArch64 Execution State using 64-bit wide and the AArch32 Execution State using 32-bit wide general purpose registers. While the instruction set and the privilege level mapping in the AArch32 Execution State stays the same as in the ARMv7 architecture, the AArch64 Execution State is organised as shown in figure 3.5 and has a different instruction set A64.

Changing between Execution States on the same level is not possible. That means, the system has to first switch to the higher exception level as shown in the previous section, then perform the requested change of the upper exception level and switch back to the original exception level. Of course, such a change between the Execution States has to meet some rules - the most important one is, that changing to AArch64 Execution State requires switching from a lower exception level to a higher one. The following figure 3.7 summarises this rules stated in the ARM Programmer's Guide²⁹:

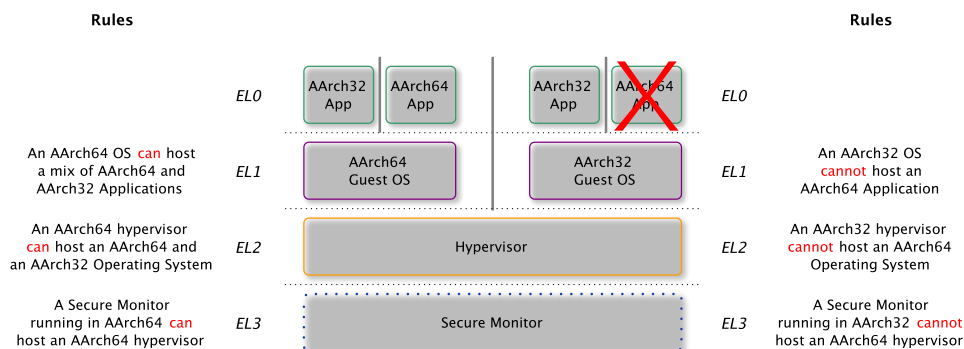


Figure 3.7: ARMv8-A Execution States rules

An example of a correct Execution State change would be an application running in a 32-bit Execution State at EL0 on a 64-bit Operating System executing at EL1 and a second application, that needs to be executed in a 64-bit execution state at EL0, on the same Operating System. In such a case, the 32-bit application can change to the OS exception level in AArch64 execution state by calling the Supervisor Call instruction or by receiving an interrupt. Then the OS can change the execution state of the exception level EL0 to AArch64 and switch back to EL0.

The two most important limitations in the context of Execution States are that it is not possible to check the execution state of the actual code running on a specific exception level but only for higher exception levels³⁰ and that code running at EL3 cannot take an exception to a higher exception level. Therefore, code executing at EL3 cannot change its execution state, except by going through a reset³¹.

²⁹ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-8 f.

³⁰ cf. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16146.html>, December 21, 2017

³¹ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-9.

3.2.3 Startup and Reset

The ARM documentation refers to the startup, i.e. powering on the CPU, as cold reset³². A processor based on the ARMv8 architecture always starts execution at the highest exception level, provided that the SoC manufacturer does not apply any additional firmware code to the boot process. In contrast, the Execution State, in which a processor is running immediately after powering it up, is *IMPLEMENTATION DEFINED*³³. This means that the SoC manufacturer defines this explicitly with a hardware based signal being either logic zero or logic one as an input to the corresponding AA64nAA32 pin of the processor.

As already mentioned, code executing at EL3 can only change its execution state by going through a so called warm reset³⁴. Every core has its own reset input and executes the according exception immediately after their reset. In addition, this exception cannot be masked³⁵. While the execution state after a warm reset is software defined by setting the AA64 bit in the RMR_EL3 register, the reset vector for the highest Exception Level (i.e. the location of the instruction that the ARM processor jumps to when an exception is raised) is again *IMPLEMENTATION DEFINED*³⁶.

Further details on resetting an ARMv8 processor can be found in the ARM Architecture Reference Manual³⁷ as well as in the processors Technical Reference Manuals³⁸.

The first requirement (cf. REQ-0 in section 2) for porting the Muen SK is that the target processor architecture supports a 64-bit execution state. According to the previously explained mechanism, the following qualification can be stated:

REQ-0 - IMPLEMENTATION DEFINED: *The ARMv8 architecture principally supports a 64-bit execution mode. But as the initial execution state as well as the reset vector are defined by the manufacturer of the specific SoC, the fulfilment of this requirement can only be qualified on the basis of the target hardware.*

³²[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.9, page D1-1795.

³³<https://community.arm.com/processors/f/discussions/2874/aarch32-in-armv8> and <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16239.html>, December 21, 2017

³⁴[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.9, page D1-1795.

³⁵[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, page 10-2.

³⁶[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D7.2.85, page D1-2448.

³⁷[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.9.2, page D1-1797 f., provides for example a code sequence to request a warm reset, followed by a pseudocode description in section D1.9.3.

³⁸[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2016, chapter 2, section 2.3.3, page 2-14, and chapter 4, section 4.3.76, page 4-114 as well as appendix A.3.

3.3 Virtualization Basics

Unlike Intel's VT technology, the ARMv8-A Virtualization Extension consists of a number of additional extensions to existing ARM architecture technologies. Accordingly, only one of the ARM documents mentioned above contains a short section dedicated to virtualization³⁹. However, the ARM Developer Community provides a summarising document on virtualization⁴⁰.

Two of the main features of the ARMv8-A Virtualization Extension are a dedicated Exception Level EL2 for the hypervisor code (cf. section 3.2.1), support for trapping exceptions that change the core context or state and an additional exception type generated by the Hypervisor Call instruction `HVC` with a 16-bit payload targeting the exception level EL2⁴¹. These features are explicitly intended for the implementation of a type I hypervisor⁴².

REQ-1 - FULFILLED: *The ARMv8-A architecture explicitly provides the demanded mechanisms to run a type I hypervisor.*

With respect to the second requirement of the Muen SK in the area of virtualization (cf. section 2.1), it can in advance be stated that the ARM virtualization technology does not support any automatic storing or loading of the guest's state. On the contrary, the hypervisor code has to load both its and the guest's context completely into memory or from memory respectively when performing a context switch. At least, the ARMv8 architecture supports a performance optimized possibility for handling the corresponding registers with the Store and Load Pair instructions⁴³. Depending on the guest system, the hypervisor and the specific processor type (e.g. ARMv8 Cortex-A53) as well as the current execution state and the exception level, the following registers could belong to the context and have to be treated accordingly:

- **System Registers:** This category of registers includes different counter, physical timer, MMU, second level address translation and cache registers as well as the Saved Program Status Register `SPSR_ELn`. An overview can be found in the ARM Programmer's Guide⁴⁴.
- **Special Purpose Registers:** The two most important registers of this category are the Stack Pointer Register `SP_ELn` and the special exception return registers. A list of any registers to be stored can be found in the ARM Architecture Reference Manual⁴⁵.

³⁹[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.5, page D1-1782.

⁴⁰[6] n.a. *AArch64 Virtualization*. 2017.

⁴¹[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, page 10-2.

⁴²[6] n.a. *AArch64 Virtualization*. 2017, chapter 1, page 4.

⁴³[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C3.2.2 f., page C3-161 ff.

⁴⁴[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.3, page 4-7 ff.

⁴⁵[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C5.2, page C5-336 ff.

- *General Purpose Registers:* Even though there exist some guidelines for the usage of the general purpose registers, the hypervisor has to store and load all these register to ensure a complete handling of the guest's context. As the Muen SK also supports Virtual Machines executing in a 32-bit environment, it is important to store and load the banked registers too. Banked registers are special purpose registers for exceptions in the AArch32 execution state that are stored in the upper general purpose registers to reduce latency for exception handling⁴⁶.
- *Floating Point and NEON Registers:* If enabled, the SIMD and floating point registers have to be stored and loaded as well⁴⁷.
- *General Interrupt Registers:* If supported by the SoC and enabled by the hypervisor, the according GICD registers have to be considered. Therefore, all pending and active states of private interrupts on the core have to be handled too.
- *Generic and Virtual Timer Registers:* In the case of guests using virtual timers, the timer registers must be saved and restored so that they generate interrupts at the expected intervals.

The physical memory, that is assigned to a guest, does not have to be handled. By using more than one stage of memory translation, the physical memory that the guest uses stays private and distinct from any others.

To get an impression of how the storing and loading of general purpose registers and system registers could look like, the following code snippet presents two examples:

```

; storing and loading the two general purpose registers x0 and x1 in AArch64 execution state
    stp    x0, x1, [memory_address]
    ...
    ldp    x0, x1, [memory_address]

; storing and loading system registers for exception Level 1
    mrs    x2, ESR_EL1
    mrs    x3, ELR_EL1
    stp    x2, x3, [memory_address]
    ...
    ldp    x02 x3, [memory_address]
    msr    ESR_EL1, x2
    msr    ELR_EL1, x3
  
```

Taking the above explanations into account, the following can be stated with respect to the requirement demanded by Muen SK:

REQ-2 - FULFILLED: *Even though a context switch has to be implemented manually in the hypervisor code, it is possible to save and restore all the required registers of a guests context. Therefore, this requirement is qualified as fulfilled by the ARMv8-A architecture.*

⁴⁶[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.5.1, page 4-13 ff.

⁴⁷[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.6, page 4-17.

3.4 Memory

Normally, processors implementing the ARMv8-A architecture have two or more levels of cache. These are usually organized in such a way that one Level 1 cache per core with different areas for instructions and data is available, one unified level 2 cache is shared by two or more cores and an external level 3 cache is used by the entire cluster. The Main Memory can be accessed over the internal bus⁴⁸.

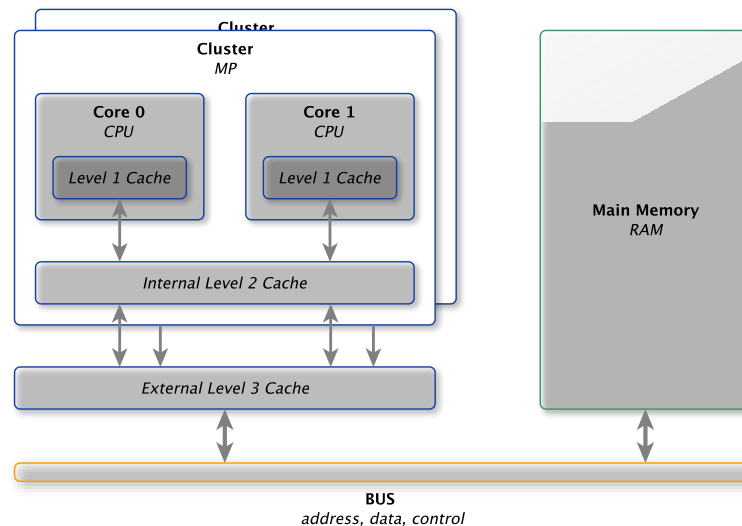


Figure 3.8: ARMv8-A standard memory organisation

3.4.1 Caches

The concrete implementation of the caching structures is not defined in more detail by the ARMv8-A architecture. The only requirement in this context is that the level 1 cache must always be designed as a set of associative caches. This type of cache divides the corresponding memory area into a certain number of equally-sized pieces, called ways. The number of such ways depends on the specific processor architecture - e.g. the ARMv8 Cortex-A53 uses a 2-way set associative instruction cache. Also not defined for the Level 1 cache is the cache addressing mode, i. e. whether a virtual address is first converted into a physical address and then a cash lookup is performed (Physically Indexed Physically Tagged PIPT) or whether the virtual address and the cache lookup are performed in parallel and finally the correctness of the found cache entry is checked against the physical address (Virtually Indexed Physically Tagged VIPT)⁴⁹. To continue the example of the last paragraph, the Cortex-A53 MPCore instruction cache (level 1) uses Virtually Indexed Physically Tagged (VIPT) addressing mode⁵⁰.

⁴⁸[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 11, page 11-1 ff.

⁴⁹<https://www.youtube.com/watch?v=3sX5obQCHNA>, December 21, 2017

⁵⁰[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2016, chapter 2, section 2.1.1, page 2-2.

The organisation and structure of the remaining cache levels is left to the manufacturers of the respective SoC. However, the ARMv8-A architecture establishes some rules in the form of policies and specifies a minimum set of cache maintenance functions for the level 1 cache:

- *Cache Policies:* There exist two categories of policies for caching structures, the allocation and the update policies. The allocation policies are Write Allocation (WA), i.e. a cache line is allocated on a write miss, and Read Allocation (RA), i.e. a cache line is allocated on a read miss. The update policies consist of the Write Back (WB), i.e. a write updates the cache only and marks the cache line as dirty, and the Write Through (WT), i.e. a write updates both the cache and the external memory system. Additionally, the ARMv8-A architecture provides some preload hint instruction. If a cache structure supports one of this features is *IMPLEMENTATION DEFINED* by the manufacturer. But in contrast to other implementation defined aspects of the caches, the support of this features must be set in the Cache Size ID Register of the processor⁵¹.
- *Cache Maintenance:* The ARMv8-A architecture demands three different ways to clean or invalidate the level 1 cache - (a) invalidation of a cache or cache line, i.e. to clear it of data by clearing the valid bit; (b) cleaning a cache or cache line, i.e. writing the contents of cache lines, that are marked as dirty, out to the next level of cache or to main memory and clearing the dirty bits in the cache line; (c) zeroing, i.e. zero a block of memory within the cache (only for data cache). All three operations must either be applicable to the entire cache (mandatory for instruction cache only) or can be applied based on a virtual address, a set index or a way number. In addition to a list of all operations, the ARM Programmer's Guide also contains some code examples for cache handling⁵². It should be noted that after the corresponding cache operations, a data or instruction synchronisation barrier always has to be called to apply the cache operations that are otherwise executed in any relative order.

According to the explanations above and compared to Intel's x86 cache management, the following qualification can be stated:

REQ-3 - FULFILLED: *Since the cache maintenance of the Intel architecture seems to be quite similar to the one of the ARMv8-A architecture and in particular a cache invalidation can explicitly be performed, this requirement has to be seen as fulfilled by the ARMv8-A architecture.*

⁵¹[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2016, chapter 4, section 4.3.22, page 4-42 f.

⁵²[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 11, section 11.5, page 11-13 ff.

3.4.2 Memory Management

The ARMv8-A architecture provides one Memory Management Unit (MMU) per core. In addition to the transparent translation of virtual addresses, the MMU also controls and enforces memory access permissions, memory ordering and cache policies for each memory region. Every Exception Level EL3 to EL1 has its own virtual address space⁵³.

The official startup code of ARM Limited also provides some code for setting up the MMU and the translation tables. In the DS-5 debugger, the code can either be traced step by step or the MMU setup result can be viewed directly in the debugger's MMU view. For the second alternative, the debugger settings must first be adjusted in order to be able to start debugging directly in the main method. To do so, the debugger must be switched to debug from symbol main in the Debug Configurations. As soon as the debugger stops at the corresponding breakpoint, the MMU view can be opened with Windows → Show View → MMU. This view contains a top-level view of the virtual memory layout (cf. figure 3.9) as well as the associated translation tables (cf. figure 3.10).

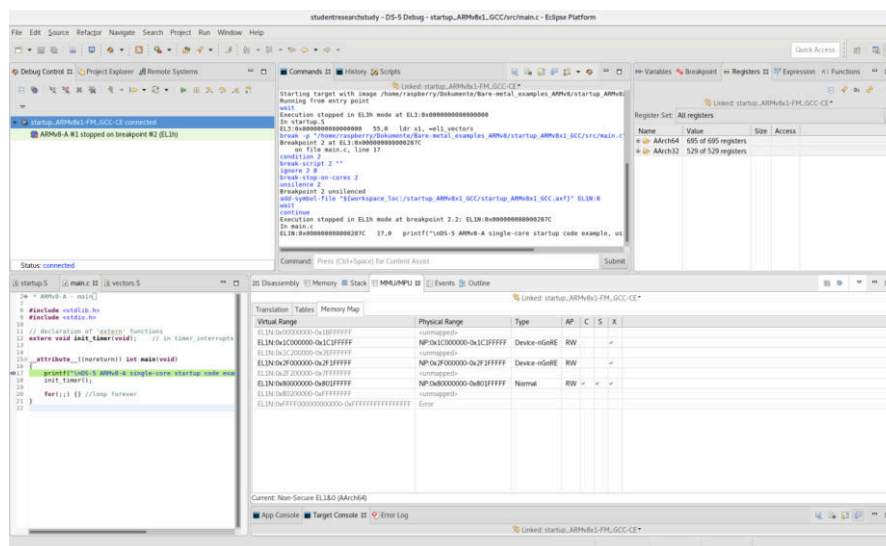


Figure 3.9: DS-5 Debugger MMU memory map

As already mentioned, the support for cache policies is implementation defined. If a SoC provides this feature the according attributes can be set in the translation table entries as defined in the Memory Attribute Indirection Register MAIR. In contrast to caching, the access permissions controlled through the translation table entries are enforced by the MMU. The access permissions can therefore be set separately on a per exception level basis. The ARMv8-A architecture defines three different types of access permissions - readable, writeable and executable. All possible combinations for a specific

⁵³[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, page 12-1.

exception level are listed in the ARMv8 Programmer's Guide⁵⁴ and the details on the registers, that have to be set accordingly, can be found in the ARMv8 Architecture Reference Manual⁵⁵.

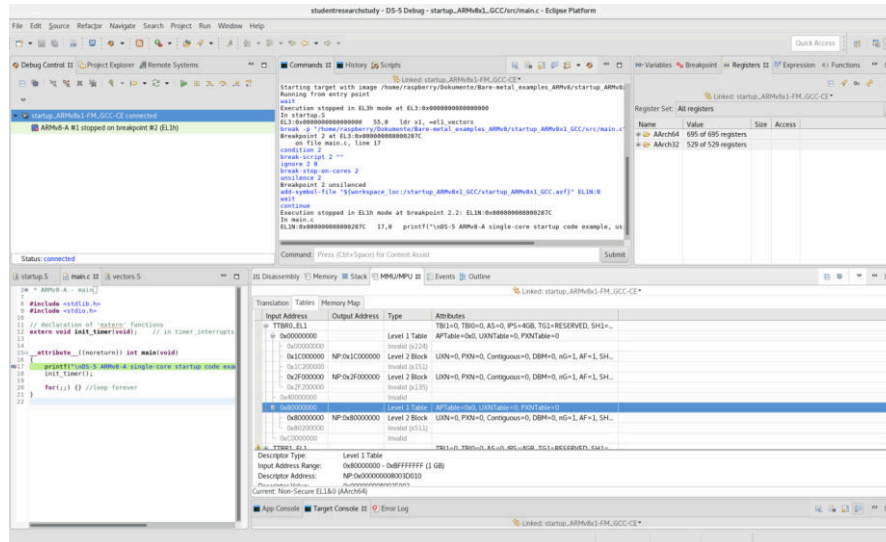


Figure 3.10: DS-5 Debugger MMU translation tables

The ARMv8-A architecture supports two different translation table formats for the AArch32 execution state, i.e. a long descriptor format with Large Physical Address Extension (LPAE) and a short descriptor format. In the AArch64 execution state, however, only the long descriptor format is available, that allows addressing with up to 48-bits. The remaining bits 63:48 of the 64-bit virtual address are used for selecting one of two registers containing the base address of the translation table and optionally the upper 8-bits can be used for tagging the virtual address⁵⁶. The ARMv8-A architecture supports up to three levels of translation tables with granule sizes of 4KB, 16KB and 64KB. It is implementation defined, which of the three sizes actually are supported by a processor. However, processors of the Cortex-A53 series must support all three formats. The addressable memory areas and sizes resulting from the different combinations of page size and translation level can be found in the list provided by the ARMv8 Programmer's Guide⁵⁷.

The Translation Lookaside Buffer (TLB) is used as a cache of recently accessed page translations (cf. section 2.2.2). However, a ARMv8-A TLB can not only store and look up physical and virtual addresses, but is also able to handle attributes such as memory types, cache policies and access

⁵⁴[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.7, page 12-23 f.

⁵⁵[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, e.g. executable regions at EL0 and EL1 in chapter D, section D7.2.88, page D7-2456 ff.

⁵⁶The ARMv8-A architecture does not specify or mandate a specific use case for tagged addressing. A use case example can be found in chapter 12, section 12.5.1, page 12-18, of the ARMv8-A Programmer's Guide [8]

⁵⁷[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.4, page 12-14 ff.

rights. In addition and in the context of virtualization, the TLB also stores the Address Space ID (ASID) and the Virtual Machine ID (VMID). Enabling and disabling as well as a minimal TLB maintenance are also supported. This means that TLB entries can be invalidated using the VMID, the virtual address or a specific exception level. The ARMv8 Programmer's Guide provides some code examples for the TLB maintenance⁵⁸ and details for the ARMv8-A processor in the Technical Reference Manual⁵⁹.

According to the above explanations, the requirements in the context of memory management stated by the Muen SK can be judged as follows:

REQ-4 - FULFILLED: *The ARMv8 architecture provides a Memory Management Unit per core with the following features:*

- (i) With the possibilities of setting up translation tables on a per exception level basis and defining the according base addresses in different registers, the ARMv8 architecture meets this requirement.*
- (ii) As the ARMv8 architecture provides access permissions controlled through the translation table entries, this requirement can also be rated as fulfilled.*
- (iii) The MMU provided with the ARMv8 architecture has to enforce the access permissions and hence must also be able to check them. This requirement can therefore be qualified as fulfilled.*
- (iv) Even though the supported page sizes are implementation defined by the processor specification, all of the ARMv8-A processor series support at least the 4KB sizes. Therefore, this requirement is met too.*

3.4.3 Advanced Memory Virtualization

To be able to run complex virtual machines, the Muen SK relies on the Second Level address translation provided by Intel's EPT technology. The ARMv8-A Virtualization Extension explicitly provides a similar mechanism for nested page tables to isolate the guest operating systems⁶⁰.

Using the ARMv8 Virtualization Extension, the hypervisor is responsible for both its own memory management and that of the guest OS. In a first step, the MMU of the exception level EL2 with the corresponding hypervisor vector tables has to be configured to translate the virtual addresses of the hypervisor correctly. In a second step, the hypervisor must set up and manage the second level address translation mechanism for each virtual machine by enabling the ARMv8-A SLAT mechanism and setting up the corresponding translation tables⁶¹. A correctly applied SLAT then translates the interme-

⁵⁸ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.1, page 12-5.

⁵⁹ [9] e.g. chapter 4, section 4.2.6, page 4-7 f.

⁶⁰ [6] n.a. *AArch64 Virtualization*. 2017, chapter 1, page 5.

⁶¹ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.6, page 12-20.

diate physical memory addresses of the VM to physical memory addresses. The exception handling of aborts during SLAT address translations has to be done by the hypervisor on exception level EL2.

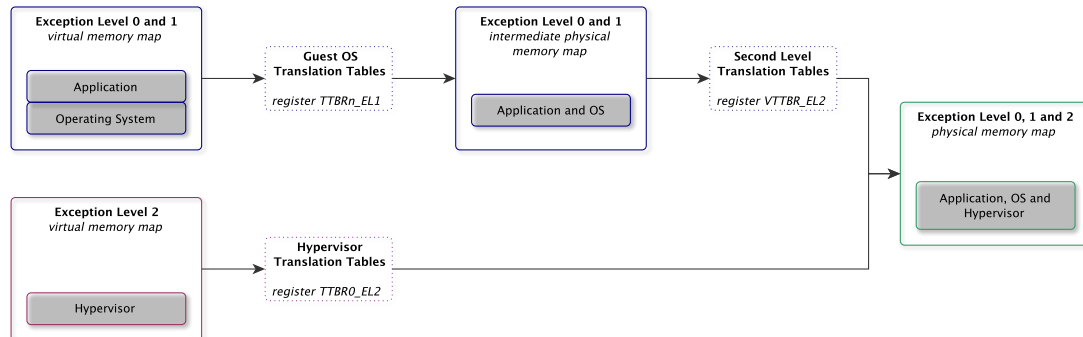


Figure 3.11: ARMv8-A Second Level Address Translation

Code examples for enabling the ARMv8 Second Level Address Translation can be found in the AArch64 Virtualization documentation provided by ARM Limited⁶². Additionally, two practical examples of the usage of nested page tables in the context of a separation kernel can be found in the Phidias hypervisor code⁶³ written by Jan Nordholz⁶⁴ and the HASPOC source code⁶⁵ by Vinnova⁶⁶. Therefore, the following can be stated:

REQ-5 - FULFILLED: The ARMv8-A architecture provides a Second Level Address Translation mechanism and hence meets this requirement.

3.4.4 Multicore Environment

Neither the ARMv8 Cortex-A57 nor Cortex-A53 are Simultaneous Multithreading (SMT) microarchitectures, so at any time there is only one thread executing on one core. As there is not any multithreading support for all currently used processors of the ARMv8-A architecture, the following requirement is always fulfilled.

REQ-6 - FULFILLED: fulfilled per definition

To synchronise the execution in multicore environment, the Muen SK implements a barrier realized with a spinlock using the atomic XCHG processor swapping instruction. The ARMv8-A architecture provides

⁶²[6] n.a. AArch64 Virtualization. 2017, chapter 2, section 2.1, page 7 f.

⁶³<http://phidias-hypervisor.de/repos/core.git>, December 21, 2017

⁶⁴[15] Nordholz. Design and Provability of a Statically Configurable Hypervisor. 2017, chapter 4, section 4.5, page 30.

⁶⁵<https://haspoc.sics.se/source.html>, December 21, 2017

⁶⁶<https://www.vinnova.se/en>, December 21, 2017

some synchronisation primitives that can be used to implement such a barrier⁶⁷. As an example, the Phidias hypervisor implements a spinlock in its assembler file `lock.S`⁶⁸.

REQ-7 - FULFILLED: *The ARMv8 architecture provides different synchronisation primitives to fulfil this requirement.*

3.5 Exception Handling

As already mentioned in the last chapter 2, the various processor architectures and the corresponding literature use different terms (e. g. exception, interrupt, signal, event) for the temporary interruption of a running process by an interruption cause. In the ARM terminology, such an interruption is referred to as an exception. The ARM documentation defines an exception as a condition or system event that requires some action by privileged software (i.e. an exception handler) to ensure the continuous functioning of the system and differentiates between the following four types of exceptions - interrupts, aborts, resets and exception generating instructions.

The exception handling is about the same for all types of exceptions. As soon as an event occurs that causes an exception, the processor hardware automatically performs the following actions:

- (i) *Update Processor State:* The processor automatically stores the processor state `PSTATE` into the System Processor State Register `SPSR_ELn` of the exception level where the exception is taken. That means - if an exception occurs at `EL0` it is taken to `EL1` (as long as there is not any hypervisor at `EL2` and the exception handling is set to be done by the next higher exception level) and therefore the processor state would be stored to `SPSR_EL1`.
- (ii) *Store Return Address:* In a second step, the processor stores the return address to be used at the end of the exception into the register `ELR_ELn` of the exception level (again) where the exception is taken.
- (iii) *Exception Syndrome:* After storing the return address, the processor writes all the information needed to allow the exception handler to determine the reason for the exception to the so called Exception Syndrome Register `ESR_ELn`. Note, that this register is updated only for synchronous and SError exceptions - status informations on (external) interrupts (i.e. IRQ or FIQ, cf. section 3.5.1) have to be generated and handled by an external interrupt controller (preferable a GIC, cf. section 3.5.6).
- (iv) *Exception Handler:* The next action, that the processor performs, is branching to a vector table that contains entries for each exception type. Each exception level has its own exception vector table containing up to 16 instructions in AArch64 execution level to handle and eventually branch

⁶⁷ [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter B, section B2.9, page B2-121.

⁶⁸ <http://phidias-hypervisor.de/repos/core.git>, December 21, 2017

to a more sophisticated exception handler. A detailed description of such an exception table used in AArch64 execution state can be found in the ARMv8 Programmer's Guide⁶⁹. Warning: even though the described registers are automatically updated, they are not automatically stored to memory when the exception level is changed within the exception handler. A change of the exception level has to be implemented manually, as described in section 3.3.

- (v) *Returning and Restoring:* As soon as the exception handler is done and calls the `ERET` instruction, the processor restores the processor state of the application, in which the exception occurred, according to the state values stored in the `SPSR_ELn` register. After completion, the application continues its normal program flow at the location stored in `ELR_ELn`.

The following section describes the registers used to handle exceptions. In addition, the ARMv8-A Virtualization Extension provides a separate register `HCR_EL2` that allows a hypervisor to handle all exceptions by routing or trapping them all to the exception level EL2. A detailed view of this register with explanations to the settable bit positions can be found in the ARM AArch64 Virtualization documentation⁷⁰ as well as in the ARM Technical Reference Manual⁷¹.

Another feature in the context of virtualization provided by the ARMv8-A architecture are virtual exceptions. If the hypervisor is given full responsibility for handling exceptions, it can forward virtual exceptions to its guest systems. The ARMv8-A architecture supports the three exception types: Virtual SError, Virtual IRQ and Virtual FIQ. Further information can be found in the AArch64 Virtualization documentation⁷².

Taking into account the explanation on exception handling and, in particular, the features of the ARM Virtualization Extension, it can be stated:

REQ-9 - FULFILLED: *The ARMv8-A Virtualization Extension explicitly provides interruption handling that guarantees the **exclusive** treatment of interrupts by the hypervisor. Therefore, this requirement is fulfilled.*

In this context, it is also worth mentioning that the ARMv8-A architecture automatically masks all external interrupts after an exception is taken to an upper exception level. However, the exception handler can explicitly allow nested exceptions. The ARM Programmer's Guide contains some more details including a code example⁷³.

⁶⁹ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, section 10.4, page 10-12.

⁷⁰ [6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.4, page 9 f.

⁷¹ [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D7.2.34, page D7-2302.

⁷² [6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.6, page 10 f.

⁷³ [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, section 10.5, page 10-14.

3.5.1 Interrupts

The ARMv8 architecture refers to external, asynchronous interruptions as interrupts and defines two different types - the interrupt request IRQ and the fast interrupt request FIQ. An FIQ is just a higher priority interrupt request that is handled „faster“ by disabling IRQ and other FIQ handlers during its exception handling⁷⁴. Both interrupt types are physical signals to the core that are usually connected to an external interrupt controller. Since all asynchronous exceptions can principally be masked, also IRQ and FIQ can be handled accordingly by setting the DAIF exception mask bits in the SPSR_ELn register⁷⁵. However, a General Interrupt Controller GIC is required for further control of interrupts (cf. section 3.5.6).

As the ARMv8-A architecture provides an exception handling register on a per exception level basis (including EL2 for running the hypervisor), the according requirement can be qualified as follows:

REQ-10 - FULFILLED: *The ARMv8-A architecture provides an enabling and disabling mechanism for asynchronous, external interrupts for every exception level and therefore fulfils this requirement.*

3.5.2 SErrors

Another asynchronous exception type is the System Error (SError). This type of exception can have a number of possible causes depending on the SoC and the processor implementation, because in all of the Cortex-A5x processor series there is a separate physical signal to the core specified for the SError. The most common cause for an SError are asynchronous data aborts⁷⁶. An example would be a mistake in a translation table that marks a ROM as read/write. If the corresponding memory is also marked as write-back cacheable, an attempt to write to the address region would initially go into the cache. At some point later the cache line(s) will get evicted, trigger a write-back of the dirty data and the memory system returns a fault (write to read-only slave), which is classed as an asynchronous SError. As already mentioned, all asynchronous exceptions can be masked and the same applies for the SError (cf. section 3.5.1).

3.5.3 Aborts

In the ARMv8 terminology, an abort is a synchronous exception generated either on a failed instruction fetch (instruction aborts) or a failed data access (data aborts). As synchronous exceptions cannot be masked, they have to be handled as described above. Further information on synchronous exception handling can be found in the ARM Technical Reference Manual⁷⁷.

⁷⁴In AArch32 execution state, the FIQ has its own set of banked registers, cf. also 3.3)

⁷⁵[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.14.2, page D1-1836 ff.

⁷⁶[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, section 10.2, page 10-7.

⁷⁷[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.13, page D1-1826.

3.5.4 Exception Generating Instructions

The execution of certain instructions can generate exceptions. On the one hand, this includes all requests for software running at a higher exception level, i.e. the Supervisor Call **SVC**, Hypervisor Call **HVC** and Secure Monitor Call **SMC**. On the other hand, the exception handling can also be configured in such a way that various other instructions are disabled or cause a trap exception. As an example, cache maintenance instructions can be trapped to EL1 from EL0 by setting the according bit `SCTLR_EL1.UCI` in the System Control Register at EL1. The ARM Technical Reference Manual provides a complete section on all possible modifications and adjustments of the exception handling with respect to exception generating instructions⁷⁸.

As already mentioned, all exceptions can be trapped or routed to a hypervisor running at exception level EL2 by selecting the according bits in the Hypervisor Control Register. In addition, the ARM Virtualization Extension also provides a mechanism for trapping certain instructions that are often used in the context of virtualization, i.e. access to virtual memory control registers, certain system instructions (mostly maintenance instructions for caches), access to the Auxiliary Control register etc.⁷⁹. When an instruction has trapped, the hypervisor code can read the Exception Syndrome Register `ESR_EL2` to obtain the necessary information about the trapped instruction.

Due to the combination of the ARM Virtualization Extension and the handling of exception generating instructions, the following requirements of the Muen SK can be considered fulfilled:

REQ-11 - FULFILLED: *The ARMv8-A architecture supports the configuration of exception generating instructions resulting in an exit of the guest subject and therefore fulfils this requirement.*

REQ-13 - FULFILLED: *This requirement only demands that a target architecture can distinguish between the four exit reasons used by the Muen SK⁸⁰. This means in particular that the Muen SK does not require detailed status information regarding external interrupts in the context of a guest exit. Therefore, even though the exact state of an external interrupt can only be determined using a General Interrupt Controller GIC, the ARMv8-A architecture fulfils this requirement as a hypervisor can read the demanded four different reasons of a guest exit from the Exception Syndrome Register `ESR_EL2`.*

⁷⁸[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D.1.15, page D1-1842.

⁷⁹[6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.5, page 10.

⁸⁰[2] Buerki and Rueeggsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, chapter 4, section 4.4.5, page 49 f.

3.5.5 Resets

The ARMv8-A architecture does not support non-maskable interrupts⁸¹. As already described in section 3.2.3, reset exceptions cannot be masked and hence are the only non maskable exceptions. Since every reset exception is guaranteed to be executed by the core receiving it, it can be stated that:

REQ-12 - FULFILLED: *The only non maskable interrupt (NMI) not only leads to an exit of a guest subject but also to a restart of the core from EL3. Therefore, this requirement can be qualified as fulfilled.*

3.5.6 Generic Interrupt Controller

The Muen SK relies on the I/O APIC and LAPIC mechanism provided by the Intel x86 architecture (cf. chapter Muen, section 2.3.2). The ARMv8-A architecture implements a similar technology, called Generic Interrupt Controller (GIC), based on an internal GIC CPU Interface (corresponds conceptually to the LAPIC) and an external GIC Distributor (corresponds conceptually to the I/O APIC). This mechanism not only supports routing of software generated, private and shared peripheral interrupts between cores in a multicore environment but also the routing of external interrupts to (an) individual core(s). Furthermore, it enables software to mask, enable and disable interrupts, to prioritise individual sources and to generate software interrupts⁸². Additionally, the GIC technology simplifies the virtualization of exceptions for hypervisor implementations in a multicore environment⁸³.

The first major function block of the Generic Interrupt Controller technology is the GIC CPU Interface, through which the core receives an interrupt. Every core in a multicore environment has its own CPU Interface that hosts registers to identify, mask and control the states of interrupts forwarded to that core.

The second main function block of the Generic Interrupt Controller technology is the Distributor. This external component has to be implemented by the SoC manufacturer. It controls all the properties of a specific interrupt by according registers, especially the routing information and the enable status for the attached CPU Interfaces.

The details of the configuration, the initialisation and the exception handling as well as the available features are determined by the version of the implemented GIC architecture on the one hand by the respective processor according to the internal GIC CPU Interface and on the other hand by the SoC manufacturer with regard to the external GIC Distributor. For example, Locality Specific Peripheral Interrupts (LPI), i.e. message-based interrupts, are not supported with GICv1 and GICv2, whereas this mechanism can be used in all higher versions⁸⁴. The ARMv8 Cortex-A53 processor supports all GIC

⁸¹[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.14.2, page D1-1836.

⁸²[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, cf. chapter 10, section 10.6, page 10-17.

⁸³[6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.4 f., page 9 ff.

⁸⁴[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, section 10.6, page 10-17.

architectures up to version 4⁸⁵. A good example for the initialisation and configuration of a Generic Interrupt Controller GICv3 running on the ARMv8 Foundation Model can be found in the ARM Limited startup code example delivered with the DS-5 Community Edition.

Accordingly, the two requirements 8 and 14 of the Muen SK can be evaluated as follows:

REQ-8 - IMPLEMENTATION DEFINED: *The ARMv8 architecture principally supports the programmatical handling of interruptions. However, since the possibilities and the extent of this handling depend on the implementation of a GIC distributor by the SoC manufacturer, this requirement is qualified as implementation defined.*

REQ-14 - IMPLEMENTATION DEFINED: *The ARMv8 architecture only provides hardware assisted routing of interruptions to individual cores through the implementation of a GIC by the SoC manufacturer. Therefore, this requirement has to be judged as implementation defined.*

3.6 Timers

The ARMv8 architecture prescribes the implementation of a system timer for processors of the Cortex-A series (cf. section 2.4). This system timer provides up to four timer channels per core - a secure and a non-secure physical timer as well as two timers for virtualization purposes. Each of these timer channels has at least one comparator, to configure the timers to generate an interrupt when the count is greater or equal to the programmed comparator value⁸⁶. The concrete implementation of the timer is determined by the respective processor type. An example would be the Generic Timer of the ARMv8 Cortex-A53 processor series described in the ARM Cortex-A53 Technical Reference Manual⁸⁷. The following steps are usually necessary to configure the timer:

- (i) *Comparator Value:* In a first step, the comparator value for the timer has to be written to the CNTP_CVAL_ELn according to the exception level, the timer should be used for.
- (ii) *Enabling Counter:* Then, the counter and the interrupt generation have to be enabled in the register CNTP_CTL_ELn.
- (iii) *Reporting:* In the last step, the code can poll the CTP_CTL_ELn register to report the status of the according exception level timer interrupt.

⁸⁵[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2016, chapter 9, section 9.1, page 9-2.

⁸⁶[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 14, section 14.1.3, page 14-5 f.

⁸⁷[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. 2016, chapter 10, page 10-1 ff.

The virtual timers and counters provided by the ARMv8-A architecture are explicitly designed for the scheduling of guest systems. Even though written in the programming language C, one can find a valuable example of the usage of the timer mechanism supported by the ARMv8-A architecture in the source code of the xvisor hypervisor⁸⁸. Therefore, the following can be stated:

REQ-15 - FULFILLED: *The ARMv8-A architecture explicitly supports at least a timer and a counter per core that can be configured to generate an interrupt. Even though the context switch has to be implemented manually (cf. section 3.3), this mechanism can be qualified as preemptive in the sense that it triggers an appropriate exception handling.*

3.7 Device Handling

Even though out of scope for this study, it has to be mentioned that the ARMv8-A supports device emulation as well as device assignment through the already described features of the ARM Virtualization Extension, i.e. the second level address translation and the (virtual) exception handling⁸⁹.

However, in order to get full device handling support, a SoC manufacturer also has to implement and provide an SMMU (corresponding to Intel's IOMMU) that meets the ARMv8 SMMU architecture specifications for the SMMU interface⁹⁰. Therefore, the corresponding requirement can be qualified as follows:

REQ-16 - IMPLEMENTATION DEFINED: *The ARMv8 architecture only provides a fully featured device handling through the implementation of a SMMU by the SoC.*

3.8 SPARK

As already mentioned in section 2.7, the Muen SK is written in SPARK. Since SPARK is a true subset of the Ada programming language and compilers ignore the SPARK inherent annotations, every correct SPARK program is also a correct Ada program and can therefore be compiled with an existing Ada compiler such as GNAT (part of the GNU compiler collection GCC).

To be able to qualify the requirement that there has to exist an Ada Cross Compiler for the ARMv8-A AArch64 execution state, a separate evaluation case has been written (cf. Development Environment Setup Ada Toolchain, appendix A). This document shows that it is possible to compile a custom Ada Cross Compiler for the ARMv8-A architecture based on the GNAT Ada Compiler toolchain of the GNU Compiler Collection GCC.

⁸⁸cf. A general overview over the xvisor hypervisor can be found [here](#). The source code is published under the GPL-2.0 license on github (<https://github.com/xvisor/xvisor>) and the mentioned generic timer code for ARMv8 AArch64 can be found in the file `generic_timer.c` in the directory `arm64 common, basic, timer`. December 21, 2017

⁸⁹[6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.2 f., page 8 f.

⁹⁰[10] n.a. *ARM System Memory Management Unit, Architecture Specification*. 2017.

REQ-17 - FULFILLED: An Ada Cross Compiler for the ARMv8-A AArch64 architecture can be compiled based on the GNAT Ada Compiler toolchain.

```

david@oslo:~/muen/rt5$ make clean
david@oslo:~/muen/rt5$ make
cp -a src/* /home/davidloosli/muen/rt5/obj/adainclude
gprbuild -RTS=.obj --config=aarch64-elf.cpr --target=aarch64-elf -p -j1 -PRTs
warning: -RTS is taken into account only in auto-configuration
rt5.gpr:21:25: warning: libraries are not supported on this platform
aarch64-linux-gnu-gcc -g -mcpu=5
/home/davidloosli/muen/rt5/src/asn/memcmp.5: Assembler messages:
/home/davidloosli/muen/rt5/src/asn/memcmp.5:9: Error: unknown mnemonic 'movq' -- 'movq %rdx,%rcx'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:10: Error: unknown mnemonic 'shrq' -- 'shrq %3,%rcx'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:11: Error: unknown mnemonic 'repe' -- 'repe'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:12: Error: unknown mnemonic 'cnpqsb' -- 'cnpqsb'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:13: Error: unknown mnemonic 'jne' -- 'jne %3'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:15: Error: unknown mnemonic 'movq' -- 'movq %rdx,%rcx'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:16: Error: unknown mnemonic 'andq' -- 'andq %7,%rcx'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:17: Error: unknown mnemonic 'repe' -- 'repe'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:18: Error: unknown mnemonic 'cnpqsb' -- 'cnpqsb'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:19: Error: unknown mnemonic 'jne' -- 'jne %3'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:21: Error: unknown mnemonic 'xorl' -- 'xorl %eax,%eax'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:24: Error: unknown mnemonic 'movl' -- 'movl %8,%ecx'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:25: Error: unknown mnemonic 'subq' -- 'subq %rcx,%rdi'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:26: Error: unknown mnemonic 'subq' -- 'subq %rcx,%rsi'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:27: Error: unknown mnemonic 'repe' -- 'repe'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:28: Error: unknown mnemonic 'cnpqsb' -- 'cnpqsb'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:29: Error: unknown mnemonic 'xorl' -- 'xorl %eax,%eax'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:30: Error: unknown mnemonic 'movb' -- 'movb -i(%rdi),%al'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:32: Error: unknown mnemonic 'movb' -- 'movb -i(%rsi),%dl'
/home/davidloosli/muen/rt5/src/asn/memcmp.5:33: Error: unknown mnemonic 'subl' -- 'subl %edx,%eax'
gprbuild: *** compilation phase failed
makefile:4: die Regel für Ziel „/home/davidloosli/muen/rt5/obj/adaltib/libgnat.a“ scheiterte
make: *** [/home/davidloosli/muen/rt5/obj/adaltib/libgnat.a] Fehler 4

```

Although ARM provides official guidelines for porting code from ARM A32 to ARM A64 assembly⁹² as well as from IA-32 to ARM A32⁹³ and many freely available tutorials can be found online, the runtime could not be translated during this study due to time constraints. Therefore, it is not possible to make a final judgement regarding the corresponding requirement:

REQ-18 - TESTING REQUIRED: Even though it should be possible to build a Muen Zero Footprint Runtime for the SPARK 2014 programming language and the ARMv8 AArch64 execution state with freely available software, the fulfilment of this requirement has to be tested in a further study.

⁹³ <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0274b/index.html>, December 21, 2017

3.9 Requirement Comparison

It has been shown that more than two thirds of the requirements of the Muen SK are directly supported by the ARMv8-A architecture. None of the prerequisites had to be qualified as *unsupported*. The fulfillment of the remaining requirements only depends on the target hardware and therefore on the implementation of the ARMv8 architecture by the respective SoC manufacturer. The following requirements had to be judged as *IMPLEMENTATION DEFINED* and thus have to be qualified based on the target hardware platform, i.e. the Raspberry Pi 3:

number	requirement	topic
REQ-0	The processor architecture has to support 64 bit data-path widths, integer size and memory address widths as well as to be able to execute 32 bit applications.	basics
REQ-8	A target processor architecture has to provide a mechanism to programmatically handle interruptions.	interruption handling
REQ-14	A target processor architecture should optionally provide a technique to fast process interruptions between cores.	interruption handling
REQ-16	A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for according interruption and memory features.	device handling

Table 3.1: *IMPLEMENTATION DEFINED* requirement summary

4 Raspberry Pi 3

The Raspberry Pi 3 is the third generation of the Raspberry Pi series and the target platform for this study. The first part of this chapter provides a general overview of the Raspberry Pi 3. In the following sections, the hardware platform is discussed with respect to the requirements qualified as *IMPLEMENTATION DEFINED* in the previous chapter 3.

4.1 Overview

The Raspberry Pi 3 Model B is the latest single board computer developed and released in February 2016 by the Raspberry Pi Foundation. The main component of this small computer is the BCM2837 System on Chip (SoC), which implements an ARMv8 Cortex-A53 processor with four cores. Also worth mentioning in the context of this study are the 1GB RAM, the Micro SD port and the 40-pin GPIO provided by the platform. Further details on the specifications can be found on the homepage of the Raspberry Pi Foundation ¹.

The Raspberry Pi 3 Model B was chosen as target platform for this study because it is the first Raspberry Pi generation that is capable of running software written for the 64-bit execution state. In addition, the Raspberry Pi single board computers are explicitly intended for experimentation and are therefore almost not „brickable“ as well as inexpensive.

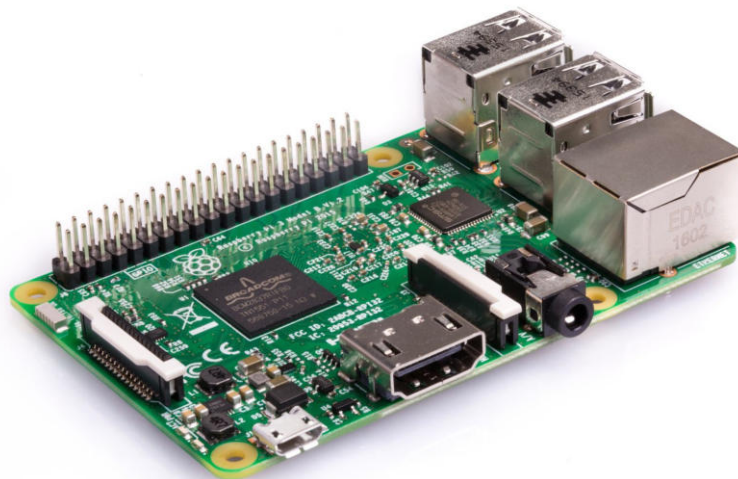


Figure 4.1: Raspberry Pi 3 Model B, © by the Raspberry Pi Foundation

¹cf. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b>, December 21, 2017

The architecture of the Raspberry Pi 3 does not quite come up to one's expectations. In contrast to most other ARM based SoC, not the ARMv8 Cortex-A53 processor but the Broadcom VideoCore is the organising part and has full control over the initialisation of each component. In addition, the VideoCore also contains and controls essential system architecture components such as the memory controller or the level 2 cache. The latter is used almost exclusively by the VideoCore and is usually bypassed when accessing the CPU². The ARM processor is only attached to the organising VideoCore and can be addressed via a corresponding CPU interface. Figure 4.2 shows a schematic overview for the architecture of the Raspberry Pi 3³.

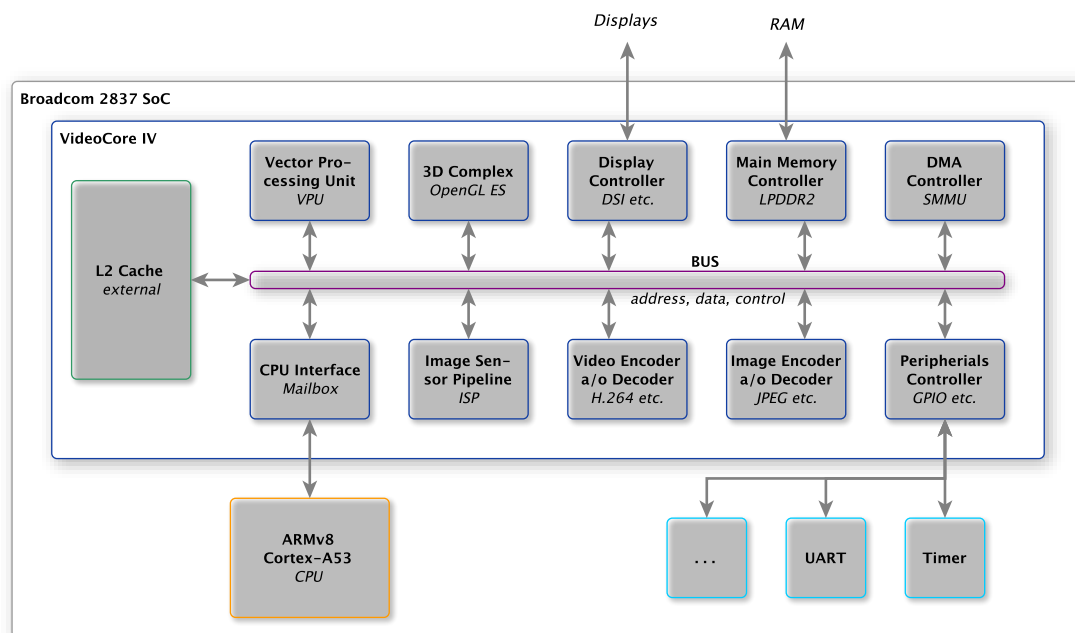


Figure 4.2: Raspberry Pi 3 schematic

4.1.1 Documentation

First of all, it has to be stated that there exists neither a complete official documentation on the Raspberry Pi 3 nor any official documentation on the changes with respect to the AArch64 mode of the Raspberry Pi 3. On the website of the Raspberry Pi Foundation, it is only mentioned that nothing has changed compared to the Raspberry Pi 2 SoC except for the ARMv8-A processor⁴. The documentation for the Raspberry Pi 2 consists of two datasheets for the Raspberry Pi 1⁵ and a supplementary

²[1] n.a. *BCM2835 ARM Peripherals*. 2012, chapter 1, section 1.2.3, page 6.

³<https://www.heise.de/ct/ausgabe/2016-8-Wie-es-mit-dem-Raspberry-Pi-weitergeht-3150082.html>, December 21, 2017

⁴<https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md>, December 21, 2017

⁵<https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md>, December 21, 2017

document for the changes compared to the Raspberry Pi 1⁶. Even though there obviously exist differences between the 64-bit and the 32-bit mode of the Raspberry Pi⁷, most of the following statements should apply to both execution states⁸. Therefore this chapter is primarily based on the following literature:

- *VideoCore*: The official VideoCore IV 3D Architecture Reference Guide⁹ for the Raspberry Pi 1 serves as the main source for boot related questions.
- *Broadcom SoC*: As the primary sources for ARM Peripheral related topics, the two official BCM2836 ARM Peripherals¹⁰ and BCM2835 ARM Peripherals¹¹ documents are used.
- *Raspberry Pi Bare Metal Forum*: A lot of explanations and findings in the context of the AArch64 development can be found on the official Raspberry Pi Bare Metal Forum¹².
- *Raspberry Pi Repositories*: The Raspberry Pi Foundation maintains several Github repositories. In particular, the documentation repository was used for this chapter¹³.
- *Bare Metal Repositories*: The most important Raspberry Pi Bare Metal repositories for this study are the two Github repositories maintained by David Welch¹⁴ and by Peter Lemon¹⁵.

Because a detailed and with respect to the AArch64 architecture complete Raspberry Pi 3 hardware reference manual as well as a comprehensive guide for Bare Metal Programming on the Raspberry Pi 3 did not exist at the time of writing, a separate Raspberry Pi 3 Beginner's Guide has been started as a collection of all the existing, but widespread sources on this topic. This guide is going to be continued and developed by the author even after this Student Research Project and is going to be published under an open source license.

⁶<https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md>, December 21, 2017

⁷David Welch and Peter Lemon could show with their code that not only the base address for the kernel image but also some alternative modes for the peripherals change. A **personal assumption** in this regard is that these changes are caused by the firmware of the VideoCore initialising the ARM processor in the AArch64 execution state.

⁸Of course, this would have to be proven in a continuing study (cf. section 5.3)

⁹[13] n.a. *VideoCore IV 3D Architecture Reference Guide*. 2013.

¹⁰[5] Loo. *BCM2836 ARM Peripherals (documentary supplement)*. 2014.

¹¹[11] n.a. *BCM2835 ARM Peripherals*. 2012.

¹²<https://www.raspberrypi.org/forums/viewforum.php?f=72>, December 21, 2017

¹³<https://github.com/raspberrypi>, December 21, 2017

¹⁴<https://github.com/dwelch67/raspberrypi>, December 21, 2017

¹⁵<https://github.com/PeterLemon/RaspberryPi>, December 21, 2017

4.1.2 Bare Metal Development

The development of bare metal programs differs greatly from software development on higher abstraction levels. The variety of development tools (compiler, IDE etc.) is relatively wide, but not all available tools are suitable for a specific task. The setups for the Raspberry Pi 3 used in this study are therefore briefly explained in this section.

The first inconvenience in bare metal development is loading newly built or rebuilt kernel images from the IDE to the Raspberry Pi. Actually, there are four possibilities for this task:

- (i) *SD Card*: For the AArch64 development, this option consists in formatting an SD Card to FAT32, copying the corresponding kernel image `kernel8.img` together with the boot files `bootcode.bin`, `start.elf` as well as `config.txt` to the card, inserting the card into the card slot of the Raspberry Pi and restarting it. Further information and two code examples are recorded in the two evaluation cases *Hello Muen! on HDMI* written in assembly and *Hello Muen! on UART* written in C (cf. appendix A).
- (ii) *Bootloader*: David Welch provides a bootloader that is capable of loading a kernel image to the Raspberry Pi 3 over a serial connection. Both the bootloader and instructions for its usage can be found on David Welch's Github repository ¹⁶.
- (iii) *JTAG*: The Joint Test Action Group JTAG interface can not only be used to load a kernel image to the Raspberry Pi 3 but also allows to run a debugger like the freely available Open On-Chip Debugger (OpenOCD). Therefore, this option has been chosen for this study. A complete guide for setting up the hardware as well as the OpenOCD debugger in combination with the Eclipse IDE is contained in the evaluation cases *Development Environment Setup* (cf. appendix A).
- (iv) *Netboot*: Since the JTAG option seemed to be the most suitable one for this study, the Netboot was not tested during this project. However, a guide for this option can be found on the official Raspberry Pi Foundation homepage ¹⁷.

Of course, the compiler toolchain as well as the IDE depend on the programming language used in a specific project. Nevertheless, the GNU MCU Eclipse IDE from Liviu Ionescu ¹⁸ has to be mentioned here, because it has been used as a development environment in almost all experiments of this study and can be adapted to different languages. The GNAT Programming Studio (GPS) ¹⁹ of the Community Edition ²⁰ provided by AdaCore was used for the development of code examples written in Ada.

¹⁶ <https://github.com/dwelch67/raspberrypi>, December 21, 2017

¹⁷ <https://www.raspberrypi.org/blog/pi-3-booting-part-ii-ethernet-all-the-awesome>, December 21, 2017

¹⁸ <https://github.com/gnu-mcu-eclipse/org.eclipse.epp.packages/releases>, December 21, 2017

¹⁹ <https://www.adacore.com/gnatpro/toolsuite/gps>, December 21, 2017

²⁰ <https://www.adacore.com/community>, December 21, 2017

4.2 Boot Process

Due to the special architecture of the Raspberry Pi 3 (cf. section 4.1), the boot process also does not correspond to the one of most other ARM development boards. As soon as the Raspberry Pi is turned on, the VideoCore assumes control over the boot process while the ARMv8 Cortex-A53 processor is still off and uninitialised²¹. The VideoCore then takes the following actions²²:

- (i) *First Stage Bootloader*: The VideoCore starts the boot process by executing the first stage bootloader stored in ROM on the Raspberry Pi SoC. This bootloader initialises and reads the SD card and loads the second stage bootloader from the SD card into the level 2 cache.
- (ii) *Second Stage Bootloader (bootcode.bin)*: This bootloader enables and initialises the SDRAM. While for earlier versions of the Raspberry Pi it loads the third stage bootloader loader.bin from the SD card into RAM, the second stage bootloader for the Raspberry Pi 3 supports loading ELF files and therefore directly loads the GPU firmware from the SD card into RAM.
- (iii) *GPU firmware (start.elf)*: The start.elf first initialises the GPU, second loads, reads and executes the CPU configuration file config.txt and finally loads the kernel image into RAM.

The above described boot process already suggests that there are basically two possibilities for configuring the ARMv8 processor. The first option is to modify the configuration file accordingly²³. For all non hardware dependent configurations, the processor can also be initialised manually.

As already explained in section 3.2.3, the initialisation of the ARMv8 processor into the AArch64 execution state depends on a hardware signal to a pin of the processor. Since the VideoCore starts the ARMv8 processor by default in the AArch32 execution state and since a warm reset depends on a hardware defined reset register with an unknown address in AArch64 execution state, the only way to initialise the ARMv8 processor in 64-bit mode is to add the following lines to the config.txt file:

```
arm_control=0x200
kernel_old=1
```

Even though it seems that there exists only this option to start the ARMv8 Cortex-A53 processor on the Raspberry Pi 3 in the AArch64 execution state, the corresponding requirement can be qualified as fulfilled:

REQ-0 - FULFILLED: *The Raspberry Pi 3 supports the initialisation of the ARMv8 processor in a 64-bit execution state and hence fulfils this requirement.*

²¹ cf. boot process explained by David Welch on <https://github.com/dwelch67/raspberrypi>, December 21, 2017

²² <https://www.raspberrypi.org/documentation/.../bootflow.md>, December 21, 2017

²³ Details to the configuration possibilities can be found on <https://www.raspberrypi.org/documentation/configuration/config-txt> as well as on the Raspberry Pi Foundation Github repositories, December 21, 2017

4.3 Exception Handling

First of all, it has to be stated that the interrupt controller provided by the Raspberry Pi 3 SoC is neither programmable nor does it implement the Generic Interrupt Controller (GIC) interface specified by the ARMv8 architecture.

The exception handling on the Raspberry Pi 3 is also special. The documentation distinguishes between two different types of interrupts, i.e. core related and core un-related interrupts. The category of the core related interrupts includes the four timer interrupts, a performance monitor interrupt and the four Mailbox interrupts for each core. The only thing that can be determined programmatically with respect to core related interrupts is whether to send an interrupt to either the IRQ pin or the FIQ pin as well as to disable the interrupt handling at all²⁴. An example of a Mailbox interrupt handling can be found in the evaluation case *Hello Muen! on HDMI*. All other interrupts and exceptions (GPU interrupts, local timer interrupts, AXI error and Peripheral interrupts) are assigned to the core un-related interrupts category²⁵. These interrupts have to be enabled, configured and handled completely in code by setting the according bits of an interrupt register of the corresponding interrupt type as well as by setting up the processor correctly²⁶. An example for an UART interrupt handling can be found in the evaluation case *Hello Muen! on UART*.

The question now arises as to whether the described exception handling meets the requirements of the Muen SK. In order to be able to assess this question, one has to take a closer look at two practical examples. Both the Xen hypervisor²⁷ and the Kernel Virtual Machine KVM²⁸ explicitly state that they rely on an implementation of the GIC interface specified by ARM Limited. While the Xen hypervisor therefore does not support the Raspberry Pi 3, KVM circumvents this problem by implementing the GIC specification in a virtual GICv2 interface. The second option would also allow the Muen SK to run on the Raspberry Pi 3. However, since the Muen SK requires a smallest possible code base and the Raspberry Pi 3 does not implement the GIC interface, the corresponding requirements derived in chapter 2 have to be qualified as not fulfilled.

REQ-8 - NOT FULFILLED: *The Raspberry Pi 3 does neither support a fully programmable interrupt controller nor the GIC interface specified by the ARMv8-A architecture. Therefore, this requirement has to be judged as not fulfilled.*

REQ-14 - NOT FULFILLED: *Even though the Raspberry Pi 3 provides a mechanism to enable fast interrupt requests FIQ, it does support an inter-core communication due to the missing implementation of the GIC interface. Hence, this requirement has to be qualified as not met by the target platform.*

²⁴ [5] Loo. *BCM2836 ARM Peripherals (documentary supplement)*. 2014, chapter 3, section 3.2.1, page 5.

²⁵ [5] Loo. *BCM2836 ARM Peripherals (documentary supplement)*. 2014, chapter 3, section 3.2.2, page 5 f.

²⁶ cf. <https://www.raspberrypi.org/forums/viewtopic.php?f=72&t=38076>, December 21, 2017

²⁷ https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper, December 21, 2017

²⁸ <https://lwn.net/Articles/557132/>, December 21, 2017

4.4 Device Handling

Since this topic is out of scope for this study, the device handling on the Raspberry Pi 3 is not discussed in detail. However, it can be stated that even though the Raspberry Pi 3 has a separate MMU for device handling it does not implement the SMMU interface specified by ARM Limited^{29 30}. Therefore, the corresponding requirement derived in the second chapter is not met:

REQ-16 - NOT FULFILLED: *The Raspberry Pi 3 does not support the SMMU interface specified by the ARMv8-A architecture.*

4.5 SPARK

In the context of this study, it was also tried to build the official AdaCore Zero Footprint Runtime for the ARMv8-A AArch64 execution state with hardware specific adaptations for the Raspberry Pi 3 from the AdaCore Github Repository³¹. However, this attempt also failed. A detailed description can be found in the evaluation case *Problem Description Toolchain*.

²⁹[11] n.a. *BCM2835 ARM Peripherals*. 2012, chapter 1, section 1.2, page 4 ff., and chapter 10, section 10.6.3, page 158 f.

³⁰https://www.reddit.com/r/raspberry_pi/comments/4aonbh/why_are_there_two_mmus_on_the_bcm2835 and <https://www.raspberrypi.org/forums/viewtopic.php?f=72&t=138108&p=920301>, December 21, 2017

³¹<https://github.com/AdaCore/bb-runtimes/tree/gpl-2017/aarch64/rpi3>, December 21, 2017

5 Conclusion

The aim of this feasibility study was to evaluate the ARMv8 Virtualization Extension for the porting of the Muen SK to the ARMv8 architecture as well as to carry out a risk assessment on its portability to the target platform Raspberry Pi 3 with regard to a possible bachelor thesis. This chapter is dedicated to this two aspects of the study.

5.1 ARMv8 Architecture

Principally, the ARMv8 architecture and the ARMv8 Virtualization Extension can be considered suitable for porting the Muen SK. Nevertheless, there are some risks involved that have to be addressed.

The first point and at the same time the one with the highest risk for the bachelor thesis is the context handling. In contrast to Intel's VT-x technology, the ARM Virtualization Extension does not provide any automatic handling of a context switch (cf. section 3.3). In addition, the registers, that have to be stored, depend to a certain degree on the respective guest system and the current execution state of the subject. Therefore, the context switch has to be implemented completely by the hypervisor developer.

As on the Intel x86/64 architecture, the caching structures of the ARMv8 architecture too have to be considered as potential sources of side channels. Since the ARMv8 architecture does not specify the implementation of the level 2 and an optional level 3 cache, it is also important to investigate the actual implementation of the caching structures by the manufacturer of a target SoC.

The two specifications of the Generic Interrupt Controller and the System Memory Management Unit by ARM Limited also pose a certain risk. Due to the large number of different versions and sometimes only partial implementations of the interfaces by the manufacturers of a SoC, these two components have to be examined particularly thoroughly when choosing a target platform.

5.2 Raspberry Pi 3

First of all, the missing documentation for the AArch64 mode of the Raspberry Pi 3 has to be considered as problematic, since a precisely described and defined operation mode is essential, especially for high-security applications.

The first problem could be mitigated by an open source firmware. Although Broadcom has published the documentation for the Raspberry Pi 1, a large part of the firmware is still only available in a binary format. Since the VideoCore also has complete control over the initialisation of the hardware, many details can only be estimated (e.g. memory allocation VideoCore vs. CPU). This also has to be qualified as a major risk for porting the Muen SK to the Raspberry Pi 3.

The last two risks are related to the implementation of the GIC and the SMMU interfaces specified by the ARMv8 architecture. Even though the two interfaces can be implemented in software, this involves on the one hand a high risk with regard to the bachelor thesis and on the other hand it is fundamentally contradictory to the requirement of a smallest possible code basis stated by the Muen SK.

Therefore the risk for choosing the Raspberry Pi 3 as the target platform is too high, especially without any further investigations. As a conclusion, it cannot be qualified as suitable for porting the Muen SK to the ARMv8-A architecture with respect to a possible bachelor thesis.

5.3 Further Investigations

In this final section of the study, an approach for further investigations is presented. In a first step, one of the following reference kernels could be used as a starting point for the porting of the Muen SK to the ARMv8-A architecture:

- *HASPOC hypervisor*: The HASPOC hypervisor is a high assurance security kernel for the ARMv8 architecture that is available as open source software under the terms and conditions of the Apache License 2.0. The documentation and the source code can be found on the HASPOC homepage ¹.
- *seL4 microkernel*: According to the official seL4 homepage ², the seL4 microkernel is the most advanced member of the L4 microkernel family. The source code is published under the GPLv2 and the BSD2 license on Github ³.
- *Xvisor hypervisor*: The Xvisor hypervisor is an open source type I hypervisor that supports full virtualization also for the ARMv8-A architecture ⁴. The source code can be found on the Xvisor Github repository ⁵.
- *Phidias*: As already mentioned, the Phidias hypervisor developed by Jan Nordholz follows the same principle as the Muen SK but seems to support the ARMv8-A AArch64 architecture ⁶. The source code is published on the Phidias Repository ⁷.

¹ <https://bitbucket.org/account/user/sicssec/projects/HASPOC>, December 21, 2017

² <https://sel4.systems>, December 21, 2017

³ <https://github.com/sel4/sel4>, December 21, 2017

⁴ <http://xhypervisor.org/>, December 21, 2017

⁵ <https://github.com/xvisor/xvisor/tree/v0.2.10>, December 21, 2017

⁶ [15] Nordholz. *Design and Provability of a Statically Configurable Hypervisor*. 2017.

⁷ <http://phidias-hypervisor.de/repos/core.git>, December 21, 2017

In a second step, further classifications of the actual features used by the Muen SK in the context of the Programmable Interrupt Controller and the System Memory Management Unit would have to be carried out. In addition, the implementation of the interfaces in software has to be balanced against the requirement of a smallest possible code base stated by the Muen SK.

Depending on the findings of the first two steps, an alternative ARMv8 platform may have to be considered. It is recommended to investigate the Hardkernel Odroid C2 based on an AMLOGIC S905 SoC as the first alternative platform. This target platform seems to be documented in detail and to have hardware support for the GICv2 as well as the SMMU interface.

As the last part of the investigation before porting the Muen SK to the ARMv8-A architecture, additional clarifications of the registers, that have to be saved during a context switch, could be helpful.

Appendix

A List of Related Documents

- Glossary
- Hello Muen! on HDMI - *bare metal assembly code for Raspberry Pi 3*
- Hello Muen! on UART - *bare metal C code on Raspberry Pi*
- Development Environment Setup - *Assembly and C/C++ toolchain, JTAG debugger and IDE for ARMv8 AArch64*
- Development Environment Setup - *Ada toolchain, JTAG debugger and IDE for ARMv8 AArch64*
- Problem Description Toolchain - *Ada toolchain ARMv8 AArch64*
- Raspberry Pi 3 AArch64 - *An Unofficial Bare Metal Beginner's Guide (to be continued)*

B Project Assignment AVT (german)

Untersuchung der Portierung des Muen Separation Kernel auf ARM

Studiengang: Informatik (I)
 Semester: HS 2017/2018 (18.09.2017-18.02.2018)
 Durchführung: Studienarbeit

Fachrichtung: Sicherheit
 Institut: ITA: Internet-Techn. und Anwend.
 Gruppengrösse: 1 Studierende(r)
 Status: zugewiesen

Verantwortlicher: Steffen, Andreas
 Betreuer: Rügsegger, Adrian-Ken
 Gegenleser: [Nicht definiert]
 Experte: [Nicht definiert]
 Industriepartner: [Nicht definiert]


Ausschreibung: Der Muen Separation Kernel (SK) ist ein spezialisierter Microkernel der als Plattform für Hochsicherheitssysteme am INS entwickelt wird. Muen gewährleistet eine strikte und zuverlässige Isolierung von Komponenten und schützt sicherheitskritische Funktionen vor fehlerhafter Software, die auf dem gleichen physischen System läuft. Um eine besonders hohe Vertrauenswürdigkeit zu erreichen, wird die Programmiersprache SPARK 2014 eingesetzt.

Der SK wurde speziell für die Intel x86_64 Architektur entwickelt und verwendet Intel VT-x und VT-d für die Separierung der Komponenten.

Diese Arbeit hat zum Ziel, die ARMv8/AArch64 Virtualisierungserweiterungen zu untersuchen und zu evaluieren, wie die Technologie zur Portierung des Muen SK auf ARM eingesetzt werden kann.

Als Zielhardware ist das Raspberry Pi 3 vorgesehen.

Voraussetzungen: Gute Linux-Kenntnisse
 Interesse an systemnaher Entwicklung

Bewerbungen: Gruppe: Loosli 
 Einschreibung: Studienarbeit
 Status: Arbeit zugewiesen (Priorität Student: 1)
 Studierende: Loosli, David
 Kommentar: Zur Sicherheit bewerbe ich mich hiermit noch offiziell - ich bin mir nach einem Gespräch mit einem Mitstudenten nicht mehr ganz sicher, ob die Arbeit bereits mir zugeteilt ist.

Fenster schliessen

Bibliography

- [1] Benjamin A. Braun, Suman Jana, and Dan Boneh. "Robust and Efficient Elimination of Cache and Timing Side Channels". In: *CoRR* abs/1506.00189 (2015), p. 15. URL: <http://arxiv.org/abs/1506.00189>.
- [2] Reto Buerki and Adrian-Ken Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2013. URL: <https://muen.codelabs.ch>.
- [3] Tessaleno Devezas, João Leitão, and Askar Sarygulov. *Industry 4.0 - Entrepreneurship and Structural Change in the New Digital Landscape*. Covilhã (Portugal) and Saint Petersburg (Russia): Springer International Publishing AG, 2017. ISBN: 978-3-319-49603-0.
- [4] Eduard Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2nd ed. Urdorf (Switzerland): dpunkt.verlag GmbH, Heidelberg, 2010.
- [5] Gert van Loo. *BCM2836 ARM Peripherals (documentary supplement)*. revision 3.4. Cambridge (England), 2014.
- [6] n.a. *AArch64 Virtualization*. version 1.0. Cambridge (England): ARM Limited, 2017.
- [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. version B.a. Cambridge (England): ARM Limited, 2017. URL: <http://www.arm.com>.
- [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. version 1.0. Cambridge (England): ARM Limited, 2015. URL: <http://www.arm.com>.
- [9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. revision r0p4. Cambridge (England): ARM Limited, 2016. URL: <http://www.arm.com>.
- [10] n.a. *ARM System Memory Management Unit, Architecture Specification*. version 3.0 and version 3.1. Cambridge (England): ARM Limited, 2017. URL: <http://www.arm.com>.
- [11] n.a. *BCM2835 ARM Peripherals*. version 1.0. Cambridge (England): Broadcom Europe Ltd., 2012.
- [12] n.a. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*. Santa Clara (USA): Intel Corporation, 2017. URL: <https://software.intel.com/en-us/articles/intel-sdm>.
- [13] n.a. *VideoCore IV 3D Architecture Reference Guide*. version 1.0. Irvine CA (USA): Broadcom Ltd., 2013.
- [14] Gil Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". In: *Intel Technology Journal* 10.3 (2006), pp. 167–177.
- [15] Jan Nordholz. *Design and Provability of a Statically Configurable Hypervisor*. Berlin (Germany): Technische Universität Berlin, 2017.

- [16] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *ACM Operating Systems Review* 17.7 (1974), pp. 412–421.
- [17] John Rushby. “Design and Verification of Secure Systems”. In: *ACM Operating Systems Review* 15.5 (1981), pp. 12–21.
- [18] Andrew S. Tanenbaum and Herbert Bos. *Moderne Betriebssysteme*. 4th ed. München (Germany): Pearson Studium, Hallbergmoos, 2016.

List of Figures

2.1	Intel x86 protection mode, protection rings hierarchy	10
2.2	example of a memory hierarchy	12
2.3	example of a one level paging with partitioning	14
2.4	example of a one level address translation	15
2.5	simplified interruption process	19
2.6	timer component	24
3.1	JTAG adapter with Raspberry Pi 3	31
3.2	DS-5 Community Edition restrictions	32
3.3	DS-5 Community Edition project view	33
3.4	DS-5 Community Edition debug view	33
3.5	ARMv8-A Exception Levels in AArch64	34
3.6	ARMv8-A Exception Level Switch debugger view	36
3.7	ARMv8-A Execution States rules	37
3.8	ARMv8-A standard memory organisation	41
3.9	DS-5 Debugger MMU memory map	43
3.10	DS-5 Debugger MMU translation tables	44
3.11	ARMv8-A Second Level Address Translation	46
3.12	gprbuild Muen SK ZFP output	54
4.1	Raspberry Pi 3 Model B, © by the Raspberry Pi Foundation	56
4.2	Raspberry Pi 3 schematic	57

List of Tables

2.1	requirement summary part one	26
2.2	requirement summary part two	27
2.3	requirement summary part three	28
3.1	<i>IMPLEMENTATION DEFINED</i> requirement summary	55