Modern hardware abstractions for firmware

Doctoral Thesis

Author(s):
Humbel, Lukas

Publication date:
2022

Permanent link:
https://doi.org/10.3929/ethz-b-000539181

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information, please consult the Terms of use.
Modern hardware abstractions for firmware

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by

Lukas Humbel

Master of Science ETH in Computer Science, ETH Zurich

born on 12. October 1987

citizen of Boniswil AG, Switzerland

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. Margo Seltzer (University of British Columbia),
co-examiner
Dr. David Cock (ETH Zurich), co-examiner

2022
Abstract

Operating systems inevitably make assumptions about the hardware execution environment. These assumptions influence the design of application programming interfaces (APIs), internal data structures, as well as the hardware model in verified operating systems. Due to the growing hardware diversity and complexity, these assumptions get violated often. This leads to hard-to-port operating systems, APIs that leak underlying implementation details and can invalidate the guarantees of verified operating systems.

This dissertation revisits the assumptions about two critical, widely used but often overlooked subsystems: The interrupt delivery subsystem and I²C, a low-speed configuration bus, that is used to interact with actors and sensors, including critical ones such as voltage regulators.

For both of these subsystems, we propose two new well-specified abstractions built on a formal model. These subsystems have grown in diversity: A general-purpose operating system has to support interrupt controllers from multiple vendors, understand interrupt delivery of many different subsystems and perform configuration of these controllers. Each subsystem offers different ways of delivering interrupts. Hardware virtualization support of device pass-through adds another layer of complexity, as interrupts can be targeted directly to virtual machines.

We present a formalization of the interrupt network that includes configurability, a crucial component as controllers vary widely in
their configuration options. We refine this model until it is suitable for constraint solving and show its generality by expressing 24 real interrupt controllers in this model. We then use this solver-based approach to configure the interrupt system of a real operating system, Barrelfish. Existing approaches in commodity operating systems, like Linux and FreeBSD, rely on heuristics to configure interrupts. By nature, these approaches will overlook configuration options. Our approach does not rely on heuristics and finds optimal results. Furthermore, it represents the current configuration in a formally defined way. We show that the overhead of solving the configuration problem with a constraint solver is negligible in comparison to driver startup and that the approach scales well in a typical scenario.

\(^{2}\)C attached devices are widely used for critical tasks such as controlling voltages, sensing temperatures, and storing persistent device configurations. The standard is written in natural language and leaves ambiguities to device developers. Thus, the bus is plagued with incompatibilities between host interfaces and devices. The few existing formalizations of I\(^{2}\)C do not incorporate partially compliant devices. In practice, these hardware controllers are often replaced by an inefficient software-only approach.

To tackle this I\(^{2}\)C device diversity, we present a modular formalization of I\(^{2}\)C and a framework for verifying device driver interactions. We use model-checking to show equivalence to an abstract layer specification. Model-checking was chosen to minimize the developer effort, but the approach is formulated in a tool-agnostic model. Partially compliant devices can be expressed in this framework by interfacing with a different layer. These layers are fine-grained, which enables partially compliant devices to be expressed with little developer effort.

Device and driver models are expressed in a custom language that can automatically generate C code, Promela, and Verilog. The generated code results in compact, low-overhead implementations. The code for interacting with a partially compliant device is comparable in complexity to existing approaches, but guarantees correct interaction with a device model, offers automatic code generation, and easier to understand error reporting.
Zusammenfassung

Betriebssysteme machen zwangsläufig Annahmen über die Hardware auf welcher sie ausgeführt werden. Diese Annahmen beeinflussen den Entwurf von Programmierschnittstellen (APIs), internen Datenstrukturen sowie des Hardwaremodells von verifizierten Betriebssystemen. Aufgrund der wachsenden Vielfalt und Komplexität der Hardware werden diese Annahmen immer häufiger verletzt. Dies führt zu Betriebssystemen die schwierig zu portieren sind, APIs, bei denen zugrundeliegenden Implementierungsdetails durchscheinen und dass die Garantien von verifizierter Betriebssysteme verletzt werden kön-
nen.

In dieser Dissertation werden die Annahmen über zwei kritische, weit verbreitete aber oft übersehenen Subsystemen überdacht: Das Interrupt Verteilungs Subsystem und \( I^2C \), ein Konfigurationsbus, der zur Interaktion mit Aktoren und Sensoren verwendet wird. Unter anderem wird \( I^2C \) dazu verwendet sicherheitskritischen Schaltungsteilen wie Spannungsreglern zu konfigurieren.

Für diese beiden Subsysteme schlagen wir zwei neue, strikt spezi-
fizierte Abstraktionen vor die auf einem formalen Modell aufbauen. Beide Subsysteme haben an Vielfalt zugenommen: Ein Allzweck-
Betriebssystem muss Interrupt-Controller von mehreren Herstellern unterstützen, die Interrupt-Verteilung vieler verschiedener Subsyste-
me verstehen und die Konfiguration dieser Interrupt-Controller vor-


Um diese I²C-Gerätevielfalt zu bewältigen, stellen wir eine modulare Formalisierung von I²C und ein Framework für die Verifizierung von Treiber-Geräte Interaktionen. Wir verwenden Model-checking, um die Äquivalenz zu einer abstrakten Spezifikation zu zeigen. Model-checking wurde gewählt, um den Entwickleraufwand zu minimieren, das Modell selbst ist aber unabhängig von Model-checking formu-

Acknowledgments

First and foremost, I would like to thank my friends and colleagues that listened and helped me on countless office interactions, collaborated on the research presented in this dissertation, and simply created a pleasant work environment.

I would like to thank Timothy “Mothy” Roscoe for offering me this opportunity, helping with research, from ideas to paper writing, and acting as a living reference on computer science history. Furthermore, special thanks go to David Cock, not only for taking part in my committee, but also for providing feedback and valuable formal methods background during the past years. I would also like to thank Margo Seltzer for taking part in my committee and providing valuable in-depth feedback.

To Dejan Milojicic. Thank you for the opportunity to intern at Hewlett Packard Labs. I would also like to thank my fellow intern Leonid Azriel for making the internship so enjoyable.

To Melissa. Thank you for not only being an amazing colleague and collaborator, but also a great partner. To our daughter, Maya, thank you for understanding that “Frozen” has to wait sometimes.

To my fellow friends and colleagues in the Systems Group, Barrelfish, and Enzian teams. Reto, Roni, Daniel, Nora, Michael, Simon, Gerd, Anastasiia, Andrea, Monica, Vojislav, Debopam, Moritz, Stefan, Pravin, Abishek, Renato, Claude. Thanks for your support and enlightening discussions at coffee breaks.
Thanks to Simonetta, Jena, and Nadia for smoothly organizing everything. Finally, thanks for all the unconditional support, love, and dinners I received from my parents Lisbeth and Rolf.
## Contents

1. **Introduction**  
   1.1. Related Publications ............................................. 19  

2. **Background**  
   2.1. Hardware Descriptions ............................................. 22  
   2.1.1. Topology ............................................................. 22  
   2.1.2. Interface Description Languages .............................. 24  
   2.1.3. Behavioral Description for Hardware Synthesis ........... 25  
   2.1.4. Behavioral Description for Software ........................ 26  
   2.2. Current Trends in Operating Systems ......................... 27  
   2.3. ACPI ......................................................................... 28  
   2.4. Devicetrees .............................................................. 30  
   2.5. Conclusion ............................................................... 37  

3. **Interrupt Controller Configuration**  
   3.1. Introduction .................................................................. 40  
   3.1.1. Background ............................................................. 45  
   3.2. Static Interrupt Configuration ....................................... 52  
   3.2.1. Model and Syntax .................................................... 53  
   3.2.2. Modeling Real Systems .......................................... 56  
   3.2.3. Reductions and Algorithms ...................................... 61  
   3.3. Interrupt System Configuration ..................................... 64  
   3.3.1. Configurable Model ................................................... 65
## 3.3. Prolog Representation
3.3.2. Prolog Representation ........................................ 69
3.3.3. Interrupt Controller DSL .................................... 73
3.3.4. The Controller Zoo .......................................... 77

## 3.4. Implementation in an OS
3.4.1. The Routing Service ......................................... 102
3.4.2. Topology Discovery ........................................... 104
3.4.3. Clients ...................................................... 106
3.4.4. Evaluation ................................................... 107

## 3.5. Conclusion
3.5.1. Future Work ................................................ 112

## 4. I^2C Specification
4.1. Introduction ................................................... 116
4.2. Background .................................................... 118
4.2.1. I^2C Basics ................................................ 118
4.2.2. Non-Compliant Devices .................................... 120
4.3. Related Work .................................................. 125
4.4. Approach and Tools ........................................... 127
4.4.1. Programming Model, DSL, and Backends .................. 133
4.4.2. Calculus ................................................... 137
4.5. The I^2C Model ................................................ 140
4.5.1. Layering of I^2C ............................................ 140
4.5.2. Layer 0: Electrical Layer .................................. 140
4.5.3. Layer 1: Symbol Layer .................................... 141
4.5.4. Layer 2: Byte Layer ....................................... 142
4.5.5. Layer 3: Transaction Layer ................................ 144
4.5.6. Layer 4: Driver ............................................. 146
4.5.7. Layer 5: World ............................................. 147
4.6. Evaluation ..................................................... 148
4.6.1. Expressing Non-Compliant Devices ....................... 148
4.6.2. Verification Runtime ....................................... 152
4.6.3. Execution on a Raspberry Pi ............................... 153
4.6.4. Synthesizing Hardware .................................... 154
4.7. Conclusion and Future Work ................................... 157
1. Introduction

A core task of an operating system (OS) is to act as an illusionist. The OS exposes to applications an illusion of the hardware. For example, the file system abstracts the underlying storage system. Between this exposed illusion and the actual physical hardware, an OS often maintains additional internal data structures which further abstract an aspect of the hardware. To extend on the example, between the file system and a hard disk drive (HDD), an OS may include a block storage layer, which stores continuous blocks of memory in a linear fashion. These illusions and internal data structures are driven by the needs of applications as well as assumptions about the structure of the hardware. For example the block storage layer is built this way, because the underlying hardware (HDDs) expose a block oriented interface.

Hardware is developing at a rapid pace and existing OSs face the problem of integrating hardware that does not conform to the assumptions made by the OS design in the first place.
Chapter 1 - Introduction

The cost of modifying an existing abstractions is high because of the large number of clients. In a general purpose OS like Linux, clients of, for example the file system (e.g. applications) cannot be known \textit{a priori}. Even changing internal OS abstractions can be difficult, as a large number of drivers interact with this subsystem. In Linux, each driver uses an interrupt subsystem Application Programming Interface (API) to receive asynchronous notifications from the managed device. Modifying this interface would require changing all device drivers.

Since the cost of changing an abstraction is high, in practice new hardware that violates the OSs assumptions are shoehorned into existing abstractions. For example, to interface with byte-addressable Non-volatile Memory (NVM) a block abstraction is used, ignoring the advantages of byte addressability. While this approach requires little developer effort it has a number of drawbacks: Hardware improvements might not be offered to clients (e.g. the byte addressability of NVM). Such new hardware features are often exposed by using flags in APIs. The use of flags to indicate special behaviour creates an non-generic abstraction that expose underlying implementation details, as those flags are only implemented by that specific backend. Another common assumption is about the configurability of devices. For example, interrupt controllers are assumed to be freely configurable. If a device violates this assumption a configuration heuristic is used. This can lead to suboptimal system configuration, such as interrupt sharing, and it is hard to predict how the heuristic will behave on new systems. Both scenarios make it hard to port an operating system and incorporate new hardware.

In formally verified OS, these abstractions influence the assumed hardware model, which makes these challenges even more acute. Verified operating systems are mathematically proven to follow a specification. Some verified operating systems prove higher level properties of this specification. For example, SeL4 guarantees \textit{timeliness} (i.e. an upper bound on interrupt delivery latency). But the formal model makes assumptions about the execution environment. The SeL4 guarantee of timely delivery of interrupts assumes that interrupts are delivered correctly to the CPU in the first place. In practice
this means that a correct configuration for the interrupt controllers must have been found.

This dissertation focuses on two important but often overlooked subsystems: The interrupt subsystem with its configurability and I²C, a pervasive low-speed configuration bus. We first analyze how existing operating systems and abstractions work, and how related hardware descriptions handle these subsystems in Section 2.

Correct interrupt delivery is important, for example, in a Real-time Operating System (RTOS). If the scheduler does not receive timer interrupts, it can not ensure the promised guarantees. Interrupt delivery is also becoming more complex; Kernel bypass network stacks and device pass-through for virtual machines require frequent and correct reconfiguration of the interrupt delivery mechanism. Backward compatibility and the variety of devices, buses, and delivery methods makes this a surprisingly complex problem.

In particular, we articulate the following hypotheses:

• Current interrupt models do not express interrupt controller configurability.

• Our approach called the decoding net, is a suitable formal basis for expressing interrupt networks and their configurability.

• A constraint solver using this model can find interrupt controller configurations.

• A solver-based OS implementation is feasible and has a better policy mechanism separation.

In Section 3, we analyze the interrupt routing and configuration options on a variety of systems. We propose a formal model to reason about interrupt delivery in the interrupt network. We distill and define a common "interrupt controller" abstraction applicable to all interrupt controllers we encountered. Unlike existing abstractions, our approach exposes configurability of the interrupt controllers. We argue that this is necessary to separate interrupt routing policy from driver implementation. We show that this abstraction is useful in
practice for declarative interrupt network configuration and use it at runtime to configure hardware on a real operating system, Barrelfish. We further express the static part in a theorem prover to reason about interrupt controllers.

OSs also make assumptions about the behavior of peripheral devices, such as those connected via I²C. A common assumption is that the devices adhere to the standard, which influences the design of I²C APIs. Correctly interacting with I²C connected devices is important as they often fulfill critical system management tasks, such as setting voltages. At the same time, the I²C bus is plagued with non-compliant devices and buggy host controller hardware. Thus drivers and other clients of this API must work around these bugs and, due to the API design, have to resort to disabling host controller hardware completely and using a software-based implementation (bit-banging). This CPU-intensive solution is less than ideal.

In particular, we articulate the following hypotheses in Section 4

• Interactions with non-standard I²C devices must be possible. Verifying an implementation solely against the (informal) standard is not useful in practice.

• Existing I²C verification approaches have only verified a controller or a device against (an interpretation of) the standard.

• Our layered approach assigns clear semantics to standards conforming as well as partially compliant operations.

• Furthermore, our approach is also suitable for formal reasoning. In particular, we model check driver device interactions.

I²C systems are often one-off designs; The hardware seldom changes and is used only in a specific use case. The same can not be said for software; The Linux I²C stack is used in many different hardware scenarios. We compare the approaches in terms of code complexity and clarity when interacting with non-compliant devices and show that our approach is equivalently complex.
We develop a thinly-layered specification of an I²C stack, including both controller and device side. The specification consists of an executable part that can interact with real I²C devices. We show that the specification is consistent within itself, the device correctly parses correctly the commands generated from the controller. We also automatically extract the executable part into a compact C representation that can be used even on low-end microcontrollers. Furthermore, we also generate Verilog from the executable specification.

1.1. Related Publications

This dissertation is based on the following published publications and theses.

The implementation of the interrupt configuration system is built on Barrelfish\textsuperscript{1} to which many people have contributed. Chapter 3 builds on the static model presented in [Ach+17a] and the solver based technique presented in [Hum+17].

The specification of I²C was motivated by the work on Enzian\textsuperscript{2}. Chapter 4 is an extension of the work presented in [Hum+21], which in turn is built upon the thesis [Sch20].


\textsuperscript{1}http://www.barrelfish.org
\textsuperscript{2}http://www.enzian.systems


The following theses have analyzed existing work and influenced the design of both chapters:


This chapter discusses existing hardware models, descriptions and languages with particular focus on our use cases:

- The topology and configurability of interrupt controllers.
- The topology of I²C buses and I²C device behavior.

Thus, we are interested in approaches that describe topology: Which interrupt controllers are present and how are they connected? Which I²C buses contain which devices?

For I²C devices and controllers, we also focus on the behavior of those devices, thus behavioral hardware descriptions are also relevant.

In Section 2.1 we give an overview of existing hardware descriptions. In Section 2.2 we discuss current related operating system research developments. In Section 2.3 and Section 2.4 we provide more detail of the most related, commonly found system descriptions: ACPI and devicetrees. We conclude the chapter with a quick comparison of the models we encountered in Section 2.5.
Chapter 2 - Background

2.1. Hardware Descriptions

We start in Section 2.1.1 with relevant descriptions of the hardware topology. This includes runtime discoverable platform descriptions as well as DSL based approaches. These are declarative approaches that focus on expressing hardware subsystems in a machine readable format.

Interface description languages, presented in Section 2.1.2, are conceptually also declarative, as they describe the register layout of devices. But since some device registers are accessible only using convoluted mechanisms, some of the projects also describe abstract device behavior.

Hardware descriptions appear in a wide range of areas and for a variety of purposes. Hardware design languages are used for simulation, verification and synthesis of electrical circuits. These languages focus on the behavior of a circuit; we discuss those in Section 2.1.3.

Some approaches specify the behavior of hardware, but not with the goal of describing circuits, but an abstract device behavior; for example, to synthesize device drivers. We discuss those in Section 2.1.4.

2.1.1. Topology

Runtime discoverable: Platforms offer information to operating systems in various ways: In the x86 architecture there have always been efforts to provide information and functionality at runtime. The Basic Input/Output System (BIOS), Advanced power management (APM) and more recently The Advanced Configuration and Power Interface (ACPI) [UEFI17] promised to deliver such hardware information in a runtime discoverable, operating system agnostic way [Gro03]. We discuss ACPI in Section 2.3.

Since ARM is strategically targeting the server market, it has defined the ARM Server Base System Architecture (SBSA) [arm21] which allows a universal operating system to be built as on x86. Notably, it mandates the presence of ACPI. But the diverse range of ARM SoCs manufacturers show no sign of adopting this or any
similar standard. For end-user device manufacturers, little economic incentive for providing a standard conforming platform exists. These standards would prevent vendors from differentiating themselves from competition with software features.

Some busses such as PCI Express [PCI04] and the Universal Serial Bus (USB [USB17]) are discoverable and allow the operating system to query available devices. The I\(^2\)C is not discoverable and thus relies on external information about the topology.

In summary, discoverable hardware descriptions are an essential building block in extracting hardware topology. Any effort to represent a whole subsystem must be able to access these descriptions. However, the discoverable subsystems provide only a partial view of the whole system (in the case of interrupts) or none at all (in the case of I\(^2\)C). For the interrupt topology, we incorporate runtime information in our OS implementation. For I\(^2\)C we found existing descriptions lack any behavior of attached devices.

Embedded ARM SoCs do not offer any runtime discovery mechanism. The loaded software (trusted firmware, UEFI environment, bootloader, operating system) are expected to have hardcoded hardware maps to find hardware resources and their interconnections. The proliferation of large numbers of such SoCs made existing operating systems (Linux, Zephyr) as well as bootloaders (Das U-Boot) adopt a human read- and writable platform description: devicetrees. We discuss devicetrees in Section 2.4.

We also encounter topology information in some academic projects: Singularity [Spe+06] uses language safety to isolate processes from each other. Drivers are implemented like a normal process, except that drivers need access to specific hardware resources. Unlike in today’s systems, the resources needed by a driver are declared statically in a driver manifest. While we appreciate expressing resource needs in a stringent machine readable fashion, in this case, it is focused on starting device drivers. It falls short, for example, in the description of the interrupt subsystem: Interrupts are represented in a flat, enumerated namespace and the description does not capture the possibility of dynamic interrupt routes, multiple CPU cores where a driver could be
executed or other situations. To be fair, this paper was published in the year 2006. At this time, more elaborate interrupt delivery mechanisms such as MSI-x were not yet widely available.

Schüpbach *et al.* argues that using declarative techniques [Sch12] makes an operating system more general and flexible than one based on C configuration heuristics. The paper illustrates this using PCI bridge programming and creating interrupt assignments for the PCI interrupt routing. The latter is included as a sub-problem in our work, we also extend it to capture interrupts outside of the legacy PCI domain, finding routes and generating configurations for all controllers on the interrupt route. We follow (in Section 3) very much the same approach: Using declarative techniques and solvers to obtain a flexible implementation.

### 2.1.2. Interface Description Languages

Devil [Mér+00] is a interface description language for devices. Devil focuses on the format and access of device registers. Since devices are complicated, this description contains more behavior than one might expect at first: Devil understands how to access both banked and index-based registers. It also understands which registers cause an action and thus must not be cached. However it does not describe any semantics of any register. NDL [CE04] can be seen as an extension of Devil, it does contain a similar register description but also contains a imperative language for manipulating these device registers. Mackerel [Bar13], a DSL used in the Barrelfish operating system, is another example of an interface description language.

Hail [Sun+05] is similar to NDL but introduces a layer of indirection to handle devices that are best accessed using non-memory mapped access (such as I/O ports or if the OS provides a special API for accessing PCI configuration space). Furthermore, it also allows the specification of constraints, which can be either invariants or sequential constraints. These properties then are translated to assertions at runtime.
2.1. Hardware Descriptions

These interface description languages do neither describe any topology nor device behavior and are thus not directly applicable to our problem. They could however be useful in a future effort when writing or verifying an interrupt controller driver. In fact, the interrupt controller drivers we implemented in Section 3.4 use Mackerel.

2.1.3. Behavioral Description for Hardware Synthesis

Hardware description languages (HDL) traditionally denote languages that work at the register-transfer level (RTL). HDL tools transform RTL into a lower representation, such as a bitstream for an FPGA or wiring layouts. RTL is also a suitable abstraction level for simulation. The most common input languages that specify hardware at the RTL level are VHSIC hardware description language (VHDL [1E19]) or Verilog [1E06].

High-level Synthesis (HLS) is the process of using a general-purpose language to describe hardware [Nan+16]. Under this umbrella, almost all programming languages can be considered hardware description languages: from OpenCL kernels [Jo+20] to Python [LZB14] as well as functional languages such as Haskell. For example Bjesse et al. [Bje+98] leverages the powerful type system of Haskell.

Such hardware description languages are unsuitable for use by operating systems for a number of reasons. There are policy reasons for hardware vendors not to provide sources for their hardware, as it would allow competitors to learn details about the implementation or even allow them to create counterfeit chips straight away.

Second, RTL descriptions are designed for synthesis and not for outside analysis. For example to determine which addresses an address decoder specified in RTL decodes, a halting-problem like problem has to be solved. Clearly RTL description are only of limited utility for operating systems.
2.1.4. Behavioral Description for Software

There are languages, that conceptionally operate at the level of RTL, but are not aimed at synthesizing hardware. They abstract from the precise hardware behavior by combining multiple states of the backend into a single state. For example they abstract away from the precise RTL level implementation of an ALU but merely express that it performs an addition. Or, in the case of network interface devices, they merely state that a packet is sent, instead of describing how exactly the packet is sent.

There are many cases in which such a hardware description is needed, such as synthesis of device drivers, operating systems, or for the verification of low-level assembly code. While these efforts describe behavior and often are executable, they abstract away from circuit level implementation details. Conversely, ARM Architecture Specification Language (ASL) \cite{Rei16} Instruction Set Architecture (ISA) specification does not contain any information about the CPU architecture, and synthesis is either impossible or would lead to a very inefficient hardware design.

Recent academic efforts such as Sail \cite{Arm+19} and x86-TSO \cite{Sew+10} also fall in this category. They have contributed languages and toolchains for formally analyzing instruction set architectures. A similar ISA description language is also being used in an operating system synthesis effort \cite{Hu+19, Hol+20}.

Our work does not focus on the behavior of CPUs, but on the behavior of devices. For description languages, devices can be seen as a generalization of CPUs, as they process not only an instruction stream but react to more complex events. In the particular case of I²C we needed particular language features to allow the per-layer verification and thus we did not use any of the existing languages.

Termite \cite{Ryz+09} uses a device specification to synthesize device drivers. The abstraction is roughly on the RTL level, although typical device descriptions contain only subset of the registers, just enough to capture the behaviour that is relevant to the OS, but not from the hardware perspective.
2.2. Current Trends in Operating Systems

The termite approach is applicable to a single interrupt controller, if a suitable, common abstraction of an interrupt controller has been found. For network cards - as presented in the paper - the fundamental operation is sending packets. It is unclear what the fundamental operations of an interrupt controller are. We define such a common abstraction for the interrupt forwarding behavior in Section 3. For describing an entire interconnected network of interrupt controllers or bus devices, the language is not suitable, as it is not designed for describing topology.

2.2. Current Trends in Operating Systems

The verification of operating systems (such as SeL4 [Kle+09], CertikOS [Gu+16], or the Hyperkernel [Nel+17]) and drivers ([Bal+10]) has recently sparked much interest in the systems community. Improvements from the formal methods community now allow the targeting of larger verification projects.

At the same time, network and storage speeds are increasing more quickly than CPUs. This makes it infeasible to use a kernel based network stack for high performance networking applications. Operating systems researchers have recognized this trend: IX [Bel+14], Arrakis [Pet+14] and DemiKernel [Zha+21] are dataplane operating systems. Device access is possible without a kernel invocation, interrupts are either relayed by the control or, if suitable hardware support exists, directly delivered to the applications.

Combining these two trends in a cloud setting, where other tenants must not be trusted at all, creates new requirements for interrupt delivery: Interrupt delivery (ideally directly to userspace using hardware extension) is critical for the functioning of such kernel bypass applications and thus must be verified. We argue this is only possible with a suitable model of interrupt delivery.
2.3. ACPI

The Advanced Configuration and Power Interface (ACPI [UEFI]) is the standard for discovering hardware in x86 based systems and it is making inroads into the ARM based server ecosystem, as ACPI is required to fulfill the ARM Server Base System Architecture [arm21].

ACPI is a rather large specification, consisting of more than 1000 pages. The specification abstracts away the hardware from the operating system in two ways: ACPI defines tables that allow the system to inspect the current hardware arrangement, and it defines a bytecode with entry points for the operating system. The bytecode can invoke methods on objects, which are laid out in a hierarchical namespace. These nodes reference other nodes, for example the AC adaptor node type has a method that returns node identifiers of consumers. This creates links inside the hierarchical node structure. So far, this datastructure looks similar to devicetrees. Unlike devicetrees that have unstructured nodes and a schema language, there is only one place where ACPI is defined: The specification itself. Thus a large part of the specification (chapters 6 until 15) defines the objects and their methods.

**IRQ** ACPI does describe parts of the interrupt subsystem implicitly. Structure information (to which controller are interrupts forwarded from a controller) is not given explicitly, but has to be known by the operating system implementer. As an example, the OS has to know that if an IOMMU is activated, the output of an IOAPIC (IOAPIC) is forwarded to that IOMMU. No forward pointer from the IOAPIC exists to the IOMMU.

The operating system can find, in the multiple APIC descriptor table, the memory mapped location of IOAPICs (on x86) and GICs (on ARM) interrupt controllers. PCI devices expose a method that can be used to determine the GSIVs of that device.

ACPI aspires to enumerate all "wired interrupts" by assigning them a Global system interrupt vector (GSIV). This is a rather crude

---

1 Citation from the specification, Section 5.2.12.15
method to describe the network of interrupt controllers, and it only works because the GSIV must be resolved to a interrupt controller in the ACPI context before it can be actually useful. For example IOAPIC objects have a system vector base property, which specifies the lowest GSIV number that is handled as input. From that base number, the IOAPIC accepts a consecutive number of GSIVs. How many is determined by the IOAPIC type. As of now there exists only 24 and 16 input IOAPICs.

Unfortunately, the implicit topology makes it impossible to reason about remote interrupt receivers such as co-processors. It also excludes the notion of message signaled interrupts, core local interrupts and interprocessor interrupts.

ACPI not only describes the system, but also exposes methods, written in a bytecode, that can be interpreted by the operating system. Some interrupt controllers can be controlled using this interface. For instance, the PCI link device can be configured using the set resource settings method, which takes GSIVs as arguments. In essence, there is a whole driver for PCI link devices encoded. Unfortunately this is not the case for other controllers, such as the IOAPIC. A programmer is supposed to query the (memory mapped) location and type of an IOAPIC using ACPI, and then continue programming it like any other memory mapped device.

Nevertheless using ACPI for configuring parts of the interrupt system is attractive because of the vendor supplied bytecode. We show, in Section 3.4, how we integrate our interrupt controller implementation with the ACPI mechanisms.

I2C ACPI does offer a System Management Bus (SMBus) interface designed for interacting with a subset of power-related I2C devices, in particular batteries, chargers, power selectors and a system managers. These devices’ nodes are required to be located in the SMBus host controllers node, so the structure is strictly specified.

However this I2C interface is plagued with the usual problem of implementing high level I2C transactions - the inability to operate with non-compliant devices. Thus operating systems often fall back
using a lower level interface (as we show in Section 4).

A general problem with ACPI is that the discoverable ACPI information contains bugs, and operating systems include mechanisms to override the default behavior. For example FreeBSD includes the device.hints [Fre] that allows passing override information to the kernel. In addition, similar to FreeBSD, Linux also contains a 5000 line C file describing non-standard PCI files [Mar]. The Barrelfish research OS proposed encoding these quirks in the system knowledge base [Sch+11].

2.4. Devicetrees

Devicetrees are a hardware description syntax and toolchain that has been adopted by Linux and its community to support initialization in non-discoverable environments. This has spurred adoption outside of the Linux kernel: The Das U-Boot bootloader [Eng21], FreeBSD [Jaw] or the embedded operating system Zephyr [Pro] all use devicetrees as their configuration format. The widespread adoption and acceptance makes it an interesting study subject for two reasons: First, analyzing the format and tools itself can help us to learn and leverage our own efforts of describing hardware subsystems. Second, the large number of devicetrees (in the Linux kernel repository there are more than 2300 DTS files as of today, see Figure 2.1) make it attractive to use it as a database in our own effort. Devicetrees do have a specification [OfI15] specifying the syntax as well as the binary encoding. It says very little about which nodes and properties are allowed (explained below). This is a stark contrast to the ACPI specification, that defines in detail which logical objects should be present. Not surprisingly, this also makes the devicetree specification (59 pages) significantly shorter than the one of ACPI (roughly 1000 pages).

The devicetree format  A Device Tree describes a tree of nodes. Each node has a set of properties (key-value pairs). Keys must be identifiers; values must be one of the elementary datatypes. The
2.4. Devicetrees

Figure 2.1.: Devicetrees in the Linux kernel repository
elementary datatypes are strings, byte arrays and cell arrays. The first two are self-explanatory and the third is (by default) an array of unsigned 32-bit integers, but with prefixes, the cell size can be changed to 8, 16, 32 or 64 bits. Syntactically, cell arrays are enclosed in angle brackets (< and >). A node can contain an arbitrary number of child nodes. Each node has a name (which must be unique in its context) and optionally is followed by an @ and a decimal address. The standard does not mandate any meaning of this address, but in practice it is often used to encode a unique identifier, for memory mapped devices this is often the base address. It is permitted to use any key, but some keys have a further specified interpretation. The \#address-cells and \#size-cells define the layout of the cell array associated with the reg property. Which is used to store the memory base address. The first \#address-cells elements of the cell array describe the starting address, and the remaining \#size-cells elements describe the size. Outside of nodes, the devicetree file can also contain commands such as the /dts-v1/. These are sequentially interpreted and can be used to remove, change and add nodes. This mechanism allows a form of reuse; A devicetree file can include another and slightly modify it. These commands are interpreted at compile time and are invisible to the software client. An example of the devicetree syntax is given in Listing 2.1 which shows an excerpt of the Toradex Colibri iMX8QXP board.

Nodes can have an assigned label, which in turn can be used to refer to that node. An example illustrating the node references is shown in Listing 2.2. At the compile step, the labels are turned into a integer that uniquely identifies the node (called the phandle). A node can also be referred to by path, which creates a symbolic reference string. This string can then be used by the runtime to perform another lookup.

```plaintext
/dts-v1/;
/
  model = "Toradex Colibri iMX8QXP/DX on Colibri Evaluation Board V3";
```
Devicetrees

```dts
#address-cells = <0x02>;
#size-cells = <0x02>;

cpus {
    #address-cells = <0x02>;
    #size-cells = <0x00>;
    cpu@0 {
        // identify the device as a physical cpu core
        device_type = "cpu";
        // value used by the OS to find a suitable driver
        compatible = "arm,cortex-a35";
        // CPU identifier
        reg = <0x00 0x00>;
    };
    cpu@1 {
        device_type = "cpu";
        compatible = "arm,cortex-a35";
        reg = <0x00 0x01>;
    };
}

memory@80000000 {
    // identify the device as a physical memory
    device_type = "memory";
    // the memory addresses start at 0x80000000
    // the memory size is 0x40000000 addresses
    reg = <0x00 0x80000000 0x00 0x40000000>;
}
```

Listing 2.1: Simplified Colibri devicetree

```dts
// dt_ref.dts
/dts-v1/;

/ {
    // references a node by label
    property1 = <&label1>;
    property2 = <&label2>;
    // references a node by path
```
Devicetree abstractions  The syntax of devicetrees is sufficiently defined that we were able to successfully write a parser and reimplement the node merging algorithm [Rüe]. However, we were less successful in extracting operating system independent information. As a starting use-case we were interested in extracting the memory mapped register locations of devices. At first this seems straight forward; the reg property defines the starting address and size (in accordance with the #size-cells and #address-cells). However, critically it does not state whether the address is encoded in little or big endian. Furthermore, as we argue in [Ach+20], the memory hierarchy has to be represented as a directed graph, and different actors in the memory system (observers)
have different views of the memory system. Devicetrees assume a single observer and therefore, addresses in the reg properties are not qualified by an observer. These observers could be, for example, a CPU or a device performing direct memory accesses. The lack of context makes it impossible to minimally (without extraneous mappings) configure an IOMMU.

In Linux, the free-form properties of a node are further restricted using bindings. These files are traditionally a natural language description of which parameters in a devicetree node have to be present and how they are interpreted. More recently, Linux has begun switching to machine readable bindings in the YAML format. The bindings are written in JSON schema [Her], which is a schema for JSON data and the schema itself is a JSON [Hut] - despite that in Linux neither the schema (in YAML) nor the data to be checked (a devicetree syntax) are actually JSON. JSON schema can check the presence of properties, constrain the values of properties to a value range and require a certain structure. JSON schema can not check for referential integrity. A schema checker for devicetree exists, but it is not integrated in the mainline Linux repository.

The interrupt subsystem is defined outside of the node tree structure. A node can trigger an interrupt by including the interrupts property, which is specified as an array of integers. Each group of #interrupt-cells represents one interrupt. The format of an individual interrupt is not further defined. The interrupt-parent references another node, to which the interrupt is sent. A node can have the interrupt-nexus property that turns a node into a static interrupt translating node. interrupts-extended does the same, but can state an individual interrupt parent for each interrupt. Configurable interrupt controllers are declared using the empty interrupt-controller property. If interrupts but no interrupt parent is specified, a node is assumed to trigger an interrupt at the CPU. This property is used on last-level interrupt controllers but, somewhat inconsistently, also on device nodes.

This specifies the interrupt topology in a way that is independent of the existing node structure and describes a directed graph. While
this is better than the memory topology, which exists only implicitly, it still lacks the ability to express an interrupt targeting another core and does not include core local or interprocessor interrupts. The specification also does not express the configuration options of an interrupt controller, that we show in Section 3 is necessary to separate configuration policy from driver implementation. Despite the shortcomings, the devicetree authors have realized that there is not one, unique hierarchy but there are many, and they may look different for different subsystems (i.e. the memory subsystem is not structured in the same way as the interrupt subsystem).

Outside of the scope of devicetrees, but still crucial for an actual operating system, is the inclusion of runtime discoverable interrupt sources, such as PCI and USB. Despite describing only the static part, it would still be helpful to define entry points into the devicetree interrupt structure from PCI interrupts.

Devicetree as database  Since devicetrees are not stored in a centralized repository, the devicetree database does not exist. The Linux repository contains the largest collection of devicetree files. Unlike the interrupt system, the I²C subsystem is not even mentioned in the devicetree standard. Thus, there is no specified way of describing a bus controller and determining which devices are connected to it. In practice, the convention is that I²C has a node name that starts with i2c.

This node name is also matched by the corresponding binding, which states that there must be a #size-cells property set to zero and a #address-cells property set to one. Further, the clock-frequency property must be in range of 1000 to 3000000, and that the i2c-bus must be a not further specified object. The binding makes no statement how devices connected to this controller should be specified. The schema format could require all children to be of a certain, I²C device type. It can not require the absence of other devices referring to the controller by phandle, as there is no way of referencing other nodes. In other words, we can not state that no other node refers to this one, and thus enforce that all other devices on the bus must be children.
2.5. Conclusion

<table>
<thead>
<tr>
<th>Feature</th>
<th>ACPI</th>
<th>devicetrees</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit destinations</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>topology</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>configuration</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>message signaled</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>interrupts</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>CPU local interrupts</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>methods</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>runtime</td>
<td>yes</td>
<td>no</td>
<td>implementation defined</td>
</tr>
</tbody>
</table>

Table 2.1.: Interrupt models compared

But what about actual devicetree files? Do they, despite not being specified anywhere, follow a convention? Unfortunately there is a rather big variety: The Raspberry Pi 3B+ uses phandle to bind a I²C device to a controller, while the Toradex puts devices connected to this master in child nodes. This is possible because the code interacting with these subsystems is different, and thus there is also some code duplication.

Furthermore, as we discuss in Section 4, I²C devices vary wildly in their standard conformance and behavior, so apart from topology it is crucial to also include the bus behavior of these devices.

2.5. Conclusion

A summary of the interrupt model features explained above is shown in Table 2.1. Our support for runtime depends on how the operating system interacting with the model is implemented.
This chapter begins in Section 3.1 with an explanation of the interrupt subsystem; the evolution and the challenges arising from it for operating systems, hypervisors, and firmware. In Section 3.2 we describe a static formal model of the interrupt system and its representation in Isabelle/HOL.

In Section 3.3 we extend the model to configurable systems and controllers. We then show the generality by expressing an extensive list of interrupt controllers in this model. We further describe a realization of this model in Prolog.

In Section 3.4 we use this Prolog realization to configure interrupts on a real operating system and put the solver performance in context. Section 3.5 concludes the chapter and discusses future work.
3.1. Introduction

This work presents a formal model for the unambiguous interpretation of interrupts of a modern computer system that captures the relevant features of contemporary hardware. This model can be used to concisely define terms such as \textit{vector}, \textit{interrupt route}, or \textit{interrupt controller}. Its applications include a foundation for system software verification, identifying problematic hardware designs, and generating correct-by-construction OS code.

The same model was designed to also capture memory accesses but the description of the memory related aspects is outside the scope of this dissertation.

A naive view of interrupts (often repeated in OS textbooks) is as follows: interrupts are asserted by a device, translated by a Programmable Interrupt Controller (PIC) into a CPU local “vector number”. The processor uses this vector number as an index into a jump table to call the correct “handler” for this interrupt.

This view is both plain wrong and unsuitable as a basis for verification as or even well-written software. Modern systems, from mobile phone Systems-on-Chip to large servers, are complex networks of cores, memory, devices, and translation units. Multiple interrupt controllers in this network interpose on the forwarding of interrupts. Virtualization support creates additional layers of interrupt translation as well as (virtual) interrupt destinations.

We know of no prior formal model capturing interrupt routing in modern hardware. As discussed in Section 3.1.1 even existing informal attempts to capture the increasingly diverse range of hardware to facilitate OS software development fail to adequately cover the software/hardware interface. Our starting point for this work was the need for a practical Domain-specific Language (DSL) with clear semantics for use in the Barrelfish research OS.

\footnote{As presented in \cite{Ach+17a}. Reto Achermann contributed the analysis of the memory subsystem. I contributed the analysis of the interrupt subsystem. The rest of the work has equal contributions from all authors.}
Answering a question like “Is this CPU interrupt vector triggered only by this device?” requires knowledge of the contents of interrupt controller lookup tables, their configuration registers, and crucially, a representation of the system topology. Thus the formalism has to be complete in this sense. Existing approaches, for example the ACPI interrupt enumeration discussed in Section 2.3, fall short in this regard as they do not incorporate MSIs and do not express interrupt destinations. In particular, we identify the following properties our formalism must be able to express:

**Multiple names:** Consider the overview of the x86 interrupt network in Figure 3.1. An interrupt triggered by a Peripheral Component Interconnect (PCI) device is first represented as a PCI interrupt number (1). The link device turns this interrupt into an input to the IOAPIC (20). The IOAPIC translates this vector number to an IOMMU table index (230). The IOMMU in turn creates a bus message, that encodes the destination vector and CPUs in a platform specific way. This bus message then gets sent to all LAPICs, but is decoded only by one. The result is a vector number that is finally delivered to the CPU. On ARM, interrupts can also have different vector numbers, depending to which subsystem they are directed. Thus our model has to be able to express the different representations at each step of the interrupt
Chapter 3 - Interrupt Controller Configuration

Topology diversity: This topology is not uniform on modern systems. While x86 is standardized, it consists of many discoverable elements that might not be present or are present multiple times. Of the controllers in [Figure 3.1] the IOAPIC, the link device and the IOMMU can be present any number of times. A similar problem exists with ARM SoCs. While they usually do not contain a device discovery mechanism, they are all slightly different in the devices present and how they are interconnected. Thus an operating system or a verification effort has to handle a large number of different topologies, which our formalism has to be able to express.

Multicast: An interrupt might be directed to multiple destinations. For example in [Figure 3.1] after the interrupt passes the IOMMU, it becomes a bus message. The bus message encodes in a platform specific way to which subset of cores the interrupt is to be delivered. This allows interrupts to be targeted to multiple destinations. Thus, the model must support targeting an interrupt to multiple destinations.

Non-uniform destinations: [Figure 3.2] shows a subset of the interrupt topology of a typical ARM SoC. The A9 cores each have private

Figure 3.2.: Simplified ARM interrupt network
timer interrupts. Devices like the general purpose timer can target only the A9 cores, while other device interrupts can also be directed to the M3 and DSP subsystems. The A9 cores can initiate interrupts among themselves but not target other cores. Similar complexity exists in almost all modern systems. On x86 PCs, core local timer interrupts and IPIs with different delivery options create the situation where not every interrupt can be directed to the same set of cores. Because of this non-uniformity, it is important that a formalism explicitly includes interrupt destinations.

In the next subsection we discuss in more detail the interrupt controllers present on a modern system. In Section 3.2 we present a formalism that captures the forwarding behavior and challenges mentioned in this introduction. We also revisit the two examples informally presented here and formalize them using the proposed model.

However, correct software operation requires more than just a description of a system configuration snapshot. Rather the operating system is obliged to correctly configure interrupt routing across an increasingly larger range of hardware platforms. The problem of correctly configuring the interrupt subsystem is surprisingly complex. In addition to the already mentioned problems, the configuration options of interrupt controllers vary widely. Some interrupt controllers can not produce all the outputs. For example the link device (see Figure 3.1) can only produce four different inputs of the IOAPIC instead of all the IOAPICs 24 inputs. Some controllers can not be configured at all, but perform a fixed function. Other controllers have a configurable base address and forward multiple interrupts with regard to this address. IOAPIC and IOMMU perform a lookup in a potentially large in-memory table to determine their output. Other controllers can be configured to block interrupts, but not otherwise change them.

The interrupt configuration problem is also important: a modern general-purpose OS has to handle the full complexity of a system like this, and the topology is generally not known when the OS is written. Correct operation of the OS on a new hardware system depends on
the correct configuration of this network by software.

In Section 3.3 we extend the static formal model with configurability, such that it not only represents the complete interrupt topology of real computer systems from a PC to a phone System-on-Chip (SoC), but also all possible configuration options of the network. To exemplify the configuration challenge, consider in Figure 3.1 the IOMMU. The IOMMU outputs a bus message which is forwarded to all LAPICs. The operating system can configure the IOMMU to create a different bus message for an input interrupt, but it can also configure the Local APIC (LAPIC) to accept a different set of bus messages using a bitmask. Thus the operating system has to find a configuration that works for each interrupt route that has to be set up. Our model expresses these configuration options.

We then further refine the model until it is suitable for constraint solving at runtime. This allows the operating system to specify which routes, given as source destination pairs, are to be installed. The solver will then find a suitable configuration that satisfy all these requests. Thus the operating system no longer must make use of heuristics to configure the LAPIC bitmask, but can rely on the solver output.

We then use this solver based approach to configure interrupt controllers on demand in the Barrelfish OS. The implementation consists of two parts: A DSL that describes the configuration options of interrupt controllers and a runtime component that uses this description for finding valid configurations. The DSL compiles into Prolog facts describing the configuration options of a controller. These facts are then used at runtime to configure interrupt controllers. This approach cleanly separates interrupt routing policy from controller driver implementation.

Formally modeling interrupts also has value beyond system software design since it can shed light on desirable properties of hardware designs, both complete platforms and individual controllers.
3.1. Introduction

3.1.1. Background

Modern interrupt hardware is complex. Whereas in the distant past, an interrupt was a dedicated electrical signal to the processor, today a computer has a network of interrupt controllers that can be configured to deliver many distinct interrupts generated by a given device to different vectors on different cores. New interrupt controllers are introduced all the time, whether evolutions of existing designs or new, specialized functions for particular SoCs. Each has different capabilities and constraints on the number of interrupt signals they can source and sink, and how they can map between them.

By default, a CPU executes instructions linearly. In practice, the CPU and the executing system software must react to events from devices. One way to do so is to use polling. This technique requires the CPU to frequently query all devices to determine if they have an event pending. Usually, we want to execute a program on the CPU, hence the operating system can not poll devices all the time. It is possible that the operating system only polls at specific time intervals, but this introduces a latency between event occurrence and event handling.

Interrupts solve the problems of immediate reaction and preemption. Originally, they were implemented using a dedicated wire going to the CPU. On the Intel platform this was the case up to and including the Intel 486 processor. In this model, once an electrical voltage changes or appears, the CPU saves its current execution state (including instruction pointer and registers) and continues execution at a fixed address (the interrupt handler). Starting with the Pentium, the interrupt signals do not have dedicated, physical CPU pins anymore. Instead, interrupts are mapped to messages which are sent to the CPU using a multi purpose bus, such as HyperTransport. An interrupt usually also switches to a higher privilege level, thus transferring control to the operating system.

In a typical system, multiple interrupt sources exist. Hardware devices such as keyboards generate interrupts on key-presses and network cards generate interrupts on packet arrival, buffer over- and
under-runs. A dedicated timer device can be configured to periodically send interrupts. These interrupts are used by the operating system to run the scheduler and possibly context switch to another process. The CPU provides some mechanism to tell the operating system what kind of interrupt has happened, so that it can differentiate between interrupts.

**Interrupt delivery** This differentiation is achieved with the help of an interrupt controller. The CPU provides only very few physical interrupt pins. So it uses the interrupt controller to determine which interrupt occurred. The controller multiplexes multiple interrupt lines onto one. It provides some number of interrupt pins that a system designer deemed sufficient. To signal an interrupt condition, the controller asserts the interrupt line and puts the interrupt number (the so-called *interrupt vector*) on the bus. The CPU then fetches a word from the bus to determine the interrupt number. On modern bus based systems, the interrupt vector is directly encoded by the controller onto the bus message. On x86, this bus-fetch is done transparently to the programmer. An Intel CPU looks up the vector number in a memory-resident table which stores pointers to ISRs and directly jumps to the Interrupt Service Routine (ISR). The OS has to follow the interrupt table layout mandated by the CPU.

On ARM, only one interrupt entry point has to be provided by the OS. Once the interrupt is received by an ARM CPU, it will invoke the ISR. The OS reads explicitly a interrupt controller register to determine the vector number. The OS can then take appropriate action. The ARM software-based approach offers a greater flexibility in organizing interrupt handlers to the OS.

**Interrupt sharing** Usually, there exists a hard limit of interrupts the CPU can receive, and the namespace might, as on x86, be shared with hardware exceptions. Thus with an increasing number of interrupt sources, this namespace can get exhausted. In this case, most subsystems (notable PCI) support *interrupt sharing*. When interrupts are shared, two different interrupt sources (for example two different PCI devices) will trigger the same interrupt vector on the CPU. The
OS in turn polls all candidates. While it is possible that interrupts must be shared because the number of CPU vectors are exhausted, a more common case is that the subsystem itself has a bottleneck. For example PCI only has four physical interrupt lines. If more than four cards are plugged into the bus, they will necessarily have to share the interrupt lines.

In the early days of x86, the PC-AT platform defined that two cascaded PICs (see Section 3.3.4.1) should be present and assigned fixed meanings (such as a keyboard interrupt) to almost all of the 15 available interrupt lines. Once PCI was introduced, not enough pins on the PIC were left free. Hence, the PCI designers were forced to share interrupt lines. PCI also introduces small interrupt controllers (the PCI Link Device [PCI04]) that can reshuffle the interrupts in some limited way. The OS can, in some situations, with careful link device configuration, avoid the sharing of interrupt lines. Even if sharing is necessary, it still offers the flexibility to the OS to choose which cards will share the interrupts. For example, the OS can group together performance uncritical devices.

**Multicore** Multiprocessor and multicore systems are standard today and have more than one destination for interrupts. Also, processors need the ability to interrupt each other via a mechanism called IPIs. To provide these services, the interrupt controller had to be extended. On x86 the first iteration of this is called the Advanced Programmable Interrupt Controller (APIC) architecture. It is not a single chip as the PIC was, but a network of two different types of devices: IOAPICs [Int96, Int14] have incoming interrupt lines similar to the PIC, but instead of triggering an interrupt directly on the CPU, the IOAPIC communicates the interrupt to the LAPICs [Int97] using a bus. Each logical CPU (core) has exactly one LAPIC. The LAPICs turn a bus message - if the bus messages targets to the attached CPU - to a vector delivered to the CPU. These LAPICs also provide local interrupt sources, such as timers, but the interrupt it generates is only receivable by the core that is attached to the LAPIC. An interrupt from an IOAPIC can be sent to one, all, or a specific subset of CPU’s LAPICs. The precise
mechanism of this multicast system is described in Section 3.3.4.2. These controllers are connected in a non-trivial, platform-specific network.

An interrupt vector on ARM never changes its Interrupt Identifier (INTID), while on x86 an interrupt can usually be remapped (by an IOAPIC for example) before it is delivered. The ARM GICv2 [ARM16a] supports 1024 different interrupts but not all can be delivered to all cores, and vectors cannot be changed. The compatible CoreLink GIC-400 [ARM11] adds additional constraints on its configurability, the GICv3 exists in two variants [ARM16b] with implementation-defined limits on vector size and GICv4 adds virtualization support.

Message-signaled interrupts (MSI) To overcome the high latency involved with classical out-of-band interrupts, the limited vector number and the need for expensive dedicated physical wires MSIs were introduced [Cor09].

An Message Signaled Interrupt (MSI) is a memory write targeted to a special interrupt receiving address. The receiving interrupt controller appears as a memory mapped device. Since MSIs travel in-band with memory requests, MSIs have lower latency than dedicated wire interrupts - no extra synchronization between (for example) a Direct Memory Access (DMA) memory write and an interrupt trigger is necessary. Using a part of the physical address space for generating interrupts allows a huge number of interrupts.

MSI refers to the general mechanism of encoding interrupts as in-memory writes. PCI defines two mechanisms to generate MSI: MSI and MSI-X [PCI04]. These mechanisms allow PCI cards to trigger MSIs with an uniform configuration and delivery mechanism.

The first iteration of the PCI mechanism (MSI) was defined in specification version 2.2. It allowed to up to 32 interrupts per device. Configuration was done in the PCI configuration space, using a base address register and a base word. The interrupt number is then added to the word and written to the address once the device wanted to activate the interrupt.
To support an even higher number of interrupts and more flexible delivery, MSI-X was introduced. It allows up to 2048 interrupts using a memory-resident look-up table. Each entry specifies an address and data word, thus allowing an arbitrary destination. A feature that is not relevant on x86, but is present on ARM, allows an MSI to be received by each redistributor (which is similar to the LAPIC).

**Virtualization** Another recent hardware advance is the support of virtualization which allows running unmodified guest operating systems in a Virtual Machine (VM). While the concrete implementations differ between Intel, AMD, and ARM, the offered features are similar. The goal is to reduce the overhead when mapping hardware devices into such a VM. The IOMMU (Intel VT-d [Int16b] and AMD IOMMU [Adv]) is responsible for redirecting memory requests from mapped devices such that they align with the VMs guest-physical view of the memory. The IOMMU also has an interrupt rerouting facility. Using this facility, the hypervisor can safely isolate interrupts generated by a mapped device. This means that a guest operating system can not configure its mapped device such that it triggers an interrupt on the hypervisor or on another VM. IOAPICs can be located behind an IOMMU, but what changes their behavior and configuration layout [Int16b].

While the interrupt remapping of the IOMMU is important for safety and correctness, it does still incur performance overhead: When an interrupt is received, the hypervisor is invoked. The hypervisor then decides for which VM the interrupt is designated. To take the hypervisor out of the loop and deliver interrupts directly to a VM, recent IOMMUs contain support for posted interrupts. Using this technique, when an interrupt is received for a VM that is not currently running, the interrupt gets translated to a bit set operation in a memory-resident bitmap that contains pending interrupts for the VM. Once the VM is rescheduled, the CPU will trigger the interrupts set in this bitmap.

Device pass-through especially makes sense if a device supports Single Root I/O Virtualization (SR-IOV). In this PCI Express (PCIe)
extension, a physical device can be configured to present itself as many logical devices. With this technique, it is not necessary anymore to pass-through the full device, but a sub-functionality can be passed to a VM. For example, modern network cards provide a queue per virtualized device and each virtual device supports up to 2048 MSI-X based interrupts. In this scenario, each queue can be mapped independent of other queues to VMs.

**Discovery**  The OS must discover and correctly configure the interrupt network dynamically. Some topology data can be obtained from PCI discovery [PCI04] and ACPI [UEFI7], but it is incomplete or may not exist at all. devicetree [OH15] files are used by many OSes to work around this, but the file format has no clear semantics, is error-prone [Rut13], and despite containing controller information [Red+14] fails to capture configuration constraints or cover inter-processor interrupts. Even so, the proliferation of devicetrees shows that configuration, in particular topology description, is a problem.

After discovery, correct configuration of a modern computer is essentially a network routing problem with highly constrained switches, but current OS designs reflect a legacy of much simpler hardware.

Linux, for example, defines a single namespace of “IRQ numbers” for all interrupts and then attempts to map this to a strict hierarchy of interrupt controllers. “IRQ Domains” [Lik+16; BC05] map Linux IRQ numbers to hardware sources and implicitly hard-codes the topology. Device drivers are responsible for identifying the controllers they need to program (via a driver interface) to deliver interrupts correctly. The only supported case is to deliver an interrupt to all cores, and vector numbers are assumed to be the same across all cores. Constraints in interrupt routing are not well handled and are generally special-cased in the code. For instance, the driver for the PCI link device contains a configuration heuristic to determine which IOAPIC input to use. This heuristic is not guaranteed to minimize interrupt sharing. Our approach posts this problem to a constraint solver to determine an optimal configuration.
Similarly, FreeBSD refers to interrupts using the ACPI GSI enumeration resulting in clashes when naming MSI sources. The system assumes that MSIs directly reach the CPUs, which is no longer true since the introduction of the IOMMU [Bal08].

Chen et al. [Che+16] (and also eChronos [ALM15]) verify an interruptible operating system kernel. This includes a simple verified last level interrupt controller driver for masking and acknowledging interrupts. The focus of this dissertation is on the topology and configurability of the interrupt system. Apart from these, We are not aware of any work on formalizing the interrupt subsystem.

Stepping back, a better approach is to define a formal model. This model captures the complexity of modern interrupt subsystems and provides both a basis for verifying implementations and a template for engineering a correct solution that works across a wide variety of platforms. This dissertation describes a configurable interrupt model, partially expressed in an automated theorem prover, and uses a constraint solver to derive a configuration that satisfies routing constraints.
3.2. Static Interrupt Configuration

In this section, we present a static model (defined in Section 3.2.1) of interrupt forwarding, formalized in Isabelle/HOL, and explain how it addresses the challenges stated in the introduction. We show that this model captures real-world systems in Section 3.2.2. We then define equality preserving transformations and prove their correctness in Section 3.2.3.
3.2. Static Interrupt Configuration

3.2.1. Model and Syntax

In this section, we give a brief description of our model and refer to [Ach+17a] for a full description. Note that this model is designed to capture both, memory addressing and interrupt decoding. We use the term vector (vec) interchangeable for memory addresses and interrupt vectors. We model the system’s handling of emitted vectors by a decoding net: a directed graph where each node represents a hardware component. Interrupt vectors and memory addresses are natural numbers. Decoding of a vector starts at a particular node. Thus, a name is a vector qualified by the node at which it is decoded. We therefore define a name as a tuple of i) a node identifier (nodeid) and ii) an vector (vec). Nodes are labeled with natural numbers. A decoding net is then an assignment of nodes to identifiers.

Definition 1.

\[
\begin{align*}
\text{nodeid} &= \mathbb{N} & \text{vec} &= \mathbb{N} \\
\text{name} &= (\text{nodeid}, \text{vec}) & \text{net} : \text{nodeid} \to \text{node}
\end{align*}
\]

Hardware components accept vectors (e.g. a CPU that acts as an interrupt destination) and they translate vectors and pass them on (e.g. an interrupt controller). Nodes can also do both, accept and translate, but this applies only to nodes of the memory subsystem (e.g. caches). A node is completely defined by two properties: a set of accepted vectors and a set of names it decodes an input vector to. Formally:

\[
\begin{align*}
\text{node} &= \text{accept} : \{\text{vec}\} \\
&\quad \text{translate} : \text{vec} \to \{\text{name}\}
\end{align*}
\]

The result of accept is the set of vectors accepted by a node without forwarding; The result of translate is the set of translated names for each input vector. All definitions are formalized in Isabelle/HOL. As defined so far, this model captures the static state of the system. Dynamic aspects such as configuration options, concurrency, and caching can be modeled on top of the decoding net. We do this for interrupt controller configurability in Section 3.3.1.
The model allows a node to translate a vector to multiple outputs — this is how we express the multicast behavior of the interrupt network. In the introductionary x86 example (Figure 3.1) the IOMMU forwards the interrupt to two controllers, LAPIC₁ and LAPIC₂. Thus the translate of the IOMMU would map 230 to \{(LAPIC₁, 40), (LAPIC₂, 40)\}.

Also note that there is no unique identifier for an interrupt vector in this model. All interrupt vectors are qualified at a node, and the vector can change at each decode step.

Every decoding net defines a \textit{decode relation} and an \textit{accepted-names-set} (names accepted anywhere in the net):

\[(n', a'), (n, a)) \in \text{decode}(net) \iff (n', a') \in \text{translate}(net(n), a)\]

\[\text{accepted}(net) = \{ (n, a) : a \in \text{accept}(net(n)) \}\]

From these, we define the \textit{resolution function}, which maps an input name (a vector presented to a particular node) to a set of \textit{resolved names}: the nodes at which the input vector could end up being accepted, together with the translated input vector to that node.

\[\text{resolve}(net, n) = \{n\} \cap \text{accepted}(net) \cup \bigcup (n', n) \in \text{decode}(net). \text{resolve}(net, n')\]

This is the input name if and only if the start node accepts the start vector, and then all names reachable recursively via the decode relation. The termination of this recursion depends on the structure of the net, specifically the presence of loops. We present a necessary and sufficient condition for termination in Section 3.2.3.

Nets are expressed in the following concrete syntax (EBNF, terminals are bold). This corresponds to the abstract syntax of the Isabelle formulation.
3.2. Static Interrupt Configuration

\[
\begin{align*}
net_s &= \left\{ N \text{ is node}_s \mid N..N \text{ are node}_s \right\} \\
node_s &= \left[ \text{accept} \left[ \left\{ \text{block}_s \right\} \right] \left[ \text{map} \left[ \left\{ \text{map}_s \right\} \right] \right] \left[ \text{over} N \right] \right] \\
\text{map}_s &:= \text{block}_s \text{ to } N \left[ \text{ at } N \right] \left\{ , N \left[ \text{ at } N \right] \right\} \\
\text{block}_s &:= N-N
\end{align*}
\]

Here all translations are specified by mapping a contiguous block of input vectors (specified by range \( \text{base} - \text{limit} \)) to some output node, with all vector values shifted to a new block base vector. Nodes are specified by a finite set of accepting and mapping blocks and may be initialized using an \textit{overlay} (over). If so, the node maps all input vectors 1–1 to the specified Overlay Node, unless they are captured by an accept or map block. Nodes are assigned identifiers with \textit{is} or \textit{are} (for repeated nodes). One can use the identifier instead of the assigned number when referring to the node. The map specification allows us to declare multiple destinations, which is necessary to describe interrupt systems.

In the following section, we demonstrate that this syntax, together with our model, concisely represents the vector decoding of real hardware. In Section 3.2.3, we further show that our model supports reasoning about vector translation, for example by showing that networks can be reduced to a normal form, and that this reduction can be expressed by a simple algorithm on the concrete representation of the network i.e. that the representation \textit{refines} the model.
Chapter 3 - Interrupt Controller Configuration

3.2.2. Modeling Real Systems

We now express real systems (which we use for Barrelfish development) using the syntax from the previous section. We present the OMAP4460 SoC as representative of an ARM based system and an example configuration of an x86 system. Full models are available in the publication [Ach+17a].

We focus our modeling on software-visible hardware features. We represent a block of vectors in the form \( p_z/b \) with a (hex) prefix \( p \) followed by \( z \) zeros and the block size is \( 2^b \) e.g. \( 0x20000000-0x20000fff \) is represented as \( 20000_3/12 \) with a block size is 12-bits (4kB).

3.2.2.1. A Mobile Device SoC: the OMAP4460

We consider here the Texas Instruments OMAP4460 Multimedia SoC, a good example because it is representative of the varied class of mobile processors, has been used in products like the Amazon Kindle Fire 7” and PandaBoard ES, and has some of the best publicly available documentation. The software manual for this chip [Tex14] has 5820 pages.

An interrupt raised by a device on the OMAP4460 can be routed to a single designated core or one dynamically selected. Cores themselves can also send interrupts between themselves. Each core (A9, M3 and DSP) has a distinct view of the system: some resources are core-private while others appear at different addresses, some interrupts can be received by all cores, others not. We show a fragment of the static interrupt model in Figure 3.3).

Apart from the core local timer, also the SDMA device interrupts can not target all CPUs. The SDMA can generate four different interrupts, of which two can be sent to all processing elements, the remaining two can not be sent to the DSP core. We represent these multiple destinations as shown in the SDMA node. Also, the interrupts appear as different vectos in the M3 subsystem as in the Cortex A9 subsystem. We will now show how an interrupt from the SDMA device propagates through the system.
3.2. Static Interrupt Configuration

**SDMA triggers interrupt 2:** The SDMA device can generate four different interrupts. We model this as a node that maps consecutive vectors starting from zero. Once it triggers interrupt 2 in the SDMA node, the interrupt will be forwarded to multiple controllers with different vectors. Specifically, it will be multicast towards (SPI\textit{Map}, 12) and (\textit{NVIC}, 18) of the M3 subsystem. Since the \textit{NVIC}s are configured to ignore (mask) the interrupt, the nodes will neither accept nor map these interrupts. SPI\textit{Map} translates the vector and forwards the signal to (GIC, 44) and is finally accepted (IF\textsubscript{A9:0}, 44).

### 3.2.2.2. A Desktop PC

Our example desktop machine has a quad-core processor, 32GB of main memory and several I/O devices. We focus on the diversity of interrupt paths and vector formats. The relevant nodes (with 2 cores) are shown in Figure 3.4.

**x86 interrupt bus:** The vectors directed to the LAPICs follow a simplified encoding. The most significant byte dictates the destination LAPIC, the remaining bytes indicate the interrupt vector to be delivered. Nodes that send messages to the interrupt bus must be connected to all the LAPICs. In Figure 3.4 this is expressed with the wildcard LAPIC\textsubscript{C:*}.

**Message-signaled interrupts:** The GPU issues MSIs, memory writes to a platform-specific address range with a data word. We start with (GFX\textsubscript{INT}, 0) which is translated to a memory write to (MSIRECEIVER, 0\textbackslash xfee002b0029). We model this MSI as the concatenation of the 32-bit address and the 16-bit data word since MSIRECEIVER is able to distinguish both. There is nothing that stops a MSI from using the full address width, but current systems use 32 address bits and 16 data bits. The MSIRECEIVER transforms these memory writes back to a regular interrupt message forwarded to (LAPIC\textsubscript{C:*}, 0x7d), which is discarded by LAPIC\textsubscript{1} but forwarded by LAPIC\textsubscript{0} to Core\textsubscript{0}.

**Legacy interrupt path:** The EHCI USB controller raises a legacy interrupt, which appears at EHCI\textsubscript{INT}. It propagates to the PCI link.
device LNKA with vector zero, which redirects it to the IOAPIC with vector 4. The IOAPIC is configured to forward interrupt number 4 to the bus. The message targets core 0, thus the highest byte is 0, with vector 48, thus the lowest two bytes encode $48 = 0x30$. Alternatively, the RTC device follows a similar path to the EHCI but it is directly connected to the IOAPIC. The decoding steps are $(RTC_{INT}, 0) \rightarrow (IOAPIC, 8) \rightarrow (LAPIC_{C:0}, 0x28) \rightarrow (Core_0, 0x28)$.

**Vector sharing:** The ARM platform (as in Section 3.2.2.1) separates interrupts generated by peripheral devices and inter-processor interrupts initiated by software. The Intel x86 platform in contrast does not, so for each core, we split inbound $(LAPIC_{C:*})$ and outbound $(C:*_{INT})$ interrupts into separate nodes (see Section 3.2.3 for proof). In our example, $C:1_{INT}$ triggers an interrupt, which decodes to the same destination as the EHCI$_{INT}$. We start with $(C:1_{INT}, 1)$ which is directly forwarded to the accepting node $(LAPIC_{C:0}, 48)$. Sharing may be unavoidable: an MSI device can issue up to 2048 different interrupts while an x86 core can only distinguish 256 vectors.

**Core Local Interrupts:** The LAPIC not only decodes bus messages but also has a small number of local interrupt sources, such as a timer. We express this in the LVT* (local vector table) nodes. These interrupts can not be sent to any other core, thus in our model they target the core nodes directly.
3.2. Static Interrupt Configuration

SDMA is map [0 to SPIMap at 12 to INTC at 18

to NVIC0 at 18 to NVIC1 at 18.
1 to SPIMap at 13 to INTC at 19

to NVICM3:0 at 19 to NVICM3:1 at 19.
2 to SPIMap at 14 to NVIC0 at 20 to NVIC1 at 20,
3 to SPIMap at 15 to NVIC0 at 21 to NVIC1 at 21]

GPT5Int is map [0 to SPIMap at 41 to INTC at 41]

GIC is map [44 – 45 to IFA9:0 at 44, 46 – 47 to IFA9:1 at 46, . . .]

A90 is map [0 to IFA9:0 at 0]

A91 is map [0 to IFA9:0 at 0]

T0 is map [0 to IFA9:0 at 29]

T1 is map [0 to IFA9:1 at 29]

M3MMU is map [0 to SPIMap at 100]

SPIMap is map [0 – 987 to GIC at 32]

INTC, NVIC* are accept []

IF* are accept [0 – 1020]

Figure 3.3.: Overview and model of the interrupt system of the Texas
Instrument OMAP 44xx SoC
**Chapter 3 - Interrupt Controller Configuration**

**IOAPIC** is map [4 to LAPIC\(_C:*\) at 0x000030, 8 to LAPIC\(_C:*\) at 0x010040]

**MSI RECEIVER** is map [0xf0e002b00029 to LAPIC\(_C:*\) at 0x00007d]

**GFX\(_\text{INT}\)** is map [0 to MSI RECEIVER at 0xf0e002b00029]

**IOAPIC** is map [4 to LAPIC\(_C:*\) at 0x000030, 8 to LAPIC\(_C:*\) at 0x000028]

**C:0\(_\text{IP}\)** is map [251 to LAPIC\(_C:*\) at 0x0100fb]

**C:1\(_\text{IP}\)** is map [251 to LAPIC\(_C:*\) at 0x0000ff]

**EHCI\(_\text{INT}\)** is map [0 to LNKA]

**LNKA** is map [0 to IOAPIC at 4]

**RTC\(_\text{INT}\)** is map [0 to IOAPIC at 8]

**LAPIC\(_C:0\)** is map [0x000020 - 0x0000ff to Core\(_0\) at 0x20]

**LAPIC\(_C:1\)** is map [0x010020 - 0x0100ff to Core\(_1\) at 0x20]

**CORE\(_*\)** are accept [0x20 - 0xff]

**LVT:0** is map [0 to Core\(_0\) at 0x50]

**LVT:1** is map [0 to Core\(_1\) at 0x50]

---

**Figure 3.4.:** Schematic overview and model representation of the desktop PC
3.2.3. Reductions and Algorithms

The purpose of this model is to accurately represent the complex structure of address resolution and interrupt vector routing in a format that can be easily generated and manipulated at runtime. We now demonstrate that the model also has nice logical properties: it is amenable to formal analysis and the verification of routines that interpret or manipulate the concrete syntax. All results presented here are verified.

3.2.3.1. Termination

The underlying model, as introduced in Section 3.2.1, is strictly graphical—it is defined entirely by the decode relation and the accept set. This permitted us to directly specify the hardware behavior, including decoding loops, but says nothing directly about an agent’s view of the system. To which nodes (e.g. cores) is such and such a vector directed? The resolve() function provides the link, giving a mapping from local names (vectors relative to a viewpoint) to global names (nodes that may accept the vector and the local vector at which they accept it).

HOL is a logic of total functions, and we can thus express the mapping from (nodeid, vector) to (a set of) resources as a function if, and only if, the decoding process terminates. This occurs when every path in the decode relation beginning at the input vector is finite. We express this as the existence of some well-formed ranking function $f : \text{name} \rightarrow \mathbb{N}$, that decreases on every step of the decode relation:

$$\text{wf_rank}(f, n, net) \iff \forall x, y. ((y, x) \in \text{decode}(net)) \land (x, n) \in \text{decode}(net)^* \rightarrow f(y) < f(x)$$

Termination follows as $f$ is bounded below by 0 (here $\text{dom}(net, n)$ means that resolution terminates from $n$ i.e. the arguments are in the domain of the function:

$$\exists f. \text{wf_rank}(f, n, net) \iff \text{dom}(net, n)$$
The duality between the graphical and operational views of a node is fundamental: the result of any well-defined resolution is the set of names reachable via the decode relation (i.e., lies in the image of the reflexive, transitive closure of the relation), that refer to actual resources (i.e., are in the accept set):

\[
\text{dom}(\text{net}, n) \rightarrow \text{resolve}((\text{net}, n)) = \text{accepted}(\text{net}) \cap (\text{decode}(\text{net})^{-1})^*(\{n\})
\]

This result lets us freely substitute one view of the system for another.

### 3.2.3.2. Normalization and Refinement

To use this model, we need efficient algorithms in the OS to manipulate it e.g. calculate the set of visible resources from a processor. This information is implicit in the graphical model, but not easily accessible. We might, for example, wish to produce a flattened representation that preserves the view from each processor, while making such a query efficient. One way to achieve this is to split all nodes into nodes that only accept vectors (resources) and ones that only map to other nodes. Then merging or flattening the mapping nodes gives us the desired result.

We would like to verify such algorithms. We show here that we can define and verify equivalence-preserving transformations on the semantic model, together with a notion of refinement. In the remainder of this section, we demonstrate equivalence-preservation and refinement for the first step: splitting, with reference to the accompanying sources. Further results regarding flattening are also provided in the Isabelle sources for the interested reader.

Two nets are view-equivalent, written \((f, \text{net}) \sim_S (g, \text{net}')\) if all observers in \(S\) have the same view (i.e. the results of \(\text{resolve()}\) are the same), modulo some renaming \((f\) and \(g\)) of the accepting nodes.

Let \(c\) be greater than the label of any extant node. The split net is
then defined as:

\[
\text{accept}(\text{split}(\text{net}), \text{nd}) = \emptyset
\]
\[
\text{accept}(\text{split}(\text{net}), \text{nd} + c) = \text{accept}(\text{net}, \text{nd})
\]
\[
\text{translate}(\text{split}(\text{net}), \text{nd}, a) = \{(\text{nd} + c, a) : a \in \text{accept}(\text{net}, \text{nd})\} \cup \text{translate}(\text{net}, \text{nd}, a)
\]
\[
\text{translate}(\text{split}(\text{net}), \text{nd} + c, a) = \emptyset
\]

This new net is view-equivalent to the original, with names that were accepted at \(\text{nd}\) now accepted at \(\text{nd} + c\), and no node both accepting and translating addresses:

\[
(\text{nd} \mapsto \text{nd} + c, \text{net}) \sim_S (\emptyset, \text{split(\text{net})})\] (3.1)

Splitting on the concrete representation is a simple syntactic operation. Each node is replaced as follows:

\[
\text{nd is accept A map M over O} \mapsto
\]
\[
[\text{nd} + c \text{ is accept A, nd is map M(nd} \mapsto \text{nd'}) over O]
\]

Refinement is, as usual, expressed as the commutativity of the operations (here \text{split()} and \text{split}_C()) with the state relation (here \text{parse()}), which constructs a net from its syntactic representation:

\[
\text{split(\text{parse(s)})} = \text{parse(\text{split}_C(s))}\] (3.2)

Combining \text{Equation 3.1} with \text{Equation 3.2}, we have the desired result, that the concrete implementation preserves the equivalence of the nets constructed by parsing:

\[
(\text{nd} \mapsto \text{nd} + c, \text{parse(s)}) \sim_S (\emptyset, \text{parse(\text{split}_C(s))})\] (3.3)

Together with the equivalent result for flattening, we can verify that the nodes that we read directly from the transformed model are exactly those that we would have found by (expensively) traversing the original hardware-derived decoding net for all vectors.
3.3. Interrupt System Configuration

In this section, we tackle the problem of *configuring* the interrupt subsystem.

We first extend the static model (Section 3.2) with configuration options (Section 3.3.1). We then realize this model in Prolog and an interval constraints solver (Section 3.3.2). We then define a DSL (Section 3.3.3) that describes the per-controller configuration options. Finally, we use this DSL to express a large number of real controllers (Section 3.3.4), to show the generality of the proposed constructs.

This serves as a basis for implementing a flexible interrupt controller scheme in Barrelfish (Section 3.4).
3.3. Interrupt System Configuration

3.3.1. Configurable Model

The model presented in Section 3.2 provides us with a description of a single configuration of the interrupt delivery network. To analyze the configurability of the interrupt network, we need a more expressive model. Our goal is to refine the model to a representation that describes individual interrupt controller configurability isolated of the configurability of other controllers and is suitable for a solver to produce configurations.

We present the detailed refinement of the static model to the configurable one in Appendix B. The brief overview presented here explains the configurable model in enough detail to understand the following chapters.

On the most abstract level, we define admissible configurations of a system as a set of \( \text{net} (\text{net}_0 : \{\text{net}\}) \). A concrete configuration is an element of that set. We also define the decode relation on \( \text{net}_0 \), a name decodes to another if all the configurations decode those two names:

\[(a, b) \in \text{decode}_0 \text{net}_0 \iff \forall n \in \text{net}_0. (a, b) \in \text{decode}(n)\]

A set of nets is useful in theory and any model of a configurable system should have such an interpretation as set of nets. However, in practice there are additional concerns: Configuration happens per controller and many controller instances share the same configuration options (e.g. they belong to the same class). Furthermore we observe that the configurability can be further split into two classes: distinctC and explicitC. Ultimately, we propose to express a configurable net as defined in Definition 2.
Definition 2. *net5* and *nodeconf*

\[
\begin{align*}
nodeid &: \mathbb{N} \\
\text{classid} &: \mathbb{N} \\
\text{port} &: \mathbb{N} \\
\text{nodeconf} &: \text{explicitC} [\mathbb{N}] \mid \text{distinctC} \text{ vec} \rightarrow (\text{port}, \text{vec}) \\
\text{acc} &: \text{nodeid} \rightarrow \{\text{vec}\} \\
\text{conf} &: \text{nodeid} \rightarrow \{\text{nodeconf}\} \\
\text{inst} &: \text{nodeid} \rightarrow \text{classid} \\
\text{conn} &: \text{nodeid} \rightarrow \{(\text{port}, \text{nodeid})\} \\
\text{classE} &: \text{classid} \rightarrow [\mathbb{N}] \rightarrow \text{vec} \rightarrow \{(\text{port}, \text{vec})\} \\
\text{classD} &: \text{classid} \rightarrow \text{vec} \rightarrow \{(\text{port}, \text{vec})\} \\
\text{net5} &: (\text{acc}, \text{conf}, \text{classE}, \text{classD}, \text{inst}, \text{conn})
\end{align*}
\]

The *net5* definition works as follows: Each controller is represented using its *nodeid*. The *acc* returns the set of accepted vectors of a controller. Furthermore, each controller has a set of possible configurations (*conf*). This is the only configurable part of the network, everything else is static. This configuration, together with the class of the controller (given by *inst*) defines the local forwarding behavior. The local forwarding behavior does not yet produce a name (an input vector at another controller) but a vector relative to a localized output port. The *conn* function connects the local output port with the input of another controller. This is how the *net5* definition can be lifted to a set of nets. In the appendix the definition of *lift4* (Definition 14) formalizes this description.

This formalization has a set of benefits: It is possible to reason about the controller configurability in isolation of the network. Controller classes are known before runtime, for example from a datasheet, and can be statically described. In Section 3.3.3 we make use of this fact and define a DSL to describe these classes. At runtime, an OS only has to determine which instances are present and how they are connected (by defining the *inst* and *conn* functions).
3.3. Interrupt System Configuration

The node configuration \( \text{nodeconf} \) has two different options. The explicit class has no restrictions, it interprets the parameters \( \text{distinctC} \) in any way. For example, it can use the integer list in the corresponding configuration to encode a base register and map consecutive addresses with respect to that base register. This is precisely what the MSI controller is doing.

Another special case of a generic node is the fixed function: Its \( \text{nodeconf} \) is empty and \( \text{classE} \) yields always the same \( \text{vec} \rightarrow \{(\text{port}, \text{vec})\} \) function.

\( \text{distinctC} \) can be interpreted as a special case of \( \text{explicitC} \). However this distinction is useful for constraint solving. The \( \text{classD} \) function can be seen as a predicate over tuples of type \( (\text{in} : \text{vec}, \text{port}, \text{out} : \text{vec}) \) (a mapping). If the partial configuration function of \( \text{distinctC} \) returns the same \( (\text{port}, \text{out}) \) pair when applied to \( \text{in} \), the corresponding result is the singleton set \( (\text{port}, \text{out}) \). \( \text{classD} \) restricts only the type of mappings that are possible. This has several consequences: Distinct configurable nodes either return the empty set or a singleton, they never translate to multiple outputs. If a mapping fulfills the predicate \( \text{classD} \), the mapping can be added to the configuration - affecting only the mapping of the same input vectors, but without affecting any other mapping. Analogously, the removal of a mapping is always possible from the configuration without affecting anything else. While \( \text{distinctC} \) is a special case of \( \text{explicitC} \), it can be used as a compact representation of a large but sparse configuration space.

In practice, controllers that perform any sort of lookup in a table or an in memory radix tree can be expressed as \( \text{distinctC} \). For example the IOAPIC has 24 input lines. For each input line, a separate configuration register exists. Thus each input can be configured independently from other inputs. This makes this controller expressible as \( \text{distinctC} \). Note that \( \text{classD} \) allows to restrict the input and output, for example to require that the output is formatted as a x86 compatible bus message.

In the following sections, we will realize this representation in Prolog and incorporate information from different sources: \( \text{acc}, \text{inst}, \text{conn} \) will be instantiated at runtime using hardware discovery or static description. \( \text{classD} \) and \( \text{classE} \) will be described in a DSL and
compiled to a Prolog representation. \textit{conf} will be determined using a constraint solver given \textit{route} constraints on the net. We did not define \textit{route} on the \textit{net}_5, as we will lift the representation in Prolog to that of \textit{net}_4 before applying route constraints.
3.3.2. Prolog Representation

We realize the model closely following Definition 13 and Definition 11. Instead of natural numbers, we use Prolog atoms for \textit{nodeid}. The predicates representing \textit{conn} and \textit{acc} are straight-forward representations of the model. \textit{classid} consists of an atom and a list of integers of constant parameters. This \textit{classid} is then used to define the instance to class mapping \textit{inst}.

\textit{conf} is represented using a hash to store a configuration for each instance. Values in this hashmap can be both, instantiated integers or (constrained) variables for the solver. We use the latter case to express not just a fixed configuration, but a set thereof.

We use the solver of the Interval Constraints (IC) library shipped with EclipseCLP. Thus the \textit{In}, \textit{Out}, \textit{Port}, and \textit{Conf} variables will be constrained variables that need to be solved with a corresponding solver from the library. The library also defines the constraining operators by using the \# prefix. For example, to constrain a solver variable \( X \) to be between 0 and 10 the expression \( X \# >= 0, X \# < 10 \) is used.

```prolog
1 cmap_explicit(pciMsi,[],C,In,Out,0) :-
% Two configuration variables
2 C = [BaseAddr,BaseData],
3 BaseAddr #>= 0, BaseAddr #< 2^32,BaseData #>= 0,
4 BaseData #< 2^16,
5 In #>= 0, In #< 2^4,
6 % Out encodes a memory write with
7 % 32 bit address + 16 bit data
8 Out #>= 0, Out #< 2^48,
9 Out #= BaseAddr*65536 + BaseData + In.
```

Listing 3.1: Explicit predicate of the MSI controller

For \textit{explicit} controllers, the global configuration hash \textit{conf} stores a list of integers, closely following the model definition. \textit{classE} is encoded in the \textit{explicit configurable map} predicate \texttt{cmap_explicit(ClassId, Params, Conf, In,Out,Port)} which follows closely the model definition. The model returns a set of tuples, while in Prolog this predicate evaluates true for all the set members. Listing 3.1 shows the predicate of the
MSI controller. The MSI controller has a configurable base address and a base word, thus two variables are stored in the configuration \( C \). The last line defines the relation of all the input and outputs. The output of this controller is a memory write, which we encode as 32 address bits concatenated with 16 data bits. The multiplication by 65536 shifts the address left by 16 bits.

In the **distinct** case, the configuration is represented using two lists of the same length, the input- and output list. Each index in these lists represents an input-output pair. Unlike in the generic case, these values will always be integers (and not variables). This is a slight deviation from the model, instead of having a partial function we store pairs of input and output. We ensure that there are no duplicates in the list. Furthermore, instead of computing the intersection \( \{ p(\text{vec}) \} \cap \text{classD}(\text{classid}, \text{addr}) \), see [Definition 14], we ensure that \( p \) only ever contains mappings that are in the \( \text{classD} \) relation. \( \text{classD} \) is represented in \( \text{cmap_distinct}(	ext{ClassId}, \text{Params}, \text{In}, \text{Out, Port}) \), using the same set representation as before. [Listing 3.2] shows an example of the predicate for the first generation IOAPIC with 16 addressable CPUs. The first two lines in the predicate body are used for encoding a x86 bus message into one integer. The third line encodes the actual controller constraints: As the IOAPIC has 24 physical inputs, each input must be between 0 and 24. The output is encoded as bus message, consisting of the three fields: \( \text{DestMode} \),

```
cmap_distinct(ioapic, [], In, Out, 0) :-
    % Encode DestMode, Dest, Vec into Out.
    Out#= DestMode*65536 + Dest*256 + Vec*1 +
    0, DestMode#= 0, DestMode #< 2, Dest#= 0, Dest #<
    256, Vec#= 0, Vec #< 256,
    % Bounds needed for encoding.
    DestMode #>= 0, DestMode #< 2^1, Dest #>= 0, Dest #<
    2^8, Vec #>= 0, Vec #< 2^8,
    % The actual predicate
    In#=0, In#<24, Vec#=32, Dest#<16.
```

[Listing 3.2: Distinct predicate of the IOAPIC]
Vec and Dest. Dest must be between 0 and 16 as this IOAPIC can address at most 16 CPUs.

```prolog
% cmap for distinct configurable controllers
% Conf        - System Configuration (a hash)
% (Inst, In)   - Inst is an atom describing the
%                controller instance
% (Inst, Port, Out) - Vector Out at Port
cmap(Conf, (Inst, In), (Inst, Port, Out)) :-

  % Obtain instance configuration
  concat_atoms(Inst, '_in', InTag), % InTag is an atom like ioapic_in
  concat_atoms(Inst, '_out', OutTag),
  (not hash_contains(Conf, InTag) ->
    hash_insert(Conf, InTag, []) ; true),
  (not hash_contains(Conf, OutTag) ->
    hash_insert(Conf, OutTag, []) ; true),
  hash_get(Conf, InTag, Ins),
  hash_get(Conf, OutTag, Outs),

  % Apply class constraint
  instance_class(Inst, Cls, Params),
  cmap_distinct(Cls, Params, In, Out, Port),

  ( %Reuse existing element
    (element(Pos, Ins, In), element(Pos, Outs, Out))) ;
  %Insert new, all elements must be different
    ( NewIns = [In | Ins], alldifferent(NewIns),
      NewOuts = [Out | Outs], alldifferent(NewOuts),
      hash_set(Conf, InTag, NewIns),
      hash_set(Conf, OutTag, NewOuts)
    )).

% This is cmap for generic configurable controllers
cmap(Conf, (Inst, In), (Inst, Port, Out)) :-
  % We use Inst as an identifier to get a list of
  % variables
  (hash_contains(Conf, Inst) -> hash_get(Conf, Inst, InstanceConf) ; true),
```

3.3. Interrupt System Configuration
Using the predicates `cmap_explicit` and `cmap_distinct`, we construct a type independent `cmap(Conf, Inst, In, Port, Out)` as shown in Listing 3.3. There are two rules for this predicate, one for each type. The type of the instance is determined by its associated class. The predicate is simple to implement for the explicit type, the configuration of the instance is looked up in the configuration hash `conf` and passed on the `cmap_explicit`. For the `distinct` case, we have to consider again two cases: If the input is in the input list, then the output must be at the same position in the output list. If this is not the case, we construct solver constraints: The input must be different from any configured input and input/output must fulfill the predicate `cmap_distinct`.

Using `cmap` we construct the `decode4` (see Definition 11) relation by requiring the output to be mapped with the `conn` predicate. The last and ultimately useful predicate is `route(Conf, Hops, Terminal, HopsOut)`, which is true if: `Hops` is a prefix of `HopsOut`, `HopsOut` is a list of names such that two consecutive elements fulfill `cmap` using the configuration `Conf`, and the last element of `HopsOut` unifies with `Terminal`.

The `route` predicate is resatisfiable until it explored all connections that start from `Hops`, and end in terminal, it does so by performing a Depth-first search (DFS) in the controller network, that we have implicitly encoded in the decode predicate. The configuration (`Conf`) does contain constrained variables, and we need to pass them to the solver first (using the labeling predicate from the IC EclipseCLP library).

Our implementation of the generic Prolog predicates (`cmap`, `route` etc.) consists of 99 lines. The description of the controllers consists of 370 lines DSL, which compiles to 244 lines Prolog.
3.3.3. Interrupt Controller DSL

The goal of the DSL is the description of interrupt controller classes and their associated *explicit* or *distinct* configuration options.

The compiler turns this into the corresponding Prolog predicate. It supports bitfields (*struct*) as input and output and generates automatic de- and encoding of the *struct* to an integer. It also issues the correct IC library predicates and maps bitwise operations to a custom delayed goal, as these are not directly supported in the IC library. Decoupling the controller description from the implementation also ensures that no other Prolog facilities are used and that the predicates are indeed *pure*. It also opens the door for using another solver and generating interface code, although we have not implemented this.

The DSL also permits expression of connections and instances, which will be used only for illustration purposes but not in the Barrelish implementation.

The top-level elements are *class*, *instances*, and *connections*. Their syntax is informally given in [Listing 3.6](#), the placeholders of this informal description are given in EBNF in [Listing 3.7](#). The predicate syntax uses the Prolog logical operators: , is a conjunction (*logical and*); ; is a disjunction (*logical or*). If the type declaration (*typedecl*) consists of an integer, it describes the bit width of the input (or output). If it is a *struct* the elements are accompanied with bit widths. Also identifiers follow Prolog syntax, in particular variables must start with an uppercase character.

The compiler is implemented in Haskell with the Megaparsec libraries. Before the output is generated, the compiler performs the following checks:

- It ensures that the configuration predicate uses only valid variables. These are configuration variables, class parameters, input, output, and the implicitly available *Port* variables. If the input (respectively output) is of scalar type, then the variable name *In* (*Out*) can be used in the predicate. If the corresponding types are *structs*, only the member variables can be used.
Chapter 3 - Interrupt Controller Configuration

- The compiler checks that no variables are defined twice in a scope, for example by a duplicate struct member.
- Instantiations use a valid class name and the right number of parameters.
- The connections refer only to existing instances.

Listing 3.4 contains an example controller definition in the DSL. The compiler will compile this into the Prolog output Listing 3.5. The compiler has replaced the struct with a integer encoding and inserts necessary bounds on the variables. If we simplify the predicate by removing unnecessary parentheses, it matches our manually created example Listing 3.2.

```c
class ioapic() {
    In : 8;
    Out : struct {DestMode: 1, Dest : 8, Vec : 8};

    Conf = distinct {
        In #>= 0, In #=< 24,
        Port=0,
        Vec #>= 0, Vec #=< 255,
        Dest #>= 0, Dest < 16; //Limited to 4 bits
    }
}
```

Listing 3.4: DSL example

```prolog
cmap_distinct(ioapic,Params,In,Out,Port) :-
    In #>= 0, In #=< 2^8,
    Out#= DestMode*65536 + Dest*256 + Vec*1 + 0,
    DestMode#= 0, DestMode #< 2,
    Dest#>= 0, Dest #< 256,
    Vec #>= 0, Vec #< 256,
    Params = [],
    (((((In)#>=0)), ((In)#=<24)), ((Port)#=0)),
    ((Vec)#>=32)), ((Vec)#=<255)),
    (((((DestMode)#=0)), ((Dest)#>=0)),
    ((Dest)#=<16))); (((DestMode)#=1)),
    ((Dest)#>=0)), ((Dest)#=<16))).
```

Listing 3.5: Example Prolog output
3.3. Interrupt System Configuration

```c

class <classname>(<varlist>) {
    In : <typedecl>;
    Out : <typedecl>;
    Conf = <distinct | explicit(<varlist>)> { <pred> };
}

instances {
    <classname> <instancename>(<parameters>);
    ...
}

collections {
    <instancename>.<int> -> <instancename>;
    ...
}
```

Listing 3.6: Top-level syntax

```
⟨char⟩ ::= [‘a’-‘Z’] | [‘A’-‘Z’]
⟨id⟩ ::= ⟨char⟩ (⟨char⟩ | [‘0’-‘9’] | ‘_’)*
⟨varid⟩ ::= [‘A’-‘Z’] (⟨char⟩ | [‘0’-‘9’])*
⟨classname⟩ ::= ⟨id⟩
⟨instancename⟩ ::= ⟨id⟩
⟨int⟩ ::= ([‘0’-‘9’]) | (‘0x’ [‘0’-‘9’, ‘a’-‘f’])
⟨parameters⟩ ::= ⟨int⟩ (‘,’ ⟨int⟩)*
⟨varlist⟩ ::= ⟨varid⟩ (‘,’ ⟨varid⟩)*
⟨uOp⟩ ::= ‘-’ | ‘+’
⟨bOp⟩ ::= ‘&’ | ‘|’ | ‘*’ | ‘/’ | ‘+’ | ‘-’
⟨cEx⟩ ::= ⟨aEx⟩ (‘>=’ | ‘>’ | ‘<=’ | ‘<’ | ‘==’) ⟨aEx⟩
⟨aEx⟩ ::= ‘(’ ⟨aEx⟩ ‘)’ | ⟨varid⟩ | ⟨int⟩ | ⟨uOp⟩ ⟨aEx⟩ | ⟨aEx⟩
⟨bOp⟩ ⟨aEx⟩
⟨pred⟩ ::= ⟨pred⟩ (‘;’ | ‘,’) ⟨pred⟩
| ⟨aEx⟩ ⟨cEx⟩ ⟨aEx⟩
```

75
Listing 3.7: Syntax elements of the interrupt controller description language

\[
\langle \text{typedef} \rangle ::= \langle \text{int} \rangle \\
\text{ | 'struct' '{' \langle \text{varid} \rangle ':' \langle \text{int} \rangle (',' \langle \text{varid} \rangle ':' \langle \text{int} \rangle)* '{'}
\]
3.3.4. The Controller Zoo

We present now how we can express real interrupt controllers in the model. Table 3.1 lists all the controllers.

**Expressing Real Controllers** Note that there is no unique representation of an interrupt system: identifiers of ports and vectors can be changed while preserving the controller’s semantics and even splitting and merging of controllers is possible – it is theoretically possible to express an entire interrupt system using a single controller and a complex predicate for valid mappings. Practical considerations of modularity, reuse, and readability determine a “good” representation.

Our model abstracts away from the fact that there might be complex interactions between controllers. For example, the legacy Intel PIC controller can query a cascaded interrupt controller to determine which vector was triggered downstream. Another example are bus messages on the APIC bus that can be used to address the lowest priority CPU. In this lowest priority mode, runtime information (the CPUs executing priority) is necessary to determine to which CPU the interrupt will be delivered. In practice, this is done using multiple bus cycles, but the execution priority is outside of our model scope. We abstract away from this behavior: In our model, an interrupt is translated forward and all information is contained in the interrupt message. This makes the analysis and generation of configuration easier, and as we will see in Section 3.3.4 works well.

There are multiple options to incorporate this complex behavior. In the case of delivering lowest priority interrupts, we model the interrupt to be delivered to all addressed cores. Thus in our model, lowest priority is indistinguishable from a broadcast interrupt. This has the benefit of the interrupt message delivered to local APICs to resemble closely the description in the Intel datasheet, and thus being easily understandable by anyone familiar with the APIC interrupt delivery system.

Some cases, such as the IOAPIC are represented with more than one model. This choice was made because the IOAPIC changes its connections and configuration format drastically if its output is
connected to an IOMMU. In practice, it is so different, that an IOAPIC behind an IOMMU needs another driver, as even the bit layout of the memory mapped configuration layout is changed. Client code must choose the correct model when the connections are discovered.

3.3.4.1. X86 Legacy Controllers

The Intel 8259A PIC [Int88] was originally introduced for the Intel 8086 CPU in 1978. It is a discrete chip that offers 8 input lines. The interrupt forwarding behavior can not be configured, a given line will always trigger a given exception in the CPU. It is of no practical relevance today anymore, as it does not support multicore systems. Despite this drawback, x86 systems are backward compatible and still boot in the PIC PC/AT mode. Further, expressing the controller in our model is useful for validating the generality of the model.

Cascading  The chips can be configured in a cascade, which extends the available interrupt lines up to 64. In a cascade up to 8 slave PICs are connected to one master. The interrupt output pin of the slaves is connected to the interrupt request line of the master. Additionally, the three Cas lines are connected to all slaves and used to find the downstream PIC in case of a downstream interrupt request. Abstracting a bit from the wiring, we can model the input of the PIC master as a struct with the following components: IsCas is one if the interrupt triggered from a cascaded slave. If IsCas is one, then Cas describes the cascading address. Vec describes which interrupt line has been triggered at the slave PIC.

The master determines the output vector that is sent to the CPU using the IsCas, Cas, and Vec variables and adds 32 as x86 CPUs treat the first 32 vectors as exceptions.

```plaintext
1 // The output format of the master is a 8bit x86 CPU vector.
2 // Between slaves and master the messages are exchanged in the following format:
3 // IsCas : 1, - Is this from a cascaded slave (as opposed to directly triggered interrupts)?
4 // Cas : 3, - the cascade address (only matters
```
### 3.3. Interrupt System Configuration

<table>
<thead>
<tr>
<th>Controller Name</th>
<th>Input Width</th>
<th>Configurability</th>
<th>Output Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PIC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIC Master</td>
<td>3 bit</td>
<td>fixed</td>
<td>8 bit</td>
</tr>
<tr>
<td>PIC Slave</td>
<td>3 bit</td>
<td>fixed</td>
<td>8 bit</td>
</tr>
<tr>
<td>PC/AT PIC</td>
<td>4 bit</td>
<td>fixed</td>
<td>8 bit</td>
</tr>
<tr>
<td><strong>APIC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPIC (P4)</td>
<td>17 bit</td>
<td>explicit</td>
<td>8 bit</td>
</tr>
<tr>
<td>IOAPIC (P4)</td>
<td>8 bit</td>
<td>distinct</td>
<td>17 bit</td>
</tr>
<tr>
<td>LAPIC LVT</td>
<td>3 bit</td>
<td>distinct</td>
<td>8 bit</td>
</tr>
<tr>
<td>LAPIC ICR</td>
<td>19 bit</td>
<td>fixed</td>
<td>17 bit</td>
</tr>
<tr>
<td><strong>PCI (legacy)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCI Link</td>
<td>2 bit</td>
<td>distinct</td>
<td>8 bit</td>
</tr>
<tr>
<td>PCI Conf</td>
<td>2 bit</td>
<td>explicit</td>
<td>1 bit</td>
</tr>
<tr>
<td><strong>Message Signaled Interrupts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSI Receiver</td>
<td>48 bit</td>
<td>fixed</td>
<td>17 bit</td>
</tr>
<tr>
<td>PCI MSI</td>
<td>4 bit</td>
<td>explicit</td>
<td>48 bit</td>
</tr>
<tr>
<td>PCI MSI-X</td>
<td>11 bit</td>
<td>distinct</td>
<td>48 bit</td>
</tr>
<tr>
<td>PCI MSI Conf</td>
<td>48 bit</td>
<td>explicit</td>
<td>48 bit</td>
</tr>
<tr>
<td><strong>Intel IOMMU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRTE Index</td>
<td>48 bit</td>
<td>fixed</td>
<td>16 bit</td>
</tr>
<tr>
<td>IRTE</td>
<td>16 bit</td>
<td>distinct</td>
<td>17 bit</td>
</tr>
<tr>
<td>IOAPIC IRTE</td>
<td>8 bit</td>
<td>distinct</td>
<td>16 bit</td>
</tr>
<tr>
<td><strong>ARM GIC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GIC CPU IF (AF)</td>
<td>16 (49) bit</td>
<td>fixed</td>
<td>16 bit</td>
</tr>
<tr>
<td>GIC DIST (AF)</td>
<td>10 bit</td>
<td>distinct</td>
<td>10 (49) bit</td>
</tr>
<tr>
<td>GIC RED</td>
<td>16 bit</td>
<td>distinct</td>
<td>16 bit</td>
</tr>
<tr>
<td>GIC ITS</td>
<td>48 bit</td>
<td>distinct</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Table 3.1.: Characteristics of interrupt controllers, showing input and output width and the type of configurability.
Chapter 3 - Interrupt Controller Configuration

Listing 3.8: Controllers of the PIC family

**PC/AT** The PC/AT standard of 1984 mandates a cascade of two PICs, and modern systems still offer backward compatibility to this mode. If we are only interested in this mode, we can express a single controller that behaves like a two-chip cascade that simply forwards the interrupts to the CPU with the usual addition of 32.

```c
// A system conforming to PC/AT, like modern x86 systems
// still do, have a cascade of two pics. We can merge
// two instances.
class pcAtPic() {
```

```c
```
3.3.4.2. X86 APIC Family

On modern x86 systems, the APIC subsystem replaces the functionality of the PIC. The main difference to the PIC is the support of multiple processors. The APIC system was introduced with the Intel 486, the first to support a multi-socket SMP configuration. It has been extended in various ways since then, but the basic building blocks remained.

In the APIC system a logical bus is used to direct interrupts in a multi processor system. Each core is connected to the bus using a LAPIC; external interrupts are written to this bus from a variety of clients: The IOAPIC turns wire based interrupts to bus messages. So do MSI controllers, the High Precision Event Timer (HPET) timer, and the LAPICs themselves when triggering IPIs or delivering core local interrupts. Figure 3.5 shows an overview from the Intel software developers manual.

A central concept is the interrupt message that is transmitted on the interrupt bus. An interrupt message can address one, a group or all LAPICs.

**LAPIC** The LAPIC is the hardware unit that is dedicated to each CPU core and handles CPU local state. It tracks, for example, which interrupts are pending. It interrupts the actual CPU core, delivering an interrupt vector. Interrupt vectors on x86 are 8-bit and share a namespace with processor exceptions. The LAPIC listens to interrupt messages on the APIC bus and translates them to CPU vectors. Additionally, the LAPIC also contains an interface to receive a small number of local interrupts, such as interrupts from the core local timer, performance counters, and temperature sensors.
We express this functionality in different controllers: The lapic class turns bus messages into core local vectors. The lapic_lvt (local vector table, in Intel terminology) node represents the core local vectors. The functionality of sending interprocessor interrupts is represented in the lapic_ipi node. Listing 3.10 contains the full definitions.

Interrupt Message Format The interrupt message contains three important fields: The destination mode (one bit, logical or physical destination mode), the destination (8-bit) and the vector (8-bit). Vector is the interrupt vector number that will be delivered to the CPU. The interpretation of the destination field depends on the destination mode and will be explained in Delivery Modes.

IOAPIC We start with expressing the IOAPIC. First we note that there are different variants of the IOAPIC. The term first appeared for a separate IC for delivering interrupts in a SMP setting with Intel 486 (82489DX [Cor]) and then later, with the Intel Pentium became
3.3. Interrupt System Configuration

separate from the LAPIC (82093AA [Int96]). It then got extended twice to the xAPIC and x2APIC. The most important distinction is the higher number of addressable LAPICs. In the P6/Pentium version four bits are used to encode the destination, allowing 15 LAPICs to be addressed, as the all-bits-set destination is used as broadcast. In the extended version, that has been introduced in the Pentium 4 version, the destination field has grown to eight bits. This allows the addressing of 254 LAPICs, as again, the all-bits-set address, 255, is used as broadcast address.

3.3.4.3. Delivery Modes

The delivery modes can be grouped in three categories. Which one is active depends on the interrupt message that is sent, for example from an IOAPIC, and the configuration of the LAPICs. In the LAPIC hardwired APIC ID (ApicId), the reconfigurable logical destination register (Ldr) and the destination format register (Dfr) are used to decide whether an interrupt message is accepted at this specific LAPIC. We get three categories: physical destination mode, in which the local APIC Ldr/Dfr configuration is ignored, logical destination flat mode, and logical destination cluster mode.

**Physical Destination Mode** In this mode, the interrupt message targets a specific or all cores. The physical APIC ID is encoded and the Destination Mode is set to 0 (physical). The interrupt message is accepted by the LAPIC that has the same local APIC ID as the interrupt message. Additionally, all LAPICs accept an interrupt message that has all the bits set. The all-bits-set physical ID is used as the broadcast address.

**Logical Destination Flat Mode** In this mode the delivery mode of the interrupt message is set to logical. The destination contains a bitmask. A LAPIC accepts the message if the logical and of the destination and its logical destination register (LDR) is non-zero. This mode can implement multi-cast.

**Logical Destination Cluster Mode** In this mode, the destination is split into a 4-bit cluster identifier and the lower 4 bits are used to
identify a local APIC in a cluster. This mode can be used only if an additional, not standardized cluster manager is used. This cluster manager connects multiple APIC buses. Since the behavior depends on this cluster manager, we do not specify it.

**Destination Shorthand** The LAPIC can also be used by software to trigger an interrupt on other cores (IPI). This is done by a write to the Interrupt Command Register (ICR) register. The contents of the register encode a APIC bus message. Additionally, the Destination Shorthand (DS) can be used as a shortcut to generate the common use cases: Send an interrupt to all CPUs, send an interrupt to self and send an interrupt to all excluding self. The first two can be directly translated to an interrupt message, as shown in the class `lapic_icr` in Listing 3.10. The all-excluding-self shorthand however does not have directly corresponding interrupt message. There are multiple options of modeling the all-excluding-self behavior, we can include a new field in the interrupt message to specifically exclude one LAPIC from receiving the message or, alternatively, we output the message on another port, and assume this port will be connected to all LAPICs except to the one issuing the Interprocessor Interrupt (IPI) request. The second approach has the benefit of keeping the message in tune with the Intel manual [Int16a] and requires no change in the IOAPIC models. The drawback is, that the connections have to be set up in the correct way. We use the second approach in Listing 3.10.

### 3.3.4.4. Model Definitions

We express the contemporary x86 controllers in the DSL in Listing 3.10.

```plaintext
// IOAPIC as described 82093AA
class ioapicPentium() {
    In : 8;
    Out : struct {DestMode: 1, Dest : 8, Vec : 8};

    Conf = distinct {
        In >= 0, In <= 24,
        Port=0,
        Vec >= 0, Vec <= 255,
    }
}
```
3.3. Interrupt System Configuration

```cpp
Dest >= 0, Dest < 16; //Limited to 4 bits
}

class ioapicP4() {
    In : 8;
    Out : struct {DestMode: 1, Dest : 8, Vec : 8};

    Conf = distinct {
        In >= 0, In <= 24,
        Port=0,
        Vec >= 0, Vec <= 255,
    }
}

class lapicPentium(ApicId : 4){
    In : struct {DestMode: 1, Dest : 8, Vec : 8};
    Out : 8;
    Conf = explicit(Ldr : 8, Dfr : 1) {
        Port = 0,
        Out = Vec,

        // Physical destination mode
        (DestMode = 0, Dest = ApicId) ;
        (DestMode = 0, Dest = 15) ;
        (DestMode = 1, Dfr = 1, Dest & Ldr > 0)
    }
}

class lapicP4(ApicId : 8){
    // Dest is called "MDA" in the Intel manual
    In : struct {DestMode: 1, Dest : 8, Vec : 8};
    Out : 8;
    Conf = explicit(Ldr : 8, Dfr : 1) {
        Port = 0,
        Out = Vec,

        // Physical destination mode
        (DestMode = 0, Dest = ApicId) ;
        (DestMode = 0, Dest = 255) ;
        (DestMode = 1, Dfr = 1, Dest & Ldr > 0)
    }
}
```
Chapter 3 - Interrupt Controller Configuration

```c
};

// The local apic local vector table.
class lapic_lvt() {
  In : 3;
  Out : 8;

  Conf = distinct {
    In >= 0, In <= 7,
    Port = 0,
    Out >= 32, Out <= 255
  };
}

// The interprocessor interrupt source in a LAPIC:
// Translates a register write into an interrupt message. Fixed function.
// Assumes port 0 connected to all CPUs, port 1 connected all but self.
class lapic_icr(ApicId : 8) {
  In : struct {InDS : 2, InDestMode : 1, InDest : 8, InVec : 8};
  Out : struct {DestMode : 1, Dest : 8, Vec : 8};

  Conf = explicit() {
    ( // No Shorthand mode
      InDS = 0,
      InDestMode = DestMode,
      InDest = Dest,
      InVec = Vec,
      Port = 0
    ); ( // Shorthand Self
      InDS = 1,
      InDestMode = 0, // physical
      InDest = ApicId, // own id
      InVec = Vec,
      Port = 0
    ); ( // Shorthand All
```
### 3.3.4.5. Legacy PCI Interrupts

The original PCI bus [PCI04] used four physical wires for signaling interrupts (INTA, INTB, INTC and, INTD). A PCI device can implement up to four functions, each function appears as an independent device to the operating system. Most devices implement only one function; such a single function device must use the INTA line. The interrupt lines are then permuted, depending on the slot where the physical card resides. A further permutation happens when passing PCI bridges. This process is commonly referred to as PCI bus swizzle [Tom99]. In our representation, we assume this permutation has already been computed before connecting it to the next controller.

Ultimately these interrupt lines are either directly connected to an interrupt controller (on x86 the PIC or an IOAPIC) or the root complex implements a PCI link device. The PCI link device is configurable and can forward each interrupt independently, but it is connected only to a subset of IOAPIC inputs. In practice, we found that there are four interrupt lines to which each input can be forwarded. This logic is implemented in the `pciLink` class (see Listing 3.11).

---

2 We are following ACPI terminology here. The same device is called PIR in the ACPI precursor naptable.
Modern machines use the successor of PCI: PCIe. PCIe devices are required to support message signaled interrupts (explained below), but since PCIe is backward compatible with PCI. It also supports a PCIe-to-PCI bridge, thus devices that use legacy interrupts might be present on a PCIe bus, and the mechanism is still relevant today.

To forward legacy interrupts, the interrupt enable bit has to be set per device in the PCI configuration space. We represent this bit using the `pciConf` controller. Unlike the link device, this controller is guaranteed to be present for all PCI devices.

```python
class pciLink(Opt1 : 8, Opt2 : 8, Opt3 : 8, Opt4 : 8) {
    In : 2;
    Out : 8;
    Conf = distinct {
        Out = Opt1;
        Out = Opt2;
        Out = Opt3;
        Out = Opt4
    };
}

// Represents the "interrupts enabled" flag in the PCI conf space
class pciConf() {
    In : 2;
    Out : 2;
    Conf = explicit(Enabled : 1) {
        Enabled = 1, In = Out
    };
}
```

Listing 3.11: PCI interrupt controllers

### 3.3.4.6. Message Signaled Interrupts

MSI is a mechanism that uses memory writes to signal an interrupt. The memory write is captured by the chipset and gets translated to an interrupt message. Intel uses the term Platform Controller Hub (PCH) for chipset, AMD calls it simply `chipset`, on ARM the distributor with
3.3. Interrupt System Configuration

MSI extensions, Interrupt translation service (ITS) or redistributor performs this function.

The Intel software developer manual specifies the exact interpretation of the memory write. It encodes the usual fields we have encountered in the x86 message format, with the addition of a Redirection Hint (RH), a one-bit field turns off any processing of the interrupt message. In practice not setting this bit forces the destination mode to physical. This behavior is encoded in the `msiReceiver` class.

The next step is understanding the generation of MSIs. In principle, any device capable of producing memory accesses can trigger an MSI. We found the HPET the only exception outside of PCI devices that is generating MSI.

PCI defines two different configuration mechanisms, MSI[^1] and MSI-X.

In MSI mode, which predates MSI-X, a PCIe device is allowed to trigger up to 16 interrupts. All of these interrupts trigger a memory write to the same base address. The base address is configurable by system software in the per-device PCI configuration space. A base data word can be set as well, the interrupt number is added to this data word, thus the interrupts generated have always consecutive data words. This controller is defined in `pciMsi`.

While MSI stores its configuration in the space limited PCI configuration space, MSI-X uses an in memory lookup table, that has to be stored within one of the PCI memory mapped regions. It supports up to 2048 interrupts and each interrupt can be configured with an independent address and data word. This gives this controller unconstrained configurability. See the class `pciMsiX`.

There is one more element to enable the routing of PCI generated message signaled interrupts (both MSI and MSI-X): The PCI device must be allowed to bus master, otherwise, the device will never initiate bus transactions and thus never send a message signaled interrupt. We express this bit in the `pciMsiConf` class.

[^1]: The acronym MSI refers to both, the generic mechanism of delivering interrupts via memory write as well as the specific PCI implementation.
class msiReceiver() {
    In : struct {
        // 32 bit address
        Base : 12, InDest : 8, ResA : 8, InRH : 1,
        InDestMode : 1, ResB : 2,
        // 16 bit data word
        ResC : 5, DelMode : 3, InVec : 8
    };

    Out : struct {DestMode: 1, Dest : 8, Vec : 8};

    Conf = explicit() {
        // The redirection hint RH enables the forwarding
        // of the DestMode
        (InRH = 0, DestMode = 0 ;
        InRH = 1, DestMode = InDestMode
        ),
        Base = 0x0FEE,
        DelMode = 0,
        Dest = InDest,
        Vec = InVec,
        Port = 0,
        ResA = 0, ResB = 0, ResC = 0
    };
}

class pciMsi() {
    In : 4;
    Out : 48; // 32 bit address and 16 bit data word

    Conf = explicit(BaseAddr : 32, BaseData : 16) {
        Out = (BaseAddr * 65536) + BaseData + In
    };
}

class pciMsiX() {
    In : 11; // Up to 2048 different interrupts are supported
    Out : 48; // 32 bit address and 16 bit data word

    Conf = distinct {
        In >= 0
    };
}
3.3. Interrupt System Configuration

3.3.4.7. Intel IOMMU

The IOMMU was introduced to safely expose DMA performing devices to virtual machines. (Intel Virtualization Technology for directed I/O [Int16b]). To isolate those devices, a MMU is put in the path between devices and the memory system. A pagetable can then be used to restrict these devices to a subset of the memory space and perform address translation.

A virtualized guest operating system also expects that interrupts are delivered, thus the IOMMU also includes interrupt remapping support. Using this feature a hypervisor can ensure that interrupts from passed through devices can be distinguished and forwarded to the correct guest. Modern versions of the IOMMU hardware can deliver interrupts not only to the hypervisor but also to a virtual APIC memory region. This enables interrupt delivery to a running guest without involving the hypervisor (posted interrupts). A hypervisor might still turn off posted interrupts to get notified when an event can be delivered to a currently suspended VM and consequently raise the priority of the VM or perform any other policy update.

To identify the issuer of the interrupt, a 16-bit identifier must be included in an interrupt arriving at the IOMMU. In the case of message signaled interrupts originating from PCI, the 8-bit bus, 5-bit device, and 3-bit function PCI address serve as the identifier. IOAPICs behind a IOMMU must also provide a unique 16-bit identifier. This identifier

Listing 3.12: PCI MSI controller

```c
// Represents the bus master enabled flag in
// the PCI conf space
class pciMsiConf() {
    In : 48;
    Out : 48;
    Conf = explicit(Enabled : 1) {
        Enabled = 1, In = Out
    };
}
```
must be assigned by firmware and can be found by the operating system using ACPI. The same is mandatory for other devices that create interrupt messages. The Intel hardware can enforce in the Interrupt Remapping Table (IRT) that an interrupt must originate from a specific source, but it cannot use source identifier to route interrupts. Thus we excluded the source identifier in the input in the model. In a concrete implementation, a driver programming the IRT can infer this identifier after the configuration has been determined by the model.

The IOMMU interrupt remapping for message signaled interrupts works as follows: If the 4th bit of the address is set, the interrupt is in remappable format. A MSI gets translated to a table index by considering the handle (encoded in the address using the bits 5-19 and bit 2) and the subhandle (lower 16 bits of the data word). If SHV is set (the third bit of the address), then the resulting index is handle + subhandle. If SHV is not set, then the index is handle. This is expressed in the class `irteIndex`.

If the 4th address bit is not set, the interrupt is in compatibility format. The behavior of those interrupts depend on the IOMMU configuration. If the CFIS flag is set, the compatibility interrupt is forwarded. If not, it is discarded. We assume in our model that the flag is off and discard compatibility interrupts.

For remappable interrupts, the resulting index is used in the IRT, represented in class `IRTE`. The table is unconstrained and contains the usual set of fields of the x86 interrupt message format.

IOAPICs can be guarded by the IOMMUs interrupt remapping. If configured in the normal way, interrupts are generated in the compatibility format. The VT-d specification states how to reconfigure the IOAPIC such that it generates a remappable request, it completely changes the IOAPIC entry format and allows directly addressing an index in the IRT. This simplifies the IOMMU attached IOAPIC model compared to the standard IOAPIC controller since the x86 format encoding is done in the IRT. The model is in class `ioapic_irte`.

1 // Turn a MSI into a index into interrupt remapping
2 // table index
class irteIndex() {
    In : struct {
        // 32 bit address
        Base : 12, HandleH : 15, If : 1, Shv : 1,
        HandleL : 1, ResB : 2,
        // 16 bit data
        Subhandle : 16
    };
    Out : 16;
    Conf = explicit() {
        If = 1, // in remappable format
        (Shv = 1, Out = HandleH * 2 + HandleL +
        Subhandle),
        (Shv = 0, Out = HandleH * 2 + HandleL)
    };
}

// The interrupt remapping table itself is unconstrained
class irte() {
    In : 16;
    Out : struct {DestMode: 1, Dest : 8, Vec : 8};
    Conf = distinct {
        Vec >= 0
    };
}

class ioapic_irte() {
    In : 8;
    Out : 16;
    Conf = distinct {
        In >= 0, In <= 24,
        Port=0
    };
}

3.3.4.8. ARM GIC Architecture

ARM is using the Generic Interrupt Controller (GIC) architecture [ARM16b]. The GIC architecture version 3 consists of the following components (see Figure 3.6):
• The CPU interface
• Redistributor
• Distributor
• ITS

The CPU interface and the redistributor are core local (Processing Element (PE) in ARM terminology) controllers, roughly corresponding to the LAPIC. The distributor receives external wire signaled interrupts (SPIs) and forwards them to the redistributor, thus being similar in function to the IOAPIC. The ITS acts like a combination of Intel MSI receiver and IOMMU, the IOMMU-like remapping functionality is optional. There might be any number of ITS and distributors in a system, unlike on x86, the interrupts have a unique INTID that can not be changed: The interrupt at the distributor with index 1010 will always be delivered to the CPU with index 1010. The interrupt index is read in OS code from the CPU interface. Unlike on x86, where the vector arriving at the CPU is looked up by CPU hardware in the interrupt vector table. This makes supporting a large number of INTIDs possible, as no contiguous in-memory structure has to be allocated and maintained. The specification mandates support for at least 1024 INTIDs. Implementations may support arbitrary large numbers.

The distributor can be set in two modes: Affinity routing or normal mode. In normal mode, the forwarding behavior of an interrupt is configured in the **ITARGETS** register set that contains an array of 8-bit bitmaps to configure to which CPU interfaces an interrupt is delivered. To address more than eight CPU interfaces, affinity routing can be enabled. Then the **IOROUTER** registers are considered, in which a 4 times 8-bit hierarchical address is stored. If the per interrupt Interrupt Routing Bit is set, the destination address is ignored and the interrupt is delivered to all participating PE (ARM calls this the 1-to-N model). For our model, we (over-)assume that all nodes are participating, as it depends on sleep states and other runtime state of
3.3. Interrupt System Configuration

Figure 3.6.: The ARM GIC architecture. [ARM16b]

a. The inclusion of an ITS is optional, and there might be more than one ITS in an IRI.
b. There is one Redistribution per PE.
c. There is one CPU interface per PE.
the PEs we do not capture. We express these nodes in **gic_dist** and **gic_dist_af** if affinity routing is enabled. The corresponding CPU interfaces that decode the interrupt are **cpuif** and **cpuif_af**.

MSIs on ARM (GIC version 3 and newer) are supported using LPIs. Locality-specific Peripheral Interrupts (LPI) extend the INTID namespace to contain at least 8192 LPI identifiers.

The translation of a memory write to a LPI can happen using the following memory locations, it is implementation-defined which of these mechanisms are available.

- one of the redistributors.
- one of the ITSs.
- one of the distributors.

If the MSI targets the GIC redistributor, the data contains the LPI number. Each redistributor is located at a different base address, so the address determines which PE is targeted by the interrupt. The redistributors reference an in memory table containing one byte for each LPI supported by the implementation. The byte encodes enable and priority. The standard allows each redistributor to have its own table, but implementations can require all tables to be the same. To target a MSI to the redistributor, a memory write has to be sent to the redistributor, each redistributor has a different address.

The distributor can expose a similar mechanism, in which case the interrupts are routed like a wire signaled interrupt. The data word contains the Shared Peripheral Interrupts (SPI) number.

Finally, the ITS is the equivalent of the IOMMU, it uses a 16-bit issuer identifier to lookup an Interrupt Translation Table (ITT), in which the input event id (the data word) is looked up. It will be turned into an INTID that optionally targets a virtual destination and an interrupt collection id. That interrupt collection id is looked up in a collection table. The collection table in turn contains the physical address of the redistributor. While this process involves up to six lookups, since each table can be a 2 level radix tree, it
simplifies configuration: any combination of inputs can be mapped to any CPU, as seen in the node \textit{its}.

Unlike the Intel IOMMU, the ITS can use the device identifier to perform routing decisions. This can be used to for example route the same interrupt request from two different devices to different CPUs. On the other hand, the ITS requires all devices to write to the same address to trigger an interrupt, while the Intel implementation offers a range of addresses that can be used in routing decisions.

```cpp
// Exact size of INTID is implementation defined, we use
// 16 bits.
// GIC distributor without affinity routing
// Outputs a bitmap
class gic_cpuif(Id : 8) {
    In : 16;
    Out : 16;
    Conf = explicit() {
        In & Id > 0, Out = In
    };
}

class gic_cpuif_af(AffId : 32) {
    In : struct{Irb:1, Addr:32, Intid : 16};
    Out : 16;
    Conf = explicit() {
        Out = Intid,
        (Irb = 1; Addr = AffId)
    };
}

class gic_dist() {
    In : 10;
    Out : 10;
    Conf = distinct {
        32 <= In, In <= 1019 \textit{must be in SPI range}
    };
}

// GIC distributor with affinity routing
class gic_dist_af() {
    In : 10;
    Out : struct {Irb : 1, Addr : 32, Intid : 16};
```
Conf = distinct {
    32 <= In, In <= 1019, // must be in SPI range
    Intid = In
};

class gic_red()
{
    In : 16;
    Out : 16;
    Conf = distinct {
        In = Out
    };
}

class gic_its()
{
    In : struct { DeviceId : 16, EventId : 32 };
    Out : 16; // Address of the RED
    Conf = distinct {
        DeviceId >= 0
    };
}
3.3.4.9. Useful Properties

Once we have a model that can capture both the topology of a real machine’s interrupt subsystem and the functionality of its programmable interrupt controllers, we can start to formulate useful properties of a given system that can be proved (or disproved) from the model representation.

A first case is **reachability**. It is particularly the case with SoCs that a given interrupt cannot be delivered to any processor in the system. Using the model, we can derive the reachability matrix for interrupts in a given system. Furthermore, since our model also integrates configuration, we can also answer a slightly more challenging question: *Given a set of interrupt source-destination pairs, is it possible to find a configuration that connects all of them?*

For each source-destination pair, there exist multiple controller configurations that connect these two parts and devices can use different signaling mechanisms, which in turn may result in multiple distinct routes through the controller network. As long as controllers can distinguish incoming interrupts, the intermediate representation does not matter (e.g. IOMMU + MSI). Using the model, we can enumerate all possible configurations.

A second useful property of a system is **reliable delivery**: under what conditions is it guaranteed that every interrupt will arrive at the appropriate destination. This may be required to prove the liveness of the system, but even if not (where interrupts are a “hint” to improve the performance of a polling model), failure to deliver interrupts can create hard-to-diagnose performance degradation. Note that for a credible verification we have to address the reconfiguration, that is the property that during configuration updates no interrupts get lost. We did not address this yet.

A less serious but dual problem is spurious interrupts. Our model can be used to constrain the set of possible causes of a spurious interrupt received at a given core, as long as our model of each controller is sufficiently faithful.

As a final example, **distinguishing interrupts** is an important
requirement for an OS so that it can invoke the appropriate device
driver. This should be trivial (each distinct interrupt should arrive on
a different vector on a given core), actually proving that it is the case
is not, and has some similarities with a network capacity problem. A
controller is able to distinguish up to $2^{In \; \text{bit width}}$ different interrupts
where each input identifies an entry in the controller’s routing table. If
there are more interrupt sources than the smallest interrupt controller
can distinguish, interrupt sharing (two distinct sources trigger the
same destination) may occur eventually – when using a suboptimal
configuration heuristic even before all inputs have been allocated. The
model can identify which interrupts are shared and where. Thanks to
the inclusion of configuration options we can pick a minimal sharing
configuration. Currently used heuristics fail to do so on complex
machines.
3.4. Implementation in an OS

We entirely replaced the existing interrupt subsystem of the Barrelfish OS [Bau+09]. In Barrelfish, our work is made easier by the System Knowledge Base (SKB) [Sch+11] – a Prolog engine and constraint solver – which holds the hardware state of the system. We extend the SKB with our model and queries as described in Section 3.3.2.

Barrelfish’s Multikernel architecture pushes device drivers to user space, and therefore allows us to implement our own controller drivers and policies without changing the kernel.

The implementation successfully configures device interrupts on demand on all real and virtual hardware used by the Barrelfish development team, including a variety of x86 and ARMv8-based server machines and ARMv7-A development boards.

While the current implementation is based on the formal model represented in logic programming, it does not provide the assurance of a fully-verified implementation, and the low-level hardware access for discovery and register programming is hand-coded in C and Barrelfish’s (non-verified) Mackerel domain-specific language for hardware registers [Bar13].

Nevertheless, the implementation is functional, demonstrates the viability of the approach, and has greatly simplified and unified device programming across diverse platforms in Barrelfish. Moreover, it is clear that a different (perhaps more complex) implementation is possible in C for monolithic kernel systems like Linux.

Our implementation consists of 343 lines Prolog described in Section 3.3.2. The bulk of the remaining lines consists of platform discovery from ACPI and PCIe (328 lines) and interface functions with system code and debugging utilities (400 lines). The high-level architecture is shown in Figure 3.7 and Figure 3.8 focuses on the model state and depicts which sources of information, processes and Prolog predicates are involved.
3.4.1. The Routing Service

The **Interrupt Routing Service** (IRS) is implemented inside the SKB as a set of inference rules (analogous in this case to stored procedures in a relational database) and executes the routing algorithm incrementally over the SKB’s representation of the interrupt topology and current configuration. The output of the algorithm is a set of (re)configurations for specific interrupt controllers, which are then programmed by their respective driver processes.

Barrelfish implements most drivers in user space, including most of the interrupt controller drivers. The IRS model encodes the routing constraints specific to particular interrupt controller types, but the interface to an interrupt controller used by the IRS can be entirely generic, simplifying implementation.

The model contains all information necessary to route interrupts. A routing request consists of an interrupt source and an interrupt destination. For simplicity, we assume the interrupt destination is provided by the requester. Often, the interrupt destination has some freedom: It is important on which CPU the interrupt ends up, but the exact vector triggered is not important.

The routing algorithm determines a valid configuration for each controller such that all the existing routes plus the new request are satisfied, using the *route* predicate. In Barrelfish, interrupts are always delivered to one core and never broadcast. We add the constraint on
3.4. Implementation in an OS

Figure 3.8.: Overview of the predicates and queries. Red boxes are runtime processes, gray boxes are static files, and blue files are generated.
x86 that the bus message must have physical destination mode set. So far, solving time has had a negligible impact on system performance, even on complex multi-socket platforms, Section 3.4.4 provides an overview.

### 3.4.2. Topology Discovery

Various sources of hardware discovery populate the model for a given machine. Existing drivers for ACPI and PCIe in Barrelfish were simple to modify for this purpose, since they already entered discovered information in the SKB, indeed, in some cases a single Prolog rule provided an appropriate “view” over existing information.

Ideally, PCIe, ACPI, etc. would allow the OS to discover the entire interrupt topology online. Platforms such as ARMv7-A SoCs completely lack a discovery mechanism for devices and interrupts. In other cases, discovery is incomplete for one or more reasons, such as devices that are not on a discoverable bus. For non-discoverable interrupt information, we fall back to static information in compiled Prolog files provided in the startup RAM disk. The OS loads at boot the relevant predicates based on what system and core it is booting on.

Finally, even the information gained from a discovery mechanism is usually insufficient to instantiate the model, even if all the controllers are discovered. The topology itself is often represented only implicitly, such as through the hierarchy of the PCI bus. Often, certain links or translations are missing (for example, in ACPI, it is not discoverable how MSI interrupts are translated to CPU vectors). For this information, we also fall back to knowledge the system programmer has extracted from datasheets (or, conceivably, devicetree files) and coded into the configuration algorithm or a supplemental Prolog file. This information is crucial for any operating system. Our approach explicitly exposes all translation units, while in commodity systems, this knowledge is implicitly contained in program code. Note that the topology and set of controllers in the system can be entirely dynamic.
3.4. Implementation in an OS

The x86 discovery mechanism  To make use of the route predicate presented in Section 3.3.2 we need a populated model. The model components are stored in Eclipse/CLP dynamic facts, which enables us to treat the SKB like a database from which facts can be inserted (asserted) and removed at runtime. However, client code does not directly assert facts that are related to the interrupt model, instead, the client adds underlying facts (for example the contents of an ACPI table, or the devices discovered during a PCI bus scan) and then calls an instantiation Prolog predicate, that queries the facts and asserts the model facts. Thus the clients indirectly asserts those facts. We describe the process on x86, since it has the most complex discovery mechanism. On non-discoverable platforms such as ARMv7 and embedded ARMv8 we hard-code the topology.

On boot, the ACPI service scans the ACPI tables and adds them to the SKB. Once this initial discovery is done, the service invokes the add_x86_controllers, which instantiates CPUs (using the existing SKB information about CPUs and their LAPIC ID). Since we prefer to use APICs, we activate, if available, the APIC mode in the initial ACPI scan and add this to the SKB as x86_interrupt_mode fact. If PIC mode is enabled, we instantiate a PIC controller. If APIC mode is enabled, we test the presence of an IOMMU using facts from the DMAR ACPI table and instantiate the irtMapper and irt, if none is found, we instantiate a msiReceiver. This instantiates the x86 base controllers. After this is done, the second phase of the ACPI scan adds IOAPIC and PCI link controllers.

PCI discovery works similarly; after adding the basic facts about PCI devices (bus/device/function and their vendor and device id), we instantiate one of msi + msiConf, msix + msixConf, or pciConf. When doing this, we take the capabilities of the card and the capabilities of the driver into account; both are stored in the SKB. We also calculate the bus swizzle using the SKB facts about bridges and locations of PCI cards.
3.4.3. Clients

Clients of the Interrupt Routing Service (IRS) are device drivers that wish to receive interrupts from the devices they manage. When a device driver is started by the Barrelfish device manager, the device driver receives capabilities for resources it needs to access the device.

Since we express all interrupt sources in the model, the model provides a suitable namespace for naming resources. In particular it provides names for interrupt sources and interrupt destinations. We implement these as two capability types; both encode a range of interrupt identifiers and we maintain a mapping between these identifiers, and qualified names in the SKB. The source capability is passed to the driver and indicates the right of receiving an interrupt from that source. On x86, the interrupt destination capability can be allocated from an allocator. On ARM, since the INTID can not be changed, the matching destination capability is passed along with the source capability to the driver.

The interrupt capabilities are passed along with capabilities for memory-mapped I/O and communication endpoints to the driver. One of these communication endpoints is the IRS. To start receiving interrupts, the driver passes a pair of interrupt source and destination capabilities to the IRS. On x86, this destination capability can be obtained from an allocator. The interrupt source capabilities can be passed to other drivers, this could, for example, be used to delegate the right of receiving an interrupt to a VM or a kernel bypass network stack.
3.4.4. Evaluation

We perform two experiments to determine the feasibility of using a constraint solver to determine an interrupt route during driver startup. Our first hypothesis is that the overhead of interrupt routing at driver startup is small compared to the current driver startup cost in Barrelfish. In the next experiment, we test how the solver based approach scales with a growing number of devices and routes to be calculated.

3.4.4.1. IRQ Routing in the Context of Driver Startup

We determine the cost of using our solver-based interrupt routing in the process of starting a new driver. We do this by comparing the time of solving the interrupt routing problem with the overall time of starting a driver.

We use Barreflish release 2020-03 on a machine with Intel Ivy-Bridge CPU (Xeon(R) CPU E5-2670 v2 @ 2.50GHz). To determine the duration of a driver startup, we instrument our device manager (Kaluga) and measure the time between receiving a new PCI device discovered event and the time until the e1000 network card driver becomes usable.

Starting the driver requires a number of steps, we include the following to assess the total time of startup: The Barrelfish device manager determines, using the SKB, which driver to use for the device. Kaluga obtains the capabilities needed for this device by retyping Kaluga owned capabilities. Kaluga determines whether to reuse or start a new driver domain. Kaluga starts the driver domain in the latter case. The last step from the driver manager is to start the driver module. The driver module, in turn, maps the device registers and performs a hardware reset of the device. On x86, the driver allocates the interrupt destination capability. It passes the interrupt capability pair to the interrupt routing service. Ultimately, it exports a service to the OS.

About 98% percent of the driver initialization time is spent on the hardware reset of the device. This process consists of triggering a reset procedure and then waiting until the device is ready by checking
if a ready bit is set. Barrelfish implements a busy wait loop, a more efficient implementation might context switch to another task. Thus the comparison of the query time and total time should be done to the initialization excluding the hardware reset. The total time includes using the interrupt routing service and consequently also the query.

To measure the time of the query, we instrument the EclipseCLP solver directly, such that it only includes the cost of executing the query, but no inter-process communication, context switch, or other overhead.

The corresponding times are shown in Table 3.2. The query takes around 720 microseconds, independent of the delivery mechanism. The full startup of the driver takes 3.4 seconds. If we exclude the hardware initialization, this duration is reduced to 49ms.

The numbers look very similar for MSI-X: 740 microseconds for the query, while the hardware initialization takes a bit longer, the cost of the remaining setup is very similar (49ms).

In the context of the Barrelfish driver startup, the cost of using a solver to find interrupts is negligible (0.7ms out of 49ms). We discuss this cost in other contexts briefly in Section 3.5.1.

### 3.4.4.2. Scaling the Number of Devices

As we have established, the overhead in the Barrelfish implementation is small. But running real hardware limits the topology to the installed devices and does not allow us to predict the performance on unknown hardware that might have many more devices or CPUs.
Thus we run a synthetic benchmark with a varying number of devices and show the time needed to construct the constraints and solve them. We chose the topology of x86 with legacy interrupts and assume the PCI devices are directly connected to an IOAPIC. The route will have to traverse the `pciConf`, `ioapic`, and `lapic` controllers. We add `ioapics` as necessary to the system, with 512 devices there will be 22 IOAPICs, each providing 24 inputs. We direct interrupts of two different devices to the same x86 vector number but different CPUs, thus we force the solver to choose the physical destination mode.

The Figure 3.9 shows the solving while increasing the number of PCI devices. The time grows super-linear, the time per device increases. With 512 PCI devices the time to determine all configurations is 56ms.

There are two factors that contribute to the super linearity: While the DFS should stay constant per device, as the topology diameter
stays constant, the topology size might still influence the time needed due to the EclipseCLP representation: Querying for the conn predicate might increase with added number of conn rules. The other factor involved is the solver itself, which has to process more constraints. The performance of the solver is hard to predict, but this experiment illustrates that this particular problem is well suited for a solver.
3.5. Conclusion

We presented a formal model to express the interactions and topologies of interrupt controllers. Our model is capable of capturing the characteristics of a broad range of current systems and we show view equivalence preserving transformations that can be used to convert a complex system model into a flattened representation.

The extension of this model captures interrupt controller configuration. We formulated it, together with descriptions of real hardware and components in a DSL, and have shown that using Prolog and a solver as backend is practical in a real OS.

Our Barrelfish implementation was driven both by the formal model and the particular architecture and facilities of Barrelfish. However, the separation of mechanism from policy (routing) results in, we claim, an elegant solution, and we see no strong reason why the techniques are not equally applicable to monolithic systems such as Linux or microkernels like seL4.

The approach allows high-level language techniques (i.e. inference and constraint solving) to be applied to low-level concerns (interrupts), with a consequent simplification both of individual drivers for peripheral device and interrupt controllers, and also the core of the OS as a whole. A further benefit is that generic interfaces to interrupt controllers and IRS do not end up in contradiction with the behavioral quirks of specific components.
3.5.1. Future Work

In work outside of this dissertation we used the same decoding net model to express not only interrupts but also the memory subsystem, because both resemble network forwarding. In reality they are deeply connected in hardware: message-signaled interrupts and interprocessor interrupts are initiated by writing to memory addresses, and virtualization hardware can translate interrupts into memory writes. Unifying this memory and interrupt network is a natural next step.

Synthesizing a valid configuration is insufficient for correct operation, however. The system requirements are dynamic: devices are hotplugged and brought up and down by power management functions. Threads are migrated between cores by schedulers, etc. This means that the transition between correct configurations, achieved by reprogramming individual controllers, must be achieved without violating security or correctness guarantees, by creating a sequence of correct intermediate states and/or determining which tasks must be paused while reconfiguration occurs. Related work (such as [Fos18, Rei+12]) from the Software Defined Networking (SDN) community might be directly applicable if the queueing behavior of an interrupt controller is compatible with the queueing model of a SDN switch. Determining these fine implementation details of interrupt controllers is hard, as datasheets either miss or deliberately underspecify this behavior to be compatible with future changes.

When considering configuration transitions, the performance of the solver based approach might have to be revisited. For example when scheduling serverless functions (often implemented as VMs) at the tens of milliseconds scale (as proposed for example in [Hen+16]) the interrupt configuration has to be changed slightly on each scheduling decision. The solver could still be used to synthesize an initial configuration, but then only separately verified code would interact with the interrupt subsystem. How to design and verify such a hybrid system is an open question.

Discovering the topology (determining which controllers are present and how they are connected) is non-trivial on modern systems. Doing
3.5. Conclusion

this ad-hoc at runtime is a logical first step, but it prevents us from offline reasoning about all possible systems. This would be especially useful for designing and verifying a controller configuration algorithm that is not relying on a solver.

While we capture the semantics of interrupt controller and their configurability, we do not demand that the solver produces any sort of optimal configuration. By annotating mapping functions with performance characteristics, our model might be used to minimize interrupt latency by choosing an appropriate delivery mechanism [Cor09].

Another direction is to view the per controller configuration as an element of a correct controller specification. This specification would relate the output of the model with a concrete register state. Using this specification, a manually written driver could be verified. This specification could even be used to synthesize a device driver. In particular, by expressing the hardware registers and their meanings in the form of a program sketch [Hu+19] we can use synthesis techniques to generate correct register operations on each device.

We did not address the last step of interrupt delivery: How the operating system kernel interacts with the last level interrupt controller, for example, to acknowledged and mask interrupts. This problem in isolation been tackled [Che+16], but unifying this with our work would decrease the trusted computing base.

A practical extension would be to compile a devicetree file into a static system description to widen our device support. However, we have found that the lack of clear devicetree semantics still requires a manual (human) step in the translation process to a formal specification.
4.

I²C Specification
4.1. Introduction

We present a layered framework for verifying the pervasive I²C systems and provide initial layers of the I²C stack. Each layer has an executable implementation, formal specification and the adherence of the implementation to the specification is model checked.

I²C is a fundamental building block of almost all modern computer systems. It is a low-speed bus used to network most of the integrated circuits and other devices in platforms ranging from server motherboards to mobile phone Systems-on-a-Chip (SoCs), and is also used as a sideband protocol in HDMI and memory DIMMs. While typically invisible to system software, the I²C is used by embedded Baseboard Management Controllers (BMCs) to control power and clock distribution to the rest of the computer system.

For this reason, I²C is a critical (if often overlooked) component. Incorrect programming of the I²C network (for example, misconfiguring a voltage regulator) can cause irrevocable hardware damage. Moreover, I²C Controllers devices, like the BMC, which initiate transactions on the network, have almost unrestricted visibility and authority over the hardware. To build a secure machine, board firmware (such as that running on the BMC) must be trusted. For systems as complex as modern computing platforms, that requires formal verification of the software stack. That, in turn, must be carried out in relation to a faithful model of the underlying hardware.

Unfortunately, I²C is described in an ambiguous informal English-language document [NXP14]. An example ambiguity is that the standard does not specify whether devices are allowed to disappear for brief moments from the bus (such as when they are busy). While almost all significant hardware components in a modern system talk I²C, many interpret this standard differently or only partially implement it.

Thus our stance is that there is no standard. Even if we were to deliver an unambiguous formal specification and a verified implementation - it would be of only limited utility as real world devices would

---

1In this dissertation, we will use the current, more precise terms ‘controller’ for ‘master’ and ‘responder’ for ‘slave’
not adhere to it. Rather provide a toolkit that can be adapted to specific situations with little effort.

Our I²C specification is a first step in addressing this problem. We extracted logical, fine-grained layers from the real world usage and the standard of the I²C protocol. Each of those layers has an abstract specification given as a single Promela process that captures the correct behavior of the complete network (senders and receivers) at that layer. The lowest layer models electrical states on the bus and makes only minimal timing assumptions. This layering is crucial to handle partially compliant devices with little developer effort. By reusing existing layer implementations (Section 4.6.1).

In addition, deterministic, executable implementation specifications at each layer, written in a Domain Specific Language (DSL), describe end-point state machines, which are then compiled into Promela. SPIN [HL91] is then used to verify that the abstract specification at each layer is correctly implemented by the composition of the implementations at all underlying layers. We generate FPGA and C code from the executable specification, which implements a complete stack. We show how to use this code to build a controller stack (once using software only [Section 4.6.3] and once using FPGA hardware and a corresponding driver [Section 4.6.4]) for an I²C EEPROM.

Our specification can therefore serve as the basis for several applications and directions. Hardware designers can employ it as a rigorous, machine-checkable description of how compliant I²C devices must behave and generate high-coverage test suites for their designs. Firmware engineers can use it to generate functional, performant VHDL and C code for parts of their stack and build robust I²C software implementations. Finally, for proof engineers seeking to do full-stack software verification of computer systems, we provide an abstract hardware model that captures the complexity of I²C hardware on which to (partially) base refinement proofs of systems software.
4.2. Background

I²C is a de-facto standard low-speed control bus used for connecting integrated circuits on a PCB and macrocells on an SoC. Board designers appreciate its efficiency since it uses only two shared wires, and allows much of the control sequencing of a computer system to be implemented in software by a BMC. In addition, the same bus can be used to both query sensors and control actuators, allowing for a complex controller to be implemented efficiently. In this section, we describe the basic I²C protocol stack and its implementation subtleties, which have motivated our specification.

4.2.1. I²C Basics

An I²C bus has two wires, clock (SCL) and data (SDA), which are pulled up to the supply voltage. ICs may only drive the lines low, not high. I²C devices are either controllers or responders, and a bus can have multiple controllers and responders. Other devices (e.g. multiplexers), can connect different bus segments. Each responder has a bus-wide unique seven-bit address; some addresses are special and reserved. Communication is always initiated by a controller using the target responder’s address.

![I²C bus symbols](image)

Figure 4.1.: The four I²C bus symbols

The lowest level of the protocol uses the SCL and SDA wires to encode bits (0 or 1) and the start and end of a bus transaction as shown in Figure 4.1. Outside of an I²C transaction, SCL is always high. START/STOP conditions and the clock signal are generated by a controller. Responders signal a 0 bit by driving SDA low, a 1 bit by
doing nothing. When a responder cannot provide the required data in a timely manner, it can perform *clock stretching* by driving SCL low during the clock low period, blocking the bus.

![I2C bus timing diagram](image)

*Figure 4.2.* Example I\(^2\)C transaction with two messages: write one byte, then read two bytes. SDA:C and SDA:R are SDA signals asserted by controller and responder respectively. The address and byte transfers have been abbreviated.

Above the bit layer, I\(^2\)C deals in *transactions* containing one or more *messages*, as shown in *Figure 4.2*. Each message begins with a START condition, and the transaction ends with a STOP condition. After each START, the controller transmits the 7-bit responder address
and a bit indicating READ or WRITE. If a responder with that address is present it acknowledges with a 0 bit (ACK), otherwise the controller sees a 1 bit (NACK). A message will not continue after a NACK. After an ACK, the message payload follows.

If the controller sends a READ, the responder will respond to the controller with a sequence of bytes. Each byte is ACKed by the controller, otherwise the responder stops sending. Likewise, when the controller sends a WRITE, it is followed by zero or more bytes, each of which is ACKed by the responder.

The bus is idle only when both SDA and SCL are high. Collisions (two controllers starting to use the bus at the same time) are detected by a device seeing a 0 bit on the bus when it intended to transmit a 1 bit. In this case, the controllers stop transmitting and may retry the transaction later. An undefined condition [NXP14, p. 12] occurs when START and STOP, START and a bit, or STOP and a bit are generated by different controllers simultaneously.

The specification also defines High-speed mode (Hs) and Ultra Fast-mode (UFm), which fundamentally change the way the bus operates are not covered by this work.

This is the complete, basic $I^2$C protocol and appears fairly straightforward. However, many devices deviate from this standard, making it hard to capture their behavior formally.

### 4.2.2. Non-Compliant Devices

This subsection presents a number of non-compliant devices. We group those devices into three categories: First those that violate the standard required protocol, then those that use ambiguities in the standard that make them incompatible with many other devices. Finally those that have timing related bugs, for example they miss a symbol and wrongly think they still own the bus. These violations happen at different levels of the protocol stack. Our approach allows partial compliant devices to be expressed at a suitable layer.

#### 4.2.2.1. Protocol Violations

120
4.2. Background

AS5011. The I2C standard specifies a transaction header with a data direction bit. According to this data direction bit, either the controller (if the bit is set to write) or the responder (if it is set to read) must write the next data byte.

The AS5011 Hall Sensor[aus09] violates this. The protocol implemented by this sensor uses the R/W flag to indicate read or write, but even when the read flag is set, the device expects a register index to be written first. Figure 4.3 shows a I2C transaction to read from the device. The controller has to write the register index after starting a read transaction. Furthermore, the data direction switches from controller writing to controller reading when transmitting the register data byte. This is a violation of the I2C standard, while still following some of the mandated structure. We show in Section 4.6.1 how we are able to model this device in our framework.

CAT5259 The CAT5259 Digital Potentiometer[ons13] exhibits a similar problem. As the AS5011, it ignores the R/W of a message and requires every transaction to be a write transaction.

These devices violate the behavior we specify in the transaction layer, but are still compatible with the byte layer.

KS0127 The KS0127 video decoder[RG99] ignores the STOP condition unless the controller can include it in a nonstandard position, and continues writing data on the bus. This example device violates the byte layer, but is still compatible with the symbol layer.

4.2.2.2. Ambiguities in the Standard

While the standard specifies the transaction format, it does not state when devices must (and must not) react to specific transactions.
example there is no safe way to probe a I²C bus for devices - even assuming that all bus devices are following the standard. Devices are allowed to perform actions even if only a matching address byte appears on the bus. This has bricked the Thinkpad 240 when probing for available sensors by accidentally causing an EEPROM erase. The standard should specify which transactions must not have side effects.

The standard mandates that devices have a unique 7-bit address but does not require that the address is always acknowledged. This is exploited by the 24xx16 EEPROM, as it does not produce any acknowledge while the device is busy and thus may disappear entirely from the bus for short moments of time.

For a controller driver interacting with such devices it is important to understand the precise behavior of the controller stack: Will it automatically retry a transaction? Does it provide errors that can precisely locate which byte was not acknowledged?

**Wrong abstraction bugs** Another slightly different bug is the exposure of a wrong abstraction of I²C that constrains the type of transactions that can be generated thus rendering the host controller unable to talk with a given device. For example the Xilinx I²C controller is unable to correctly generate a variable length read transaction. In this transaction, the first byte read indicates how many more bytes should be read. But the Xilinx I²C controller peripheral offers an interface that does not allow receipt of these variable length buffers, since it expects a fixed size buffer in advance and will issue exactly as many bus cycles as specified. Device specific workarounds are necessary: For example it may be possible to always read the maximum number of bytes or, if the operation is idempotent, first read the number of bytes and then redo the whole transaction. But devices are not required by the standard to implement any of these guarantees.

**I²C bridging** I²C is used as a method for identifying connected monitors in the DVI protocol. A monitor must embed an EEPROM that indicates its type. However, the protocol evolved into
HDMI, a serial protocol, hence the I²C signal has to be bridged. The I²C under-specification makes this bridging hard. For example the DS90UH947 de-/serializer requires custom mechanisms (such as a mailbox register in the serializer) to support multi-controller, thus, limiting it to software implemented controllers, because the custom external synchronization must be performed before. The datasheet does not state any motivation for this choice, but presumably the many different multi-controller implementations made it difficult to rely solely on detecting START conditions on the bus, such that the manufacturer chose to implement an extra synchronization mechanism.

4.2.2.3. Multi-Controller and Timing Bugs

A well-known, described in the standard, problem of multiple I²C controllers on one bus is that an undefined condition is created when two controllers start transmitting and one transmits a strict prefix of the other. Then the prefix transmitter will terminate the transaction with a STOP and thus an overlapping data bit and a STOP (or repeated START) condition will create an undefined condition on the bus. Therefore the controllers must coordinate in a way that goes beyond what the standard mandates. Possible solutions include partitioning, fixing the transaction length, and length-prefixing. In partitioning, each device is always accessed by the same controller. Then conflicting transactions would safely conflict in the addressing phase. Another solution is to ensure that only transactions of fixed lengths happen, thus synchronizing the generation of START and STOPs. In length-prefixing each variable-length transaction starts with a length byte. If the length of the transactions are the same, there is no conflict as both will produce the STOP symbols at the same time or conflict during data transfer time. If the length is not the same, the conflict will already occur when sending the length byte. Again, this conflict happens safely when data bits are transferred.

The general bad multi-controller support creates a chicken-and-egg problem in the I²C ecosystem: Because the support is bad, nobody uses multi-controller systems, and since there is nobody using multiple controllers, there is no incentive for manufacturers to comply.
Often controller implementations have bugs that appear when using multiple controllers. These bugs are especially dangerous, as they make the device violate the arbitration protocol and thus emit spurious bus signals. Given that \( I^2C \) is often used to control mission critical components, such as voltage converters, a misinterpretation of these signals might lead to a device failure or even a component catching fire.

**STM32 Multi-controller**  The \( I^2C \) controller in the STM32 ARM SoCs seemingly respects transactions initiated by other controllers. But once a remote transaction is finished, the initiated transaction is not properly restarted but only partially from the last repeated-START condition [Dan18]. This is especially dangerous, since spurious bus signals are generated. Our specification rules out this bug, as transactions either run until completion or are completely restarted.

**AVR Multi-controller**  A similar multi controller bug is present in the AVR ATmega series of microcontrollers [Dav]. While controller one is handling an interrupt to deliver a \( I^2C \) STOP message, the hardware Stops listening to the bus, missing that another controller has claimed the bus. If the interrupt service hits the wrong timing, new data to be sent will conflict with messages on the bus.

Our specification does not directly address this problem. We assume ISR and any higher level to execute in zero time.

**BCM2835 clock stretching.**  The hardware \( I^2C \) controller in the BCM2835 SoC (used in the Raspberry Pi 1) ignores clock stretching from responder devices [Adv13] and cannot interoperate with devices that perform any clock stretching. The only workaround is to ignore the hardware and implement the controller directly in software (known as “bit-banging”). This technique is problematic because it is CPU-intensive. Our specification of the symbol level includes arbitrary clock stretching, thus our specification rules out this bug.
4.3. Related Work

I²C has served as a case study for many verification techniques, for example, in applying the Analytical Software Design methodology [Klo+09]. Using a model of an existing controller device, the authors verify correct interaction between a driver and this model. Similarly, Bošnački et al. [BMU09] study the concurrent interactions of a Linux I²C bus driver with hardware and syscalls. While our work overlaps in basic I²C properties, such as adherence to the addressing mode, we specify and construct the controller itself. Finkbeiner et al. [FRS15] study the information flow in an existing unverified I²C controller using HyperLTL, a logic that can reason about and quantify over the set of traces, and thus can correlate inputs and outputs. It is complementary to our work, since our specification could be verified against their information flow properties. I²C is considered in a simulation-assisted verification approach [Gor+06]. The assumptions in this work differ from ours: The system under verification is treated as a black-box and simulation is used to reduce the state-space, while our goal is to replace a black box with a specified, clear system.

I²C was also used as an example in the hardware software co-design approach SHIM [Edw05]. Similar to our approach, they express I²C in layers, which can be implemented in software or hardware. They do so rather coarse grained, as only two different layers are supported. A key difference is the separate semantics for hardware and software components. Thus, each component has to be implemented once in software and once in hardware. SHIM automatically generates interface code between those components. Our approach has a more restricted programming model, but can generate hardware and software from a single language and supports a verification backend.

Bos et al. [BR97] propose to express the I²C controller and devices in discrete time process algebra. Their work neither automatically verifies nor generates code. While they mention the ability to analyze deadlocks, they do not provide any conditions a higher-layer device must fulfill to guarantee deadlock freedom. In contrast, our work’s
main focus is stating sufficient correctness properties for higher abstraction levels. ACCESS.bus is a standard that builds hotplug on top of $I^2C$. Its handshake protocol has been model checked [BG96]. The $I^2C$ abstraction used is on a higher level (messages) than our model (down to level changes on the bus). Our work is complementary and could be used verify their assumptions. Other bus protocols that have been studied using model checking include the CAN bus [Pan+14] and the AMBA on chip bus [RMK03]. These bus specification ensure more guarantees than $I^2C$ such as fault confinement, liveness, priority-based fairness.
4.4. Approach and Tools

We will illustrate our approach with a simplified example consisting of two devices *Controller* and *Responder* and three layers *electrical*, *bus*, and *nibble* layer. The layers are shown in Figure 4.4. The two devices are connected, in this example, using a single wire bus that follows the same logic as in I²C: If all the devices output a one, all will receive a one in the next cycle. The task of the middle layer is to encode and decode a 4-bit message onto the bus.

The *BusController* receives a 4-bit value from the *NibbleController* and yields it bit by bit onto the electrical layer. The *BusResponder* receives bit after bit from the electrical layer and returns the nibble to the *NibbleResponder*. The responder either ACKs (or NACKs) the message by yielding a zero (or one).

An example exchange is given below, between *bus* and *nibble* layers. We denote \( x \) receiving result-value \( y \) from the lower layer with \( x \uparrow y \). This is initiated by the lower layer performing a call. The notation \( x \downarrow y \) is \( x \) sending action-value \( y \) to the lower layer, such an event is triggered by the higher layer performing a yield. In our example \( c \) is the *NibbleController* and \( r \) is the *NibbleResponder*.

\[
\ldots, c \uparrow \text{ACK}, c \downarrow 3, r \uparrow 3, r \downarrow \text{NACK}, c \uparrow \text{NACK}, \ldots
\]

We see \( c \) receiving an ACK (presumably from a past nibble transmission), sending a payload nibble 3, which is received and NACKed by \( r \), ending with the NACK arriving at \( c \).
The correctness statement for this layer is that any datum written by NibbleController will be received by NibbleResponder. We do so by ensuring that the message sequence produced by the implementation and an abstract process are equal.

We implement BusController (Listing 4.1) and BusResponder (Listing 4.2) in our DSL, whose semantics are based on coroutines. Coroutines can call, which invokes the coroutine positioned on the next higher layer in the device specific stack. The callee executes until yielding to the caller. The coroutine resumes at the last yield when called, with the local state preserved.

Each device that participates in the system can choose to implement or reuse a process for each of its layers. The system syntax construct (Listing 4.3) defines the layers and their order. In this example the Bus layer is below the Nibble. If an implementation of the Bus layer calls, it will invoke the corresponding process of the Nibble layer. The DSL will replace the process names with a combination of device name and layer name, but since in this example each process is instantiated exactly once, we refer to the generated processes by their definition name.

We consider the bus logic the ground truth and implement it in a Promela process ElBus2 (Listing 4.4). This process interacts on the passed channels, implementing the equivalent of a call in our DSL, then combines the return values to compute the next bus state (i.e. wired-AND — the bus is 0 if any agent drives it low). In our example, we pass the channels of BusController and BusResponder such that ElBus2 interacts with them. For the actual I²C verification, we implement this process for up to four bus devices and pick the one that corresponds to the number devices instantiated.

We verify the correctness of BusController and BusResponder against BusSpec (Listing 4.5), a nondeterministic process capturing permissible bus behavior, and NibbleValid, which captures allowable event sequences from the next-highest level (i.e. what the bus layer may assume). Figure 4.5 depicts this.

```plaintext
proc (int) BusController(int res) {
```
4.4. Approach and Tools

Listing 4.1: Example Bus Controller

```c
int data;
int data_pos;
int nibble_res;

nibble_res = RES_ACK;
start:
data = call(nibble_res);
data_pos = 0;
while(data_pos < 4) {
    yield ((data >> (3-data_pos)) & 1); //MSB first
    data_pos = data_pos + 1;
}
yield (1); // this reads back the ACK bit
if(res == 0) {
    nibble_res = RES_ACK;
} else {
    nibble_res = RES_NACK;
}
goto start;
```

Listing 4.2: Example Bus Responder

```c
proc (int) BusResponder(int res) {
    int buf;
    int read;
    int ack;

    start:
    buf = 0;
    read = 0;

    while(read < 4){
        yield (1);
        assert(res == 0 or res == 1);
        buf = (buf << 1) | res;
        read = read + 1;
    }
    (ack) = call(buf);
    yield (ack);
goto start;
}
```
Chapter 4 - I²C Specification

```c
system {
    layers [Bus, Nibble];

device controller {
    Nibble : NibbleController<15>;
    Bus : BusController;
};

device responder {
    Nibble : NibbleResponder;
    Bus : BusResponder;
};
}
```

Listing 4.3: Example System definition

```c
proctype ElBus2(chan p1_in; chan p1_out; chan p2_in;
    chan p2_out) {
    int scl, scl1, scl2;
    scl = 1;
    start:
        p1_in!scl;
        p1_out?scl1;
        p2_in!scl;
        p2_out?scl2;
        if 
            :: scl1 == 0 || scl2 == 0 -> scl = 0;
            :: else -> scl = 1;
        fi
        goto start;
}
```

Listing 4.4: Example ElBus2 (ground truth) implementation

**BusSpec** prescribes how actions from the nibble layer translate into events to the nibble layer e.g. the number 3 in the above example. **NibbleValid** includes all possible actions at the Nibble layer. It non-deterministically transmits a 4-bit nibble, which is either ACKed or NACKed (again non-deterministically). Correct delivery is guaranteed by **BusSpec**.
4.4. Approach and Tools

![Verification processes for verifying the Bus level. Gray processes are generated from the DSL; blue ones are expressed in Promela](image)

The verifier is an additional process (Listing 4.6) that polls the message channels and forwards messages to both **BusSpec** (written in Promela [Listing 4.5]) and the bus implementation. The bus implementation is generated by the DSL, **El_conc_run** and **Bus_conc_run** are shorthands that start processes for the corresponding layer for all devices. In our example **El_conc_run** runs **ElBus2** and **Bus_conc_run** runs **BusController** and **BusResponder**. If both produce the same result, execution continues, otherwise the verifier stalls (no transition/deadlock). Implementation correctness is then checked by using SPIN to verify the absence of deadlock in the combined process. Furthermore, we check that the actual implementation must make progress. SPIN supports detection of non-progress cycles by using annotating states as progress states. In our verification, we use only one progress state: When messages are exchanged between the verified layer. We then run SPIN again, checking for non-progress cycles. This gives us the guarantee that the implementation will always make progress and that the exchanged messages are the same as in the specification.

```proctype NibbleValid(chan ci, co, ri, ro) {```
```
int c_res = RES_ACK;
int dat;

start:
    select(dat : 0..15);
    ci?_; co!dat;
    ri?_;
    if :: ro!ACT_ACK;
        c_res = RES_ACK;
    :: ro!ACT_NACK;
        c_res = RES_NACK;
    fi
    goto start;

proctype BusSpec(chan ci, co, ri, ro){
    int dat;
    int res = RES_ACK;

    start:
        co!res; ci?dat;
        ro!dat;
        if :: ri?ACT_ACK;
            res = RES_ACK;
        :: ri?ACT_NACK;
            res = RES_NACK;
        fi
        goto start;
}

Listing 4.5: Example top level processes

/*@ the verifier process */
init {
    /* the abstract channels */
    chan abs_c_out = [0] of {int};
    chan abs_c_in = [0] of {int};
    chan abs_r_out = [0] of {int};
    chan abs_r_in = [0] of {int};
```
4.4. Approach and Tools

```
/* upstream channels */
chan nibble_c_in = [0] of {int};
chan nibble_c_out = [0] of {int};
chan nibble_r_in = [0] of {int};
chan nibble_r_out = [0] of {int};

#define conc_c_out controller_Nibble_in
#define conc_c_in controller_Nibble_out
#define conc_r_out responder_Nibble_in
#define conc_r_in responder_Nibble_out

/* The concrete machines */
run El_conc_run();
run Bus_conc_run();

/* Abstract and valid */
run NibbleValid(nibble_c_in, nibble_c_out, nibble_r_in, nibble_r_out);
run BusSpec(abs_c_in, abs_c_out, abs_r_in, abs_r_out);

/* Verify that abs and conc equal */
int x;
int x1;
progress:
do
  :: abs_c_out?x; conc_c_out?eval(x); nibble_c_in!x;
  :: abs_r_out?x; conc_r_out?eval(x); nibble_r_in!x;
  :: nibble_c_out?x; conc_c_in!x; abs_c_in!x;
  :: nibble_r_out?x; conc_r_in!x; abs_r_in!x;
  od
```

Listing 4.6: Example verifier

### 4.4.1. Programming Model, DSL, and Backends

As discussed, our DSL is based on coroutines. The language is (semantically) an executable subset of Promela with messages restricted to the `call` and `yield` primitives and an acyclic call graph. These restrictions allow for the generation of compact C code. Unlike ex-
isting C-to-Promela converters [JJ09], we describe stateful processes (coroutines). Implicit state makes it convenient to express stateful protocols such as I^2^C.

Processes have state variables of type int or intarr (fixed-size array). No global variables are allowed. The size of intarr is implementation-defined, but guarded against overflow. The DSL supports while, if and goto as control flow.

The DSL compiler is implemented in Haskell and uses the megaparsec library to parse the input. The compiler then checks that all used variables, labels, and processes are defined. If successful, the compiler instantiates the devices. It assigns each process a unique name, replaces call with the corresponding name, and passes the list of instantiated and checked processes to one of the backends.
4.4. Approach and Tools

4.4.1. C Backend

The C backend translates each process to a function and each process call into a function call. The backend assembles all state in a large static \texttt{struct}, kept intact between calls. In addition to the state variables, we add an extra instruction pointer variable, which keeps track of the execution point. At the beginning of each function, a large switch statement checks this state variable, jumping to the corresponding location. To \texttt{yield}, the next instruction location is stored in this instruction pointer state variable before executing a return. This ensures that on next invocation the process continues after the \texttt{yield}.

4.4.1.2. Promela Backend

From the language subset, Promela generation is straightforward. Each instantiated process has two unbuffered, rendezvous-type channels: \texttt{input} and \texttt{output}. A \texttt{call} sends arguments to the callee’s input channel and blocks on the callee’s output channel. Processes block on input until arguments arrive. \texttt{Yield} sends the result on the output channel and blocks on the input.

Verification properties are specified directly in Promela, exploiting nondeterminism. The complete syntax of our DSL is expressed in [Listing 4.7](#).

\[
\begin{align*}
\langle \text{file} \rangle & \ ::= \ (\langle \text{proc} \rangle \mid \langle \text{system} \rangle)^* \, \text{EOF} \\
\langle \text{proc} \rangle & \ ::= \ 'proc' \ '(', \ (\langle \text{type} \rangle \ (',' \langle \text{type} \rangle)^* )? \ ')? \ ');' \ 'id' \ '(', \ (\langle \text{varDecl} \rangle \ (',' \langle \text{varDecl} \rangle)^* )? \ '} \ ');' \\
\langle \text{block} \rangle & \ ::= \ '{' \langle \text{instr} \rangle \ '} ' \\
\langle \text{instr} \rangle & \ ::= \ 'yield' \ '(' \langle aEx \rangle \ '),' \ 'id' \ ')' \ ';' \\
& \ | \ 'varRef' \ '=' \ 'aEx' \ ';' \\
& \ | \ 'id' \ '::' \\
& \ | \ 'while' \ '(' \ 'bEx' \ ')' \ 'block' \\
& \ | \ 'if' \ '(' \ 'bEx' \ ')' \ 'block' \ ('else' \ 'block')? \\
& \ | \ 'goto' \ 'id' \ ';'
\end{align*}
\]
Chapter 4 - I²C Specification

| 'assert' '(' (bEx ')') ';' |
| (⟨id⟩ | '(' C ⟨id⟩ (',' ⟨id⟩)+ ')')) '=' 'call' '(' |
| (⟨aEx⟩ (',' ⟨aEx⟩)*)? ')' |

⟨varDecl⟩ ::= ⟨type⟩ ⟨id⟩
⟨type⟩ ::= 'intarr' | 'int'
⟨cEx⟩ ::= ⟨aEx⟩ ('>' | '>' | '<=' | '<' | '==' | '!=') ⟨aEx⟩
⟨bEx⟩ ::= 'true' | 'false' | '(' (bEx ')') |
| ⟨bEx⟩ (′and′ | ′or′) ⟨bEx⟩ | ⟨cEx⟩
⟨aLit⟩ ::= ('-'?[0-9]+)
⟨aEx⟩ ::= '(' (aEx ')' | ⟨varRef⟩ | ⟨aLit⟩ | ⟨uOp⟩ aEx⟩ |
| ⟨aEx⟩ bOp aEx⟩
⟨bOp⟩ ::= '&&' | '|' | '+' | '/' | '+' | '-' | '=' | 'π' | 'π'
⟨uOp⟩ ::= '- | '+'
⟨varRef⟩ ::= ⟨id⟩('[' aEx ']' |
| ⟨id⟩( ' [' aEx ']' |
| ⟨id⟩[])* |
| ⟨type⟩⟩
⟨id⟩ ::= ⟨char⟩ ⟨⟨char⟩ | [0-9]+ | 
| ['-']* |
| ⟨char⟩ ::= ['a'-z'] | ['A'-Z']
⟨system⟩ ::= 'system' '{' 'layers' ['[ ' (id) (',' (id))]* ']' '};' |
| ⟨device⟩* '{'}
⟨device⟩ ::= 'device' '{' ( ⟨id⟩ ':' ⟨id⟩ ⟨implParam⟩? ' ;')* |
| '}' '
| ⟨implParam⟩ ::= '<' ⟨aLit⟩ (',' ⟨aLit⟩)* '>'

Listing 4.7: Complete DSL syntax
4.4. Approach and Tools

4.4.2. Calculus

We verify our layered system with the following calculus: Each layer has an implementation $LayerImpl_i$, a valid behavior $LayerValid_i$, and a specified behavior $LayerSpec_i$.

$LayerSpec_i$ is a state machine expressed as a function over its past result/action trace returning the next result symbol. $LayerSpec_i$ specifies the correct behavior at layer $i$.

$LayerValid_i$ is a unary predicate over the result/action trace that is true if the sequence is permissible at this layer and ends with an action symbol.

$LayerImpl_i$ is a binary predicate over two result/action traces, the first argument is the result/action sequence permissible at layer $i - 1$, while the second is the result/action sequence permissible at layer $i$. We denote the binding of two such predicates with the combinator $\circ$.

Figure 4.7.: Verification illustration, the system is verified if $\omega_i = \omega'_i$
Definition 3. ◦, binary predicate binding.

\[(A \circ B)(\omega, \psi) := \forall \tau. A(\tau, \psi) \land B(\omega, \tau)\]

We also define binding ◇ to an unary predicate \(B'\) (such as LayerSpec), which results in a unary predicate.

Definition 4. ◇, unary-binary predicate binding.

\[(A \diamond B)(\psi) := \forall \tau. A(\tau, \psi) \land B(\psi)\]

LayerImpl\(_0\) is, unlike the other LayerImpl, a unary predicate. It defines the ground truth and does not have any downstream interactions.

As explained before, isolated specification of a device layer is not possible, hence LayerImpl as well as LayerSpec are defined on traces that include actions and results for all devices in the system.

In this calculus, our system is verified if it fulfills

Theorem 1. Implementation-Specification adherence.

\[\forall i. \text{LayerValid}_{i+1} \Rightarrow \text{LayerSpec}_i = \text{LayerImpl}_i \circ \text{LayerImpl}_{i-1} \circ \ldots \circ \text{LayerImpl}_0\]

This verification procedure is depicted in Figure 4.7 as an infinite sequence of directed messages \(\omega = e_1, e_2, \ldots\). We denote the sequence of all messages as \(\omega\), and the sequence of messages exchanged between layer \(i\) and layer \(i + 1\) as \(\omega_i\). Examples of messages are the action to send an acknowledgement (represented by a down arrow \(\downarrow\)) or the receipt of an acknowledgment message (depicted as an up arrow \(\uparrow\)). Even though abstractly we deal with a simple trace of events, it is useful to denote if the message is destined for the controller or responder. We denote this with \(c\uparrow x\) for a result with value \(x\) destined for the controller, and with \(r\uparrow x\) a result \(x\) destined for the responder.

A LayerImpl\(_i\) can not only be bound to other implementations, but also to a LayerSpec\(_i\). We evaluate the verification time improvements of this in Section 4.6.
4.4. Approach and Tools

We verify this property by encoding it into Promela processes, such that a **verifier** process can not make progress when a violation has been found. Adherence to the protocol is shown by **verifier** always able to make progress. $LayerValid_i$ becomes a non-deterministic process, producing all valid actions and accepting all valid results. The actions are sent to the **verifier** which duplicates the actions and sends them to both the $LayerSpec$ and the $LayerImpl$. If the $LayerSpec$ produces a result, the verifier ensures that the $LayerImpl$ produces the same result. If it differs, the verifier will not make progress. We also use this message dispatch to show liveness of the system, by marking it with a SPIN progress label and verifying the liveness check.

We currently do not verify that the layer implementation $LayerImpl_i$ adheres to $LayerValid_i$, which can be expressed in Theorem 2.

**Theorem 2.** Implementation-Valid adherence.

$$\forall i, \forall \omega_i, \forall \omega_{i-1}
LayerValid_{i+1}\omega_i \Rightarrow
LayerImpl_i\omega_{i-1}\omega_i \Rightarrow
LayerValid_i\omega_{i-1}$$

Implementing this in SPIN would require us to specify each $LayerValid_i$ twice, once as non-deterministically generating process and once as deterministic process accepting all valid traces.

Since we verify the full stack of implementations (as stated in Theorem 1), we still do get the correctness guarantees, but it is possible, that in the middle of the implementation stack, the implementations rely on behaviour not verified by the corresponding $LayerValid$. 
### 4.5. The I²C Model

#### 4.5.1. Layering of I²C

We divide our I²C stack into the layers shown in Figure 4.8, and we apply the verification principle from Section 4.4 at every layer.

The stack presented here includes two device-specific layers: World and Driver layer. We envision this process of device modelling and verification to be done for each device that is connected to the bus. This also gives a high level of assurance for the device driver represented by Driver. But it is also feasible to directly interact in a system with, for example, the transaction layer and skip the verification of the higher layers.

#### 4.5.2. Layer 0: Electrical Layer

The lowest layer, the electrical layer 0, is trusted. Hence it consists of only an implementation. It describes how two devices sending bus signals (a SCL/SDA pair, each 0 or 1) are combined into the next bus state, which is sent back to the devices. It does so by using the

![Layering of the I²C model](image)

**Figure 4.8.:** Layering of the I²C model.
4.5. The $I^2$C Model

$I^2$C mandated AND combination of signals for each wire, which is a result of the active drive low logic.

We assume a reliable delivery of bits. Like prior work [BR97], we observe that $I^2$C bus events can be discretized. We assume sampling of the bus at the Nyquist frequency of the clock, such that two samples occur during a clock high period. This allows us to distinguish START and STOP conditions from BIT0 and BIT1.

4.5.3. Layer 1: Symbol Layer

Layer interface The symbol layer connects the electrical with the byte layer. It parses a sequence of bits into a symbol and vice-versa, turns a symbol into a bit sequence. The results and actions are depicted in Figure 4.9. In addition to the $I^2$C symbols we define IDLE and STRETCH, which delay the next symbol.

The implementation differs for controller and responder, but they share a large part of the code (expressed as two sub-processes *SymbolReader* and *SdaDriver*). The controller is actively driving the clock (using a sub-process *SclDriver*). The responder is driving the clock only in one specific case: When it is processing a STRETCH action, it will delay the clock rise by one cycle. Longer clock stretching can be expressed by repeatedly issuing a STRETCH action. Both controller and responder are clock agnostic. For example the *SdaDriver* will wait until the clock rises and falls again, thus it is invariant to clock...
strecthing. Both byte controller and byte responder are invoked in the same clock cycle. The exception again is during a STRETCH, which will produce an extra invocation in the next clock cycle.

The specification defines how two symbol actions are combined into a new one. In the initial, out-of-transaction state, two IDLE commands are combined into an IDLE result (i.e. $c \downarrow \text{IDLE}, \ldots, r \downarrow \text{IDLE}$ will be followed by $c \uparrow \text{IDLE}, \ldots, r \uparrow \text{IDLE}$) as well as a START and a IDLE are combined into START (i.e. $c \downarrow \text{IDLE}, \ldots, r \downarrow \text{START}$ will be followed by $c \uparrow \text{START}, \ldots, r \uparrow \text{START}$).

If a START result has been sent, the specification enters the in-transaction state. In this state, the following action combinations are valid. Note they are symmetrical, thus we skip the sender identifier as well as symmetrical cases.

- $\downarrow \text{BIT1}, \downarrow \text{BIT1}$ produces two $\uparrow \text{BIT1}$,
- $\downarrow \text{BIT0}, \downarrow \text{BIT1}$ produces two $\uparrow \text{BIT0}$,
- $\downarrow \text{BIT1}, \downarrow \text{START}$ produces two $\uparrow \text{START}$,
- $\downarrow \text{BIT1}, \downarrow \text{STOP}$ produces two $\uparrow \text{STOP}$, and enter out-of-transaction state.
- $r \downarrow \text{STRETCH}$ is immediately followed by $r \uparrow \text{STRETCH}$, until $r$ produces a non stretch action.

The valid actions of the next higher layer follow the same in- and out-transaction states as the specification. Outside a transaction, Byte-Valid either generates an IDLE pair to remain outside a transaction or initiates a transaction by allowing the controller to generate a START. In-transaction it generates a zero or more sequence of STRETCH, followed by all the valid symbol combinations.

### 4.5.4. Layer 2: Byte Layer

Layer Interface $\text{I}^2\text{C}$ is a byte-oriented protocol, where each byte is acknowledged. This layer reads and writes symbols, turning them
4.5. The I²C Model

Results:
IDLE, START, STOP, FAIL
ACK, NACK, RES_READ, x

Actions:
IDLE, START, STOP,
WRITE, x, READ, ACK, NACK

L2: Byte
Controller Responder

Figure 4.10.: Interface of the Byte layer. Both controller and responder have the same signature for actions and results.

into bytes. START and STOP conditions are passed through. The higher layer must send START and STOP explicitly. The interface is depicted in Figure 4.10.

The implementation does not distinguish between controller and responder. START and STOP actions are passed to the symbol layer directly. WRITE, x and READ operate bitwise (MSB first transmitted). If a written bit is not correctly transmitted, the layer will report FAIL and remain silent for the rest of the byte.

The specification describes how actions are combined into results. Controller and responder are not equal anymore; the controller must initiate the transaction. As in the symbol layer specification, an IDLE pair remains outside transaction and a START/IDLE is used to enter the in-transaction state. Within a transaction the following combinations hold

- $c \downarrow \text{ACT\_WRITE, } x$ as well as $r \downarrow \text{ACT\_READ}$ are followed by $r \uparrow \text{RES\_READ, } x$, $r \downarrow \text{ACT\_}(N)\text{ACK}$, and $c \uparrow \text{RES\_}(N)\text{ACK}$. Note that the variable $x$ is bound, the written value must be the same as the read value.

- The symmetrical case of the above item where the controller reads and the responder writes.
• ACT_READ can also be combined with ACT_START and ACT_STOP. This is important for the responder, who can not predict the action of the controller, then ACT_READ is a safe choice.

The valid actions follow directly from the specification. All specified combinations are verified, with the caveat that the value of the written byte is constrained to a predefined set of 10 choices. In Section 4.6 we show the trade-offs to verify all values (0 to 255).

4.5.5. Layer 3: Transaction Layer

I^2C defines the transaction format, such that a START condition must be followed by a 7-bit address and a direction bit. Then, depending on the direction bit, the controller reads or writes a sequence of bytes. This introduces an asymmetry: It is the controller that initiates a transaction, and the responder acts accordingly. Figure 4.11 shows all the actions and events processed at this layer. Starting from this layer, controller and responder have not only distinct specifications as before but also distinct implementations. The responder also decodes a I^2C address and ignores, using NACKs, all other addresses.

The implementation of the controller is fairly straightforward: Each higher level action is turned into a sequence of START and address byte with correct direction bit. If a write is requested, it continues to write the data. If a read is requested, it reads the desired number of bytes, sending an ACK for all except the last byte (per I^2C standard) which is answered with NACK that tells the responder to stop sending. If the responder receives a NACK, it is forwarded to the next higher layer.

The responder waits for a START, which is propagated to the driver. We propagate STARTs to the next higher layer to distinguish between two consecutive writes and a write – restart – write sequence. After a START, the responder reads the address byte, checks that the addresses match, and depending on the direction bit, propagates either a RES_READ or a RES_WRITE. Since the responder cannot
know how many bytes are read, we propagate each byte read request individually to the next level. Writes on the other hand can be buffered, until the controller is done. Then the whole array of written data is passed on.

The specification describes the interaction of driver layer actions. The sequence is determined solely by the controller: If it requests a WRITE of $x$ bytes, we expect a RES_START from the responder, followed by $x$ times a RES_READ. The responder either ACKs $x-1$ times and then the controller will receive a RES_ACK, or if the responder decides to NACK before, the controller will receive a NACK.

The valid action sequences become conceptually simple but increasingly challenging to verify. The controller produces after a sequence of ACT_IDLE, any combination of ACT_READ and ACT_WRITE until finally an ACT_STOP brings it back to the initial, out-of-transaction state. However the data that is either read or written is potentially infinite in length. Since we focused on the correct delivery of data at the lower layer, the verification cases for this layer focus on increasing the length of the transaction. Hence we show that for sequentially increas-
ing payload of any length between 1 and 4 bytes our implementation conforms to the specification.

Conversely, the responder is completely driven by the result it receives. After a RES_START only ACT_IDLE is valid. After receiving START but before a STOP, the following combinations are valid: RES_READ followed by ACT_WRITE, x, RES_WRITE followed by ACT_ACK or ACT_NACK, RES_STOP, and RES_START must be followed by ACT_IDLE.

4.5.6. Layer 4: Driver

Layer interface At this layer we start implementing the protocol that is specific to our model EEPROM, a Microchip 24XX16 [Mic19]. The controller contains what typically is implemented in the device driver. From the world layer, the controller receives requests for reading or writing from the EEPROM. The responder, on the other hand, encodes the EEPROM-specific features, for example the logic for the address buffer. The responder forwards requests to an EEPROM implementation. The interface is shown in Figure 4.12.
The controllers implementation turns an ACT_WRITE_EEPROM, which is parametrized by an offset and a data array, into a long write transaction. The first two bytes determine the EEPROM write offset followed by the data to be written. The data length is not communicated explicitly; if the data is written, the controller sends a STOP condition, signaling to the responder that the transaction is over. Reading works similarly: The controller issues two-byte write transaction followed by a len-long read transaction.

The responder behaves similarly. It waits for a START, then expects a write of at least two bytes. If more bytes follow, they are interpreted as a write transaction. It assembles the written data into a buffer and, once the STOP condition arrives, passes it on (as RES_WRITE,xs) to the world layer. Read is more difficult, because by the time the first read byte must be supplied, the read length is unknown. Hence we assume there is a maximum read length, which we query from World (by issuing a RES_READ) then sending from this buffer.

The specification becomes fairly simple at this point. A controller ACT_WRITE,off,xs is turned into a responder RES_WRITE,off,xs. A controller ACT_READ,off,len becomes a RES_READ,off,maxlen, followed by a responder ACT_OK,xs, and a controller RES_OK,xs’, where xs’ is a prefix of xs with length len.

The valid action sequence is unconstrained at this point. The controller can choose between ACT_WRITE and ACT_WRITE, while the responder must subsequently receive RES_READ and RES_WRITE and reply with ACT_OK. However, we severely restrict the payload that is transmitted at this level by choosing one of 4 predefined datasets, to keep the full implementation verification time manageable.

4.5.7. Layer 5: World

Since this is the highest layer, we can not verify the behavior given a higher layer behavior. We still provide a dummy implementation that performs a defined sequence of actions which is what we evaluate on our hardware platform.
4.6. Evaluation

4.6.1. Expressing Non-Compliant Devices

A goal of our approach is to interact and verify interactions with partially compliant devices. We evaluate the effort needed to express and verify a driver and device model of the AS5011 device. As described in Section 4.2.2.1, the device requires a direction bit that is incompatible with the direction of the subsequent data transfer. We compare the effort and flexibility of our approach to a device driver written using the Linux kernel API. Note that writing the device driver is only half of the task of the verification, the developer also has to come up with a device model, which is not part of Linux.

The state of the art way of writing a driver for an I²C-attached device is using an I²C API that comes with a list of pre-defined flags to activate non-standard behaviour. The Linux I²C API (which can be accessed using `ioctl` from userspace or using the functions with `i2c` prefix) defines 9 such flags. There is no guarantee that a controller driver will handle any of these flags. The Listing 4.8 shows the function to read a register in the AS5011.

```c
#define I2C_M_NOSTART 0x04
#define I2C_M_RD 0x01
#define I2C_M_REV_DIR_ADDR 0x20

static int as5011_i2c_read(struct i2c_client *client, uint8_t aregaddr, signed char *value) {
    uint8_t data[2] = { aregaddr);
    struct i2c_msg msg_set[2] = {
        { .addr = client->addr,
          .flags = I2C_M_REV_DIR_ADDR,
          .len = 1,
          .buf = (uint8_t *)data }
    },
    { .addr = client->addr,
      .flags = I2C_M_RD | I2C_M_NOSTART,
      .len = 1,
      .buf = (uint8_t *)data }
    };
    int error;
```
error = i2c_transfer(client->adapter, msg_set, 2);
if (error < 0)
    return error;

*value = data[0] & 0x80 ? -1 * (1 + ~data[0]) : data [0];
return 0;
}

Listing 4.8: Linux read of a AS5011 register

In contrast, our approach requires first determining up to which layer the device is compliant with the provided default I2C stack. This device is compliant up to and including the byte level, but violates the transaction layer, as the transaction layer inserts direction bits in a standard conforming way. Thus we need to write a new transaction layer implementation and re-use the standard byte, symbol and electrical layers. The implementation of a register reading transaction layer (that performs the same task as Listing 4.8) is shown in Listing 4.9.

proc (int, int) TrAsController<int ctrl_addr>(int res,
    int res_data) {
    int act; int reg_idx; int reg_data; int hls_res;
    hls_res = RES_OK;

    start:
        (act, reg_idx, reg_data) = call(hls_res);

        yield (BYTE_ACT_START, 0);
        if(res != BYTE_RES_START) { goto err; }
        // I2C Address, read flag set
        yield (BYTE_ACT_WRITE, (ctrl_addr << 1) | 1);
        if(res != BYTE_RES_ACK){ goto err; }
        // Register address
        yield (BYTE_ACT_WRITE, reg_idx);
        if(res != BYTE_RES_ACK){ goto err; }
        // Register content
        yield (BYTE_ACT_READ, 0);
        if(res != BYTE_RES_READ){ goto err; }
        hls_res = res_data;
        yield (BYTE_ACT_NACK, 0);
The length and complexity of the code is comparable, but our approach has a number of benefits: It was designed with verification in mind and the same language can be used to describe not only a controller, but any bus device. In our approach it is also easier to insert error reporting: Suppose the device would not acknowledge if a wrong register address has been written. In our DSL it would be trivial to produce a custom error code in precisely this situation (by changing line 13 in Listing 4.9). Our approach is also more general: It can not only produce any byte sequence (instead of those allowed by the flags) but it can also create non-standard bytes by interacting with another layer.

To verify the interaction also, the responder side has to be expressed (Listing 4.10) as well as a system assembly (Listing 4.11) and the layer specification.
4.6. Evaluation

Listing 4.10: Responder declaration of the AS5011
4.6.2. Verification Runtime

The verification runtimes are evaluated on an AMD Ryzen 9 3900X with 32 GB of RAM running Ubuntu 20.04 with SPIN version 6.4.9.

Verification of the Symbol layer performs a complete state space search and finishes in 0.4 seconds. As mentioned in Section 4.5.4, the byte layer is verified only on a small set of payload values, hence we would like to speed up the verification time. We can do so by replacing lower \texttt{LayerImpl} with \texttt{LayerSpec}, for example showing \texttt{LayerSpec\_byte} = \texttt{LayerImpl\_byte} \circ \texttt{LayerSpec\_sym} instead of \texttt{LayerSpec\_byte} = \texttt{LayerImpl\_byte} \circ \ldots \circ \texttt{LayerImpl\_0}.

Table 4.1 shows the verification times using this method. The speedup factor depends on the layer complexity. Replacing the rather complex \texttt{Symbol} with \texttt{SymbolSpec} leads to a speedup of 10, replacing \texttt{Byte} with \texttt{ByteSpec} leads to 5 speedup.

Instead of decreasing verification time, this technique can be used to increase the search space size. For example the \texttt{Byte} layer can be completely verified (i.e. checking all 256 values for a byte write as well as for a byte read) using \texttt{SymbolSpec} in about 70 seconds.
4.6. Evaluation

Table 4.1.: Verification time in seconds using different layers of abstraction.

<table>
<thead>
<tr>
<th></th>
<th>Full Implementation</th>
<th>SymbolSpec</th>
<th>ByteSpec</th>
<th>TransactionSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>9.0</td>
<td>0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction</td>
<td>69.4</td>
<td>8.7</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>HI</td>
<td>62.9</td>
<td>6.7</td>
<td>1.0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

4.6.3. Execution on a Raspberry Pi

Our DSL can generate C code for deterministic state machines. Conceptually, the C code can interact with any layer directly. For example, it could get output from the transaction layer and translate it into Linux I²C API calls. However to profit most from the verification, the whole stack (excluding the electrical layer) can be executed. This leads to an interface that only writes and reads SDA/SCL as high/low states from the bus. We demonstrate this using the Linux GPIO interface connected to an I²C EEPROM. The boilerplate code runs in an infinite loop: reading the bus state, forwarding it to the controller state machine, reading back the command, and outputting it on the GPIOs. In conformance with the I²C specification we actively drive the data and clock line low on zero. If a one is to be written, we set the pin to a high impedance state. The process repeats after an appropriate delay accounting for the required setup and hold time.

We currently hardcode the testcase in the World implementation. To expose an interface, we would also replace the highest layer with boilerplate C code that would e.g. do non-blocking reads from a UNIX pipe to receive the commands to be sent.

The total required boilerplate code consists of 128 lines of code; most of it implements interfacing with Linux’s file-based sysfs GPIO interface. The statemachines expressed in the DSL are 830 lines
Chapter 4 - \( I^2C \) Specification

(including both, controller and responder). The generated \( I^2C \) state machine code consists of 2678 lines and compiles to a binary of 32 KB.

For comparison (see Figure 4.13), the hand written Linux GPIO based bit-banging driver uses 1221 lines of code (524 lines in \texttt{i2c-gpio.c} that accesses the in kernel GPIO interface and 697 lines in \texttt{i2c-algo-bit.c} that implements the \( I^2C \) interface). Of course this comparison is not entirely fair: Linux tries to provide a one-size-fits all implementation and thus has many more special cases hardcoded.

A popular (more than 100 stars on github) \( I^2C \) bit banging library [Bit18] targeting microcontrollers and the Linux GPIO interface uses about 800 lines (excluding the device specific probing but including counting all the preprocessor directives for accessing GPIO on different plattforms).

Even though these software stacks do not offer precisely the same functionality it should be evident that our approach does not require significant (if at all) additional effort.

4.6.4. Synthesizing Hardware

Generating C code down to the electrical layer (as done in Section 4.6.3) suffers from the usual problems of a bit-banging implementation: To correctly incorporate listening on the bus (as required for devices, clock stretching and multi-controller) the bus must be constantly polled. Under certain assumptions (such as there being only one master) it is possible to not permanently listen on the bus. But even then, running a transaction requires a lot of wasted CPU cycles.

We believe a better approach is not to make any assumptions and rather generate hardware from the DSL statemachine description. Using this technique, the bus can be constantly monitored without any intervention from the CPU. On certain events, the device could also raise an interrupt so that the CPU does not have to be polling for higher layer events.
Figure 4.13.: Lines of code comparison of different I²C implementations

- dsl
- gen
- linux
- bb

# lines of code

GPIO access implementation generated code
Our current implementation generates VHDL from the state machines, and we provide boilerplate code that redirects the output and input of a statemachine to the GPIO pins. Higher layers can be replaced by a memory mapped interface which then in turns connects to an AXI based interface. AXI is the standard on-chip ARM bus and interacts with the memory system. We implement this on a Zynq7000 SoC [Xil] which combines a hard ARMv7 dualcore and a FPGA.

For evaluating the functional correctness, we run the same code as in [Section 4.6.3] and use the same EEPROM connected to the FPGA.
4.7. Conclusion and Future Work

We have successfully demonstrated that our approach of creating and verifying layered specifications for $I^2C$ is feasible, and that it can be used to express bus interactions on a high level, to specify the expected behavior, and to verify that the specification fulfills this property. We have shown that the layering allows expression of partially compliant devices and thus that this way of verification is feasible in scenarios where relying on a strict specification is not possible.

Furthermore, the executable specification can be used to generate imperative, compact C code as well as an FPGA implementation. Both implementations have been shown to interact with real physical devices with minimal boilerplate code.

**Generality** While this work has already proven to verify desirable properties in specific scenarios, we cannot yet claim full generalization. As we omitted some $I^2C$ features we do not yet handle broadcast, variable length read transactions and multi-controller.

**The choice of model checking** The choice of model checking as benefits and drawbacks. It allowed us to create a tool that needs little developer effort for stating the correctness property (writing the `LayerSpec` and `LayerValid`). But the state space exploration can quickly run into time and memory limits when dealing with long messages.

An approach based on SMT solvers might offer a compromise between developer effort and time requirements. Ideally the approach is combined: Model checking can still be used to brute-force the guarantees up to and including the byte layer, but then the arbitrary long messages of the transaction layer could be encoded as a SMT problem - assuming the correctness guarantees of the lower model checked lower layers.

**The choice of a deterministic language for device models** The choice of the same language for responder and controller models has practical benefits. A user needs only one language and the code
generation can be used to generate code for devices as well. Our approach permits non-determinism, but it has to be injected from the Promela LayerValid process to the topmost layer. This can be inconvenient if the non-determinism is to occur at a lower layer in the stack. For example to model the disappearance of the 24xx16 EEPROM (see Section 4.2.2.2) we have to modify every layer only to produce (non-deterministically) an EEPROM busy action in LayerValid. The action is then handed down the stack which prevents the generation of an address acknowledge. This is not very convenient, future versions of the tool could offer support to write non deterministic device models directly in Promela, or any other suitable language.

**Verifying the hardware software interface and timing assumption**

While we do currently generate synthesizable hardware descriptions that produce functionally verified FPGA implementations, we have timing assumptions that might be violated as described in Section 4.6.4.

A simple way to alleviate the timing assumptions is to decouple the driver from the devices. One way do this is to create a command buffer and always let the hardware state machine advance. If the buffer is empty, by inserting explicit idle messages. In the case we are synthesizing a responder, this would lead automatically to clock stretching - as this is the interpretation of the idle command at the lowest layer. The concurrent interaction with the command buffer would ideally be verified.

**Verifying the layer calculus.** From a theoretical perspective, we so far assumed that our layer calculus itself is correct, i.e., we assumed that a layer that follows the specification can be combined with any (lower and higher) layer that also fulfills the layer contract. Given the higher order nature of this, we think that either an embedding in an existing process calculus or a from-scratch formalization in an interactive theorem prover would be useful to show the correctness of the approach. The second option would also open the possibility to reason about the system not only in a model checker, but in a theorem
prover. This could lift the restriction that we verify only on a small set of payloads, at the cost of some manual proof engineering.

**C as backend language** Chosing plain C as backend language is the obvious choice for a universal systems programming language. But since our model requires to save state at each `yield`, we generate rather unidiomatic C code that might not interface nicely with idiomatic C code. It should be possible to match better programming models that offer abstractions such as channels (like the Go programming language) or some form of coroutines (like it has been proposed for Rust [Rus]).
In this dissertation, we built formal models of hardware subsystems that were either not considered from a formal perspective so far (interrupts) or existing formalizations are not applicable to real world systems (I²C). Despite their absence in verification efforts, both subsystems are critical components.

Correct interrupt delivery is important, for example, in a RTOS; if the scheduler does not receive timer interrupts, the OS cannot ensure the guarantees promised. Interrupt delivery is also becoming more important: Kernel bypass network stacks and device pass-through for virtual machines require frequent correct reconfiguration of the direct interrupt delivery mechanism. We formalized the interrupt system in a way that captures the forwarding behavior of real interrupt controllers and is suitable for a constraint solver. The approach of using a constraint solver for finding interrupt configurations is implemented in the Barrelfish operating system. In the case of interrupts, we found a
suitable declarative description of an interrupt controller: Its abstract forwarding behavior in a decoding net.

I\(^2\)C is often used to set voltage levels of regulators, a failure to do so correctly can lead malfunction or even permanent hardware damage due to overvoltage. When verifying I\(^2\)C interactions, our first try was to express similar high level, abstract interactions. But it became clear that it is not possible to rely on an existing standardization of a transaction as not all real world devices adhere to it. Thus we created a formalization of a transaction as well as lower level concepts, using a layered approach that does both, crisply define transaction and expose it as a building block for higher level driver and device implementations. Incompatible devices can be incorporated elegantly by accessing lower layers of the stack.

5.1. Future Work

We already discussed extensions of individual chapters in Section 3.5 and Section 4.7. We briefly discuss future work that could use both models.

Both formalizations of I\(^2\)C and interrupts are contributions towards verifying whole systems instead of components. While Hardware/Software co-verification has been proposed before for designing embedded systems [SG00], our work allows including interrupts and I\(^2\)C interactions in such a verification effort. Given the widespread use, trust and criticality of hypervisors in cloud usage scenarios, we believe hypervisors are an interesting verification target of an open system (i.e. not an embedded system), where new VMs arrive at any point in time and for performance reason perform device pass-through.

Another interesting verification target is proving the isolation of trusted firmware (such as the ARM trusted firmware or the Intel management engine) from an untrusted operating systems. The I\(^2\)C model can be used as a basis for segmenting a shared I\(^2\)C bus into critical devices which should only be accessible from the trusted firmware and untrusted devices that can be accessed from both. The
interrupt model can be used to verify if the configurations are allowed that keep the trusted devices interrupting the trusted firmware. In this scenario the trusted firmware would interpose interrupt configuration requests and the model would be used as an admission test for the desired configuration.

Both of these chapters can be seen as a more complete, formal description of hardware subsystems. While none of them can replace the existing hardware descriptions such as ACPI, devicetrees and datasheets alone, they provide the concepts and abstractions that should be contained in those hardware descriptions.

A social problem emerges: How do we convince manufacturers to include formal, machine readable descriptions in (or along) their datasheets? ARM notably started to offer machine readable ISA semantics [Arm21], maybe the time for further such efforts is ripe. For manufacturers, this creates a legal problem: Should a manufacturer be liable if the formal guarantees are violated by the actual product? If so, they might refrain from publishing those specifications.

If manufacturers can not be convinced to include these specifications, maybe we can create such descriptions. To avoid creating just another standard we believe a new approach is needed, and for example be backwards compatible with a large ecosystem (such as devicetrees), similar in spirit to what TypeScript is to JavaScript. We also believe that no hardware description is ever complete, thus extensibility is crucial. Different sub communities (say the Linux community and an academic research team) should be allowed to incorporate features that matter only to them, without disrupting the other team.

To avoid fragmentation (as happened with devicetrees: U-Boot, Zephyr and Linux contain largely different devicetree descriptions of the same hardware) it might not be enough to just create a new description format or language. Rather a database of all hardware is needed. It is unrealistic to expect any corporation, research institute or individual to maintain such a database, but the open source community as a whole might be able to do so. Some crowd sourced databases such as Wikipedia, Wikidata and Openstreetmap have been successful, but it is an open question on how to structure such a database and
offer enough incentives for contributors and users in the case of hardware descriptions. The growing hardware diversity might make this proposition more attractive as time passes.
List of Tables

2.1. Interrupt models compared . . . . . . . . . . . . . . . 37

3.1. Characteristics of interrupt controllers, showing input and output width and the type of configurability . . . . . . 79

3.2. Time for solving for an interrupt route query in comparison to driver startup time on Barrelfish . . . . . . 108

4.1. Verification time in seconds using different layers of abstraction . . . . . . . . . . . . . . . . . . . . . 153
List of Figures

2.1. Devicetrees in the Linux kernel repository . . . . . . 31

3.1. Simplified x86 interrupt network. The highlighted
path is the example mentioned in the introduction. . . 41
3.2. Simplified ARM interrupt network . . . . . . . . . . 42
3.3. Overview and model of the interrupt system of the
Texas Instrument OMAP 44xx SoC . . . . . . . . . . 59
3.4. Schematic overview and model representation of the
desktop PC . . . . . . . . . . . . . . . . . . . . . . 60
3.5. Overview of the x86 interrupt message bus [Int16a]. 82
3.6. The ARM GIC architecture. [ARM16b] . . . . . . . 95
3.7. System Architecture . . . . . . . . . . . . . . . . . . 102
3.8. Overview of the predicates and queries. Red boxes
are runtime processes, gray boxes are static files, and
blue files are generated. . . . . . . . . . . . . . . . . . 103
3.9. Solving time for interrupt routes . . . . . . . . . . 109

4.1. The four I²C bus symbols . . . . . . . . . . . . . . . 118
4.2. Example I²C transaction with two messages: write
one byte, then read two bytes. SDA:C and SDA:R
are SDA signals asserted by controller and responder
respectively. The address and byte transfers have been
abbreviated. . . . . . . . . . . . . . . . . . . . . . . . 119
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Example I²C communication with an AS5011. Gray shaded parts are driven by the controller, white by the device.</td>
<td>121</td>
</tr>
<tr>
<td>4.4</td>
<td>Layering of the example</td>
<td>127</td>
</tr>
<tr>
<td>4.5</td>
<td>Verification processes for verifying the Bus level. Gray processes are generated from the DSL; blue ones are expressed in Promela</td>
<td>131</td>
</tr>
<tr>
<td>4.6</td>
<td>Workflow of the user provided files (white), intermediary files (yellow), and tools (gray)</td>
<td>134</td>
</tr>
<tr>
<td>4.7</td>
<td>Verification illustration, the system is verified if $\omega_i = \omega_j$</td>
<td>137</td>
</tr>
<tr>
<td>4.8</td>
<td>Layering of the I²C model</td>
<td>140</td>
</tr>
<tr>
<td>4.9</td>
<td>Interface of the symbol layer. The label describes the datatype of all the channels in this layer</td>
<td>141</td>
</tr>
<tr>
<td>4.10</td>
<td>Interface of the Byte layer. Both controller and responder have the same signature for <em>actions</em> and <em>results</em></td>
<td>143</td>
</tr>
<tr>
<td>4.11</td>
<td>Interface of the Transaction layer</td>
<td>145</td>
</tr>
<tr>
<td>4.12</td>
<td>Interface of the Driver layer</td>
<td>146</td>
</tr>
<tr>
<td>4.13</td>
<td>Lines of code comparison of different I²C implementations</td>
<td>155</td>
</tr>
</tbody>
</table>
List of Acronyms

ACPI  The Advanced Configuration and Power Interface
API   Application Programming Interface
APIC  Advanced Programmable Interrupt Controller
APM   Advanced power management
ASL   ARM Architecture Specification Language
BIOS  Basic Input/Output System
DFS   Depth-first search
DMA   Direct Memory Access
DS    Destination Shorthand
DSL   Domain-specific Language
GIC   Generic Interrupt Controller
GSIV  Global system interrupt vector
HDD   hard disk drive
HDL   Hardware description language
HPET  High Precision Event Timer
IC    Interval Constraints
ICR   Interrupt Command Register
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTID</td>
<td>Interrupt Identifier</td>
</tr>
<tr>
<td>IOAPIC</td>
<td>IOAPIC</td>
</tr>
<tr>
<td>IPI</td>
<td>Interprocessor Interrupt</td>
</tr>
<tr>
<td>IRS</td>
<td>Interrupt Routing Service</td>
</tr>
<tr>
<td>IRT</td>
<td>Interrupt Remapping Table</td>
</tr>
<tr>
<td>IRTE</td>
<td>Interrupt Remapping Table Entry</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>ITS</td>
<td>Interrupt translation service</td>
</tr>
<tr>
<td>ITT</td>
<td>Interrupt Translation Table</td>
</tr>
<tr>
<td>LAPIC</td>
<td>Local APIC</td>
</tr>
<tr>
<td>LPI</td>
<td>Locality-specific Peripheral Interrupts</td>
</tr>
<tr>
<td>MSI</td>
<td>Message Signaled Interrupt</td>
</tr>
<tr>
<td>MSI-X</td>
<td>MSI-X</td>
</tr>
<tr>
<td>NVM</td>
<td>Non-volatile Memory</td>
</tr>
<tr>
<td>OS</td>
<td>operating system</td>
</tr>
<tr>
<td>PCH</td>
<td>Platform Controller Hub</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI Express</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PIC</td>
<td>Programmable Interrupt Controller</td>
</tr>
<tr>
<td>RH</td>
<td>Redirection Hint</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-time Operating System</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SKB</td>
<td>System Knowledge Base</td>
</tr>
</tbody>
</table>

170
SMBus  System Management Bus
SPI    Shared Peripheral Interrupts
SR-IOV Single Root I/O Virtualization
SoC    System-on-Chip
VM     Virtual Machine
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Conference/Journal</th>
<th>Pages/Publication Details</th>
</tr>
</thead>
</table>


pp. 67–73. ISBN: 978-1-4503-7017-2. DOI: 10.1145/3365137.3365401 (cit. on pp. [26][113]).


[Hut] Ben Hutton. JSON Schema (cit. on p. [35]).


182


Curriculum Vitæ

Lukas Humbel

Education

2016–2022  Doctor of Science (PhD) in Computer Science, ETH Zurich
2011–2013  Master of Science ETH in Computer Science, ETH Zurich
2007–2011  Bachelor of Science ETH in Computer Science, ETH Zurich

Professional Experience

Research Assistant, ETH Zurich  Zurich, Switzerland. 2016-2022.
Intern, Hewlett-Packard  Palo Alto, Ca, USA. 2015.
Chapter A - Curriculum Vitæ

Publications

PLOS2021  *Generating correct initial page tables from formal hardware descriptions.*

SPIN2021  *A model-checked I2C specification.*
Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Häcki, Melissa Licciardello, Jan Schär, David Cock, Michael Giardino and Timothy Roscoe. July 2021. 27th International SPIN Symposium on Model Checking of Software (SPIN2021)

HOTOS2021  *mmapx: Uniform memory protection in a heterogeneous world.*

ATTAC2019  *Memory-Side Protection With a Capability Enforcement Co-Processor*

ITP2018  *Physical addressing on real hardware in Isabelle/HOL.*
Reto Achermann, Lukas Humbel, David Cock, Timothy Roscoe. June 2018. 9th International Conference on Interactive Theorem Proving (ITP’18), Oxford, United Kingdom.
PLOS2017  Towards Correct-by-Construction Interrupt Routing on Real Hardware
Lukas Humbel, Reto Achermann, David Cock, Timothy Roscoe. October 2017. 9th Workshop on Programming Languages and Operating Systems (PLOS’17), Shanghai, China.

MARS2017  Formalizing Memory Accesses and Interrupts.

Patents

Teaching
Fall 2021  Computer Systems (head TA, 252-0217-00L)
Spring 2021  Application-Oriented Programming with Python (252-0840-02L)
Fall 2020  Computer Systems (head TA, 252-0217-00L)
Spring 2020  Application-Oriented Programming with Python (252-0840-02L)
Fall 2019  Computer Systems (head TA, 252-0217-00L)
Spring 2019  Application-Oriented Programming with Python (252-0840-02L)
Fall 2018  Systems programming (head TA, 252-0217-00L)
Spring 2018  Application-Oriented Programming with Python (252-0840-02L)
<table>
<thead>
<tr>
<th>Semester</th>
<th>Course Title</th>
<th>Course Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fall 2017</td>
<td>Systems Programming and Computer Architecture</td>
<td>252-0061-00L</td>
</tr>
<tr>
<td>Spring 2017</td>
<td>Data Structures and Algorithms.</td>
<td>252-0002-00L</td>
</tr>
<tr>
<td>Fall 2016</td>
<td>Systems Programming and Computer Architecture</td>
<td>252-0061-00L</td>
</tr>
<tr>
<td>Spring 2016</td>
<td>Internet Architecture and Policy</td>
<td>851-0740-00L</td>
</tr>
</tbody>
</table>
B. Configurable Interrupt Model Refinement

This appendix describes the stepwise refinement of the static interrupt model to the configurable one presented in Section 3.3.1.

On the most abstract level (0), we define admissible configurations of a system as a set of net \( \text{net}_0 : \{\text{net}\} \). A concrete configuration is an element of that set. Recall that a net (Definition 1) is defined as \( \text{net} : \text{nodeid} \rightarrow \text{node} \). We also define the decode relation over \( \text{net}_0 \), a name decodes to another if all the configurations decode those two names:

\[
(a, b) \in \text{decode}_0 \text{net}_0 \iff \forall n \in \text{net}_0. (a, b) \in \text{decode}(n)
\]

Such a formulation is too generic: It allows interdependence of configuration options between two nodes. In practice we want to specify and reason about a node (controller) in isolation. Thus a better approach is to define a configurable \( \text{net}_1 \) as follows:
Chapter B - Configurable Interrupt Model Refinement

Definition 5. net1

\[
\begin{align*}
\text{nodeid} &= \mathbb{N} \\
\text{vec} &= \mathbb{N} \\
\text{name} &= (\text{nodeid}, \text{vec}) \\
\text{net}_1 : \text{nodeid} &\rightarrow \{\text{node}\}
\end{align*}
\]

A concrete $c'$ configuration is (of type $c' : \text{nodeid} \rightarrow \text{node}$) is valid if $\forall n. c'n \in \text{net}_1n$. This is a refinement of the set of the net_0 representation.

In this definition, it is not possible anymore that one controllers configuration depends on the configuration of another. Next, we introduce a (for now) opaque configuration type nodeconf, conf defines the set of valid configurations of a nodeid.

Definition 6. net2

\[
\begin{align*}
\text{nodeconf} &= \ldots \\
\text{conf} : \text{nodeid} &\rightarrow \{\text{nodeconf}\} \\
\text{nodes} : \text{nodeid} &\rightarrow \text{nodeconf} \rightarrow \text{node} \\
\text{net}_2 : (\text{conf}, \text{nodes})
\end{align*}
\]

This representation can again be lifted to the net_1 representation: \[
\text{lift}_1(\text{conf}, \text{nodes}) := \{\lambda n. \text{nodes} n c | c \in \text{conf} n\}.
\]

The nodeconf $\rightarrow$ node type is still not ideal as nodes contain global nodeids, identifiers that are not necessarily known when describing an interrupt controller. We localize nodeids by introducing ports and connections. We further assume that only the translating part can be reconfigured, which is sufficient as we will see in Section 3.3.4.

Definition 7. net3

\[
\begin{align*}
\text{port} &= \mathbb{N} \\
\text{acc} : \text{nodeid} &\rightarrow \{\text{vec}\} \\
\text{conf} : \text{nodeid} &\rightarrow \{\text{nodeconf}\} \\
\text{nodes'} : \text{nodeid} &\rightarrow \text{nodeconf} \rightarrow \text{vec} \rightarrow \{(\text{port}, \text{vec})\} \\
\text{conn} : \text{nodeid} &\rightarrow \{(\text{port}, \text{nodeid})\} \\
\text{net}_3 : (\text{acc}, \text{conf}, \text{nodes'}, \text{conn})
\end{align*}
\]
This representation has a decoupled nodes’ representation. A controller node (of type nodeconf → vec → {(port, vec)}) can be described independent of the net. We define a lifting lift2 function from net3 to net2:

**Definition 8. lift2**

\[
\text{lift2}(\text{acc}, \text{conf}, \text{nodes'}, \text{conn}) := (\text{conf}, \\
\lambda n c. \text{node} \{ \\
\quad \text{accept} := \text{acc} n, \\
\quad \text{translate} a := (\text{nodes'} n c a) \circ (\text{conn} n) \\
\})
\]

Where \( \circ \) is defined as \( A \circ B := \{(a, n)|(p, a) \in A \land (p, n) \in B\} \)

We observe that many controllers are of the same type. We incorporate this by an indirection via class.

**Definition 9. net4**

\[
\text{classid} = \mathbb{N} \\
\text{acc} : \text{nodeid} \rightarrow \{\text{vec}\} \\
\text{conf} : \text{nodeid} \rightarrow \{\text{nodeconf}\} \\
\text{class} : \text{classid} \rightarrow \text{nodeconf} \rightarrow \text{vec} \rightarrow \{(\text{port}, \text{vec})\} \\
\text{inst} : \text{nodeid} \rightarrow \text{classid} \\
\text{conn} : \text{nodeid} \rightarrow \{(\text{port}, \text{nodeid})\} \\
\text{net}_4 : (\text{acc}, \text{conf}, \text{class}, \text{inst}, \text{conn})
\]

As before, we define a lifting function lift3 : net4 → net3

**Definition 10. lift3**

\[
\text{lift3}(\text{acc}, \text{conf}, \text{class}, \text{inst}, \text{conn}) := (\text{acc}, \text{conf}, \\
\lambda n.\text{class} (\text{inst} n), \text{conn})
\]
While it is possible to use the lifting and use the \textit{decode relation} defined on the most abstract net, it is more suitable for solving to define the decode relation also on $net_4$.

\textbf{Definition 11. \textit{decode}}

\[(a, n), o \in decode_{net} \leftrightarrow \forall c \in conf\; n.\; o \in class\; (inst\; n)\; c\; a \circ conn\; n\]

Where $net = (acc, conf, class, inst, conn)$

Using the definition of $decode_4$ we define the transitive closure as route.

\textbf{Definition 12. \textit{route}}

\[route_4(net_4, n) = \bigcup (n', n) \in decode_4(net_4).route_4(net_4, n')\]

In this formulation, we have captured the configurability of a controller class independent of the net. We now further specialize the node configuration type. We found it useful to split the classes (and their configurations) into two sets: \textbf{explicit} (defined using $classE$) and \textbf{distinct} (defined using $classD$). The next refinement in \textbf{Definition 13}. Compared to $net_4$, \textit{class} is replaced with $classE$ and $classD$, the remaining elements of the net tuple are not changed. \textbf{Definition 14} defines how the new types relate to the \textit{class}.

\textbf{Definition 13. \textit{net5 and nodeconf}}

\[nodeconf : explicitC \; [N] \mid distinctC \; vec \rightarrow (port, vec)\]

\[classE : classid \rightarrow [N] \rightarrow vec \rightarrow \{(port, vec)\}\]

\[classD : classid \rightarrow vec \rightarrow \{(port, vec)\}\]

\[net_5 : (acc, conf, classE, classD, inst, conn)\]

\textit{Reusing the definitions from \textbf{Definition 9}}
Definition 14. \textit{lift4}

\[
\text{lift4}(\ldots, \text{classE}, \text{classD}, \ldots) = (\ldots, \\
\text{class classid nodeconf vec} = \\
\text{if nodeconf} = \text{explicitC} \; xs \; \text{then} \\
\text{classE}(\text{classid}, \text{xs}, \text{vec}) \\
\text{if nodeconf} = \text{distinctC} \; p \; \text{then} \\
\{p(\text{vec })\} \cap \text{classD}(\text{classid}, \text{vec}) \\
\ldots)
\]

If the partial function \( p \) is not defined on this input, the set shall be empty.