ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH zürich

Bachelor's Thesis Nr. 368b

Systems Group, Department of Computer Science, ETH Zurich

Generating Power Management Code from Declarative Descriptions

by

Linus Ulysses Vogel

Supervised by

Daniel Schwyn, Michael Giardino, Prof. Timothy Roscoe

April 2021–October 2021

D INFK

Abstract

Power management is a problem any modern computer must solve in order to work. Sometimes, computers make use of a BMC, which solves this problem in software, however the software to do so is not generally available but proprietary. Previous work at ETH Zürich has created an algorithm to solve this problem and synthesize power sequences from a declarative specification of the system.

In this thesis, I have developed a backend compiler that translates such synthesized abstract power sequences to C code which then can be compiled into an executable. I aim to set the basis for a compiler tool chain that can generate native executables for arbitrary power sequences from a declarative specification. Testing has shown that the compiler is capable of producing C programs from nothing but declarative power sequences and some information about the target system in an efficient manner.

Acknowledgements

Before starting I would like to take the opportunity to thank the Systems Group at ETH Zürich and especially Daniel Schwyn and Dr. Michael Joseph Giardino, who have not only helped me concretize my project at the beginning, but also provided me with much support through any technical problems, or lack of knowledge on my end, I faced with Enzian and their tools. and have provided invaluable feedback while writing this thesis.

Contents

Listings

List of Figures

Chapter 1

Introduction

Since the beginning of their time, computers have increased massively with regards to complexity. As a direct consequence of this increase, modern computer hardware is no longer limited to the components that absolutely need to be exposed to the operating system, but contain more hidden devices that perform various functions to allow the computer to operate. One such device is the Baseboard Management Controller, which I will call the BMC in the remainder of this thesis.

1.1 BMCs and what they're supposed to do

Generally the BMC could be almost any sort of device ranging from simple ICs over microcontrollers, CPLDs, FPGAs to even small form factor fully blown computers in and of itself. BMCs manage the lowest levels of operation in the system and ensure a safe and reliable operation of all components present in the system. To achieve this goal, they need to, amongst many other things, manage the power and clock delivery of the system, that the rest of the components rely upon.

1.2 The Problem with BMCs

As important as BMCs are in the modern world, the vast majority of such devices remain closed source and proprietary with little to no work surrounding them published, with the biggest available open source solutions being only a few years old.

Recent work by te systems group at ETH Zürich has created an algorithm that generates power sequences from a declarative specification of the system. Although this is a great step towards safe and reliable open source BMC software, its usage of Python from the implementation itself to the output being written in Python, is far from ideal, both from

the performance and verifiability point of view since many of the dependencies used through Python are community built and not proven to be correct. In addition, some devices do not quite adhere to the communication standard they claim to support, which of course makes managing a system using such devices quite a challenge.

All of the above hinder the feasibility of their solution in a productive environment and thus limit the usability of their implementation for actual use on a running system.

1.3 A solution to this problem

In order to improve this situation, I have developed a basis for a compiler backend that can compile an abstract power sequence down to C code to create a native executable. The compiler is written in a functional language and produces C code with only the C standard library and a custom runtime library as dependencies which will greatly facilitate future verification attempts, both by using a functional language and by clearly separating the synthesization of the sequence and the generation of code. In addition it is able to handle non standard compliant behaviour of devices through a declarative specification of the behaviour of such commands. Lastly, it provides a more performant execution of the native power sequences and compiler itself than the currently existing Python implementation and may thus even make dynamic power management a possibility, which was not yet achieved by previous work[15]. With this work, I aim to provide a basis for a efficient and provably correct compiler chain for not only generating power sequences, but also generating correct C programs to execute them efficiently.

I have tested my solution on actual hardware and was able to demonstrate not only its functionality, but also the feasibility of my approach. It performs mostly as expected with only minor issues that should be possible to fix in the future.

Chapter 2

Background

2.1 The Enzian System

This thesis has been heavily influenced by the Enzian system. Enzian is a research computer, designed by the Systems group at ETH Zürich. It is meant to provide a research environment for operating systems, hardware acceleration, high-performance computing and many more topics. It features a high end server grade CPU tightly coupled to a large FPGA and provides much memory and network bandwidth on both sides[4].

Like most of modern systems, Enzian has a BMC to manages its hardware on the lowest level. However, as is mentioned in the introduction, the software running on BMCs is not widely available to the public and little research has been carried out in this direction. As a result, the power management software currently running on the BMCs of Enzian systems is a self developed set of Python scripts where a lot of hard work went into[3].

2.2 The Baseboard Management Controller

As I have mentioned already in the introduction, the BMC can technically be any sort of device of greatly varying complexity. In the case of the Enzian system, the latter case applies, since its BMC is is a fully blown computer with a fully fledged multicore ARM processor, a healthy amount of system memory, its own flash storage, network capability, various peripherals and even a small onboard FPGA[5]. As is evident from this description, this device is clearly a complex piece of engineering in and of itself and may very well be more complicated than many computers that were running at the cutting edge of computing capability only a few decades ago. In the case of a research computer like Enzian, this opens up the opportunity to experiment with different designs of the BMC firmware or even smaller hardware.

The BMC's main function is to, as its name implies, manage the base board of the system. This includes, but is not necessarily limited to, managing clock distribution and synchronization for devices dependent on such clocks and managing power distribution and delivery to the various powerhungry components usually present in the system (such as the CPU, which can dissipate hundreds of watts in some cases).

Most importantly, the latter includes providing valid startup and shutdown sequences that ensure not only the safe operation of all devices of the system but also guarantee that no component exceed its designed limits.

Unsurprisingly, such low level access to the system opens the BMC up to various safety and security concerns, and thus it is important to know about the correctness of BMCs in order to provide secure and reliable platforms. In recent years, a few projects such as OpenBMC and u-BMC have started to to set the foundation of open source BMC firmware to address such issues[15].

2.3 Preceding work

This project is a continuation of previous work done by the Systems Group of ETH Zürich. They have developed a program that generates power sequences from a purely declarative specification of a system[15]. Since the correct management of power and clock signals is imperative for the system to run safely and reliably, the formal verification of such power sequences is of great interest to the research community and therefore such an approach, facilitating such formal verifications, is well founded in this situation. The system developed by Schult et al, is already capable of producing power sequences that meet expectations and provides the basis for a formal verification of power control circuitry of current and future machines. In addition to the already mentioned advantages, they also claim to reduce the development time for deriving a working power sequence and also the effort needed to update a power sequence if the hardware design evolves[1].

Not only have they provided a model language to capture power trees of modern systems and a technique to synthesize correct power sequences, but they have also created an implementation of the aforementioned and demonstrated the applicability of their approach on Enzian. This implementation is currently handling the synthesization of the power sequence and the generation of code that executes it and on top of that is written in Python, which hinders their solution in modularity, because the synthesization of power sequences does not need to directly interact with the code generation, and performance and verifiability, since Python is neither a very performant language, nor does it provide the same degree of assistance when verifying code, as many functional languages do.

2.4 Code generation

Code generators have been an integral part of many different compilers for decades now. Although code generation first only concerned itself with generating machine code for the

target platform, with more complicated and capable systems and development environments, higher level languages have started to appear in the output of code generators. Today, many development tools have abstracted the source code away from the developer and offer a graphical interface instead, while producing the necessary source code themselves.

Other environments produce C code from another, much more abstract or idealized, language and produce low level source code to aid with time efficient execution. One such example is the MATISSE compiler developed by Bispo et al., which is framework for compiling MATLAB code to C and even is capable of producing OpenCL code for acceleration of certain functions. They try to solve a similar problem and have faced similar problems like I have, especially with the introduction of typing system in a weakly/untyped context[10].

The paper 'An Approach to Generating C code with Proven LTL-based Properties', also states that automated code generation is very important in modern times and that large scale software development projects would be unfeasible without it. The abstraction that can be provided by a compiler greatly facilitates the development, maintenance, understanding and verification of large code bases that would be nearly unmaintainable and completely unverified, if they were directly written in lower level languages. The author does however also point out, that such abstraction and higher level languages usually produce some computational or storage overhead which is not always bearable, especially when using very high level abstractions on small embedded systems which are subject to hard storage and computational boundaries[14].

2.5 The I2C communication bus

I2C (shorthand for Inter Integrated Circuit) is a simple two wire communication bus that is widely used on many different IC's by many different manufacturers. It forms the bases for various different communication protocols, including protocols such as PMBus and SMBus, which will be explained in more detail below. I2C supports many different modes of operation ranging from various speed settings to multi-master operation including collision detection and many more features.

The communication across I2C always includes a master that initiates the transfer by writing a start condition and an address to the I2C bus, which alerts the slave device to handle the following communication. The master can then command the slave to either read or write to the bus by writing the read/write bit after which the data is transferred in the requested direction. This process of writing a start condition and transferring data is called a transfer. Transfers can be bundled together by simply initiating new transfer after completing the last one, without releasing the bus. After all transfers have been executed, the bus is released by the master by writing a stop condition, which tells all connected devices that the transaction has completed and the bus is no longer busy[16].

The above diagram depicts a simplified I2C transaction consisting of two I2C transfers and how the bus gets taken, using Start Conditions, and released, using Stop Conditions, by the master. The data direction is encoded in color, with white meaning master-to-slave and grey meaning bi directional transfer is possible. The address being transmitted at first

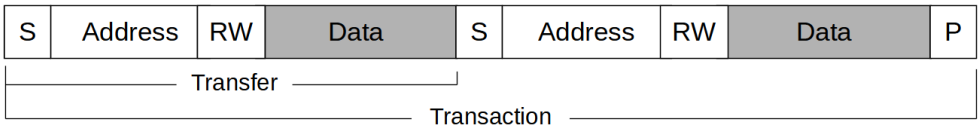

Figure 2.1: Simple Diagram of an I2C transaction with Start Conditions (S), Stop Conditions (P), Read/Write Bit (RW) and the data being transferred (white from master to slave, grey both directions possible)

serves to alert the target device to the communication and thus is always sent by the master. The following data can be transferred in both directions, however no change of direction can be performed during a transfer, but instead a new transfer has to be started with the new direction, either in the same transaction, or in a separate transaction. Some details, such as the acknowledgements, are not depicted in figure 2.1 above, however, these are conceptually more important for the physical layer of the I2C standard, and less for the network layer, which is what we're going to be interacting with during this thesis.

I2C is the de facto standard for communications between modules on a PCB. As such it is widely used by many hardware and software engineers and heavily relied upon in many systems running today. Despite this track record, only a few attempts have ever been made to create a formally verified implementation of the I2C bus[8]. Since the BMC and the devices it communicates over I2C with are of great importance to the safety and reliability of a large system such as Enzian, this lack of assurance in the communication systems is less than optimal to say the least. In the current state of the system, the current implementation of the declarative power sequencer as well as in my own implementation of the backend compiler, the I2C implementation exposed by the linux kernel is used and relied upon.

2.6 The SMBus protocol

The System Management Bus, or short SMBus, is a communication protocol that builds upon the I2C interface. SMBus provides a simple interface for transferring some standard sized data, arbitrary sized data and even some form of remote procedure calls, all of which optionally include Packet Error Checking (PEC). The SMBus standard specifically markets itself for system and power management and its ability to exchange manufacturer, control and status information[7].

The SMBus standard specifies a set of transactions that correspond to I2C transactions with one or more transfers in possibly both directions. Such an SMBus transaction consists of a command byte, that is always sent from master to slave, and some data that can be sent in either direction.

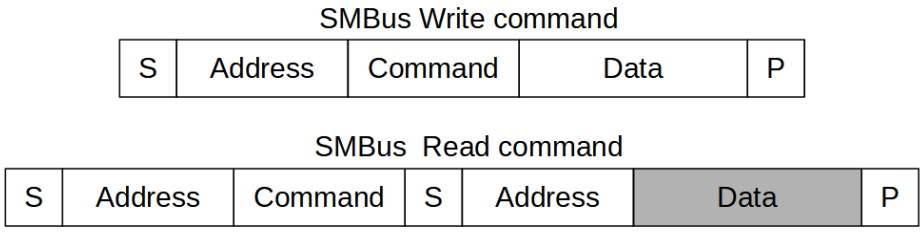

Figure 2.2: Simple examples for both read and write commands over the SMBus protocol, with white being master-to-slave and grey being slave-to-master.

Since the SMBus protocol builds on top of the I2C standard, the change of direction that was mentioned above is also visible here with the address being sent again in the middle of the transaction. Most of the commands that the SMBus standard specifies follow this general model, however SMBus also provides a simple send and receive operation, a so called Quick-Command, and some remote procedure calls. The quick command leaves out the command and data information and only transmits an address and a read/write bit before closing the transaction. This is intended for reducing bus usage for very simple and frequent tasks, such as enabling and disabling various devices. The remote procedure calls not only send a command but also some data before changing the direction of transfer and receiving some data.

2.7 The PMBus protocol

The Power Management Bus, or short PMBus, provides a means of communicating with various power management devices. It defines a set of commands on top of the SMBus protocol and an abstraction for said devices.

The commands that PMBus specifies are usually expressed as a 8-bit command number paired with a SMBus transaction for reading and another SMBus transaction for writing, though there are some commands that are either read-only or write-only. In addition, the interface that PMBus provides is paged, which means that every PMBus device present in the system can expose different subsets of the PMBus standard for different pages. This page is part of the state of a PMBus device and remains set until it is changed. Any page dependent command that is sent to the device, will happen on the currently set page for the device, regardless of any parameters of said command. Figure 2.3 below shows the first few PMBus commands, as listed in the PMBus protocol specification.

Using the fan controller on Enzian as an example, a typical interaction with the device would of course consist of the desired PMBus command, lets say the `READ_FANSPEED_1` command, but may also include a `PAGE` command beforehand to select the desired interface

Command Code	Command Name	SMBus Transaction Type: Writing Data	SMBus Transaction Type: Reading Data	Number Of Data Bytes
00h	PAGE	Write Byte	Read Byte	1
01h	OPERATION	Write Byte	Read Byte	1
02h	ON_OFF_CONFIG	Write Byte	Read Byte	1
03h	CLEAR_FAULTS	Send Byte	N/A	0
04h	PHASE	Write Byte	Read Byte	1
05h	PAGE_PLUS_WRITE	Block Write	N/A	Variable
06h	PAGE_PLUS_READ	N/A	Block Write – Block Read Process Call	Variable
07h	Reserved			

Figure 2.3: An excerpt from the PMBus protocol specification, depicting a few PMBus commands, their command numbers and some meta information.

on the device itself. Reading the fan speed of multiple fans connected to the same controller would thus consist of a `PAGE` command prefixing each of two `READ_FANSPEED_1` commands that read the fan speed on the current page. Conceptually, this interaction would look like the following:

1. `Page 0 (page number of the first channel)`
2. `Read_Fanspeed_1 (read the fanspeed on the current channel)`
3. `Page 1 (page number of the second channel)`
4. `Read_Fanspeed_1 (again, read the fanspeed on the current channel)`

This is of course formulated above the PMBus standard and thus does not show any lower operations. If one were to take a look at what happens below PMBus, the same interaction would look different, albeit quite similar:

1. PMBUS_PAGE 0

   SMBUS_WRITE_BYTE 0 with command 0x00 (The SMBus command number of the PMBus Page command is 0x00, as can be seen in figure 2.3)

   I2C_WRITE_BYTES [0x00, 0x00]

2. PMBUS_READ_FANSPEED_1

   SMBUS_READ_WORD with command 0x90 (The SMbus command number of the PMBus Read_Fanspeed_1 command is 0x90)

   I2C_WRITE_AND_READ_BYTES with write 0x90 and receiving 2 bytes

3. PMBUS_PAGE 1

   SMBUS_WRITE_BYTE 1 with command 0x00

   I2C_WRITE_BYTES [0x00, 0x01]

4. PMBUS_READ_FANSPEED_1

   SMBUS_READ_WORD with command 0x90 (The SMbus command number of the PMBus Read_Fanspeed_1 command is 0x90)

   I2C_WRITE_AND_READ_BYTES with write 0x90 and receiving 2 bytes

The above represents a 'trace' through the protocol stack, translating the PMBus commands first to SMBus commands and finally to I2C transactions.

Chapter 3

Implementation

To solve the problem outlined in chapter 1, I have developed a compiler that can translate an abstract power sequence to C code to perform said power sequence on some system. In this chapter, I will explain what exactly I have done and what exactly the compiler is capable of.

3.1 The compiler and the fancontroller

During the development of my compiler, I have been almost exclusively been working with the fan controller on the Enzian. While any reasonable system in the modern world will inevitably consist of many more components than a single fan controller, the fan controller offers a relatively representative yet safe environment to test the compilation for PMBus devices. In addition to the pure PMBus standard, the fan controller not only offers manufacturer defined commands, but also actively breaks the PMBus standard for some commands. Additionally, the fan controller is not only present on the partly populated Enzian boards, where any experimenting with the power management can only cause little to no damage, but it also is one of the few devices that is in perfect working order even on these partly populated boards, which means that fans can be attached the same way as if on a fully functional Enzian board. Lastly, the possibility to attach fans offers another advantage, especially during early development, since sequences can have both visible and audible effects that can be used as a confirmation of working communication.

The fancontroller is an MAX31785, 6-Channel Intelligent Fan Controller. This controller is PMBus capable and even breaks some PMBus and even SMBus specifications, which is a good opportunity, since such cases have to be handled as well. It implements a subset of about 50 commands of the PMBus specification and is well documented. It serves well as an example of an average PMBus capable device, since it not only makes use of the extensive set of standard PMBus commands, but also exposes quite a few so called manufacturer specific commands. These commands are obviously not defined in the PMBus standard, however, they are anticipated and have been assigned a command number, but

PMBus Standard

Command Code	Command Name	SMBus Transaction Type: Writing Data	SMBus Transaction Type: Reading Data	Number Of Data Bytes
99h	MFR_ID	Block Write	Block Read	Variable

MAX31785 Datasheet

CODE	COMMAND NAME	TYPE	PAGE 0–5	PAGE 6–16	PAGE 17–22	PAGE 255	NO. OF BYTES	FLASH STORED (NOTE 2)	DEFAULT VALUE (NOTE 2)
			(NOTE 1)						
99h	MFR_ID	Read Byte	R	R	R	R	1	FIXED	4Dh

Figure 3.1: Depicts the `MFR_ID` command as defined by the PMBus protocol specification, as well as its implementation according to the datasheet of the MAX31785. [6][9]

no corresponding SMBus transaction (as specified by the PMBus Protocol specification, under the `MFR_SPECIFIC_XX` commands[6]) and thus offer a very high degree of flexibility to the hardware manufacturer. Additionally, some of the standard PMBus commands that it exposes do not adhere to the PMBus specification. While this is not in the interest of the PMBus or SMBus standards, or any other standardization effort for that matter, this seems to be the situation in at least a few devices and thus needs to be considered if one wants to support as broad a set of devices as possible.

Lets take a look at some examples of different severity of this problem. Firstly, consider the `MFR_ID` PMBus command that is exposed by the fan controller. Figure 3.1 shows the difference between the standard and the implementation of the `MFR_ID` command, specifically how the implementation of the MAX31785 changed the SMBus transaction type from a block read *or* write transaction to a read byte, with no writing possible. This must be handled at some point during compilation of course, since the fan controller will not correctly respond to the block read of the PMBus standard.

The `MFR_ID` command does however not violate the SMBus standard and thus can still be implemented using only the SMBus interface. The `MFR_LOCATION` command in contrast, breaks the standard in a more subtle but destructive way. As figure 3.2 shows, the two SMBus transactions look very similar, however, the implementation of the MAX31785 does infer the length of the transfer to be 8 bytes, while the SMBus specification requires the length be sent before the data. This behaviour breaks the specification of SMBus and thus cannot simply be implemented by another SMBus transaction, but needs to be implemented in a raw I2C transaction. Non-compliances such as these necessitate a flexible and powerful way of implementing commands that does not include a recompile of the backend itself.

Lastly, consider the `MFR_FAN_CONFIG` command that the MAX31785 exposes. This is a manufacturer specified command and corresponds to `MFR_SPECIFIC_33` in the PMBus standard and thus the PMBus standard itself is insufficient to handle this command. In figure 3.3, I have included a comparison of the commands as they appear in both specifications,

PMBus Standard

Command Code	Command Name	SMBus Transaction Type: Writing Data	SMBus Transaction Type: Reading Data	Number Of Data Bytes
9Ch	MFR_LOCATION	Block Write	Block Read	Variable

MAX31785 Datasheet

CODE	COMMAND NAME	TYPE	PAGE 0–5	PAGE 6–16	PAGE 17–22	PAGE 255	NO. OF BYTES	FLASH STORED (NOTE 2)	DEFAULT VALUE (NOTE 2)
			(NOTE 1)						
9Ch	MFR_LOCATION	Block R/W	R/W	R/W	R/W	R/W	8	Y	(Note 3)

Figure 3.2: Depicts the `MFR_LOCATION` command as defined by the PMBus protocol specification, as well as its implementation according to the datasheet of the MAX31785. [6][9]

the PMBus one and the datasheet of the MAX31785. Due to its very nature, this command has to be defined on a device by device basis and cannot be included in a generic runtime. The compiler needs to be able to handle this command, no matter how the manufacturer decides to implement it, with some limitations of course.

The above makes it apparent that an abstract power sequence is not necessarily sufficient in terms of information about the desired output. For this reason, the compiler takes an additional input during the compilation that describes the devices present on the system and how they can be interacted with, in the rest of this thesis, this will be called the topology information and I will go into further detail in section 3.5

3.2 Concept

In order to successfully produce a power sequence for a device as the fan controller from a purely declarative specification, many vastly different problems need to be solved. Due to the differing nature of these problems, I have decided to split this compiler into a frontend and a backend. The concept of using multiple stages during the compilation process, usually a frontend and a backend, sometimes with an intermediate stage to improve optimization capabilities, is no new idea and has been around for at least 30 years, if not more. In such a model, the frontend concerns itself with problems on the input side, and produces some intermediate representation and the backend concerns itself with problems on the output side and concretizing the solution from this intermediate representation so as to fit a desired target platform[2].

In my case, there are quite a few problems that have already been solved along the way. For example, I don't really need to produce any machine code by myself, since there already exists a large amount of C compilers for any target platform one could wish for. For this reason I have decided to produce C code as output instead of machine code. In

PMBus Standard

Command Code	Command Name	SMBus Transaction Type: Writing Data	SMBus Transaction Type: Reading Data	Number Of Data Bytes
F1h	MFR_SPECIFIC_33	Mfr. Defined	Mfr. Defined	Mfr. Defined

MAX31785 Datasheet

CODE	COMMAND NAME	TYPE	PAGE 0–5	PAGE 6–16	PAGE 17–22	PAGE 255	NO. OF BYTES	FLASH STORED (NOTE 2)	DEFAULT VALUE (NOTE 2)
			(NOTE 1)						
F1h	MFR_FAN_CONFIG	R/W Word	R/W	—	—	—	2	Y	0000h

Figure 3.3: This figure depicts the definition of the PMBus command with command number 33 as specified in both the PMBus Protocol Specification and the datasheet of the MAX31785 [6][9]

addition, some version of the frontend already exists and is capable of producing working power sequences[1]. This is why I have decided to limit my work to the backend of the compiler and target a higher level language.

This backend compiler will take an abstract power sequence as input and produce C code describing a program that would execute this power sequence and change the state of the system as described by the power sequence.

The motivation behind these decisions are, apart from avoiding redundant work on my part, to decrease the necessary complexity of the frontend and backend compiler, as well as improve the modularity of the whole compilation chain. The current implementation of the sequence generator is written in Python and handles the task of both frontend and backend, which, although it works, in practice makes the the compiler much harder to maintain and troubleshoot because of tightly coupled code[15]. This tight coupling can be avoided, because the frontend does not need to know how to interact with the devices on the system, all it needs to know is what these devices can and cannot do and what requirements the sequence needs to follow in order to safely move the system from one power state into another. Actually figuring out how to communicate the sequence to the devices in the system can be left to the backend, which in turn does not need to know about many of the limitations the devices may have.

3.2.1 Abstraction Levels

The goal of abstraction in such a compiler chain is to isolate the solutions to different problems in a way that they can be used independent of the context the problem is encountered in. A compiler that is modelled after such a frontend/backend design pattern can easily be extended in functionality, for example by adding support for a new target architecture or

a new source language. The result can be an entire ecosystem of compilers that share the same levels of abstraction and the same intermediate representations and thus every single source language can be compiled for every single target platform using only one compilation module per source and target.

Although there are technically more than two stages in the compilation process of a power sequence, I will in this thesis mainly talk about a frontend, that produces an abstract power sequence from some declarative input and a backend that translates such an abstract power sequence into C code that can be compiled further down to machine code using any available C compiler. The reason for this is that I consider C as my target and thus the compilation is technically done when C code is produced. Of course this C code still needs to be compiled to machine code before it can be run, but since this problem has been solved for decades, I did not consider it part of my compiler.

3.3 Implementation of the backend compiler

Now that the concept of this thesis has been established, we can take a look at the actual implementation of the backend compiler. While the original implementation of the combined solution is written in Python, I have decided to implement the new backend in OCaml for the following reasons.

My choice for a functional language for this project is motivated by the fact, that most functional languages are very well suited for working with tree-like data structures and problems with a lot of recursion, as encountered when parsing the input files and writing the output. OCaml specifically was mostly a personal preference since I am more familiar with it than with other functional languages like Haskell. From my own experience, there is also a performance argument to be made here, as I find OCaml to be a lot more performant than Haskell. There have been comparisons between the two, that suggest that at least in some situations, this is in fact the case[12]. However since this compiler is not meant to run as a part of a time critical real-time system, the actual advantage that OCaml can offer in this regard is rather small and likely will not impact the future of this project.

Since the backend at this stage is a standalone program and not integrated into a compilation pipeline, there are some additional problems to be solved on top of the typical function of a backend compiler. The most clearly visible one being the need to parse some input, because there is no framework around the backend to create the abstract sequence the backend needs to work. For this reason, I have decided to read all the information required, including the actual input to be compiled, from a collection of files, as described above.

As the language to represent these input files, both the sequence and topology files, I have decided to choose YAML for its low verbosity, since I have written many input files manually and other languages like JSON tend to include a lot of punctuation and structuring and crucially: usually lack the support for multi-line strings, which I have made good use of in this project.

In the following I will include some samples of the code in either OCaml or YAML and mark them as such.

3.4 Input

As is the case with all compilers, some form of input must encode the desired output in an abstract way, for the backend compiler to translate it. The obvious part is the power sequence (see below in section 3.4.1) itself, which encodes the state transition the system should go through. But this information on its own does not suffice to generate a fully functional C program. While the power sequence encodes the desired state transition for the system, it does not contain any meta information about the system in question and thus the compiler cannot possibly handle the communication with multiple different devices over multiple different interfaces correctly. For this reason the backend also needs some sort of topology information about the system, that encodes the devices present in the system, the interfaces and protocols they use and devicetype specific information, such as the aforementioned non standard compliant behaviour of some devices.

I have split this topology into two different categories: the general topology and the device specific information, both of which will be explained in more detail in section 3.5. The abstract power sequence itself defines the main interface between the frontend and the backend of the compiler and is the only part of my work that directly needs to expose an interface on which other projects may rely. As such, I found it deserving of a set of definitions that describe the interface in detail, but first I want to make note of some requirements this interface has to fulfill.

3.4.1 Requirements for the Front/Back interface

Let us first state the goal of the interface in a naive fashion:

The interface should alow for any power sequence to be passed through it.

This however opens up a new question: what exactly can a power sequence be like? To answer this question lets start at the data that needs to cross the interface. Generally, a power sequence is supposed to guide the system through some set of state transitions to get from a starting state to a destination state.

Definition 1 *A power sequence s an ordered list of power commands that encodes a series of state transitions to be performed for moving the system from some source state to some destination state.*

Under the assumption that these commands take effect immediately and never require any sort of interaction between one another, this would likely suffice, however, even assuming that the commands do not depend on one another, there are cases where such simple command writes are not enough to encode a power sequence and the commands depend on the completion of another command beforehand.

To illustrate this issue, lets take the fan controller as an example. If we change the power state of the CPU to the maximum from a sleeping state of the system, it may be necessary to prepare the fan controller before the CPU starts to dissipate large amounts of energy.

The fan controller changes the fanspeed by slowly ramping the requested value to the target according to its configuration and may in some cases take several seconds to accelerate a fan from 0% to 40%[9]. In this case the power sequence may have to wait with setting the highest power state on the CPU until the fans have spun up and the system is able to carry the developed heat away from the CPU.

For this very reason, the commands crossing the interface not only need to be able to write commands and data to the devices, in some cases monitoring a value until it reaches some criterion is also required. This may seem to violate the assumption that commands would not depend on one another, however the commands actually depend on the effects that a device may report after executing another command.

With this gained knowledge, lets take a more detailed look at what such a power command needs to be able to accomplish. Trivially, there must be a mechanism for "writing" to a device, as in send a command and accompanying data to the device. And as I have described above, there also needs to be some sort of monitoring mechanism for halting the sequence until some state is reached.

Definition 2 *A power command is one of the following:*

- *A write command that can transfer some information from the BMC to some device.*
- *A monitor command that halts the sequence until some predefined state is reached.*

Of course these commands must carry some information about their target and the value they should modify or monitor. Figure 3.4 below illustrates what exactly the commands need to carry and section 3.4.2 demonstrates how the compiler represents this information internally.

1. Write Commands

   A write command needs to encode some data that should be written, as well as a destination for said data.

2. Monitor Commands

   A Monitor command needs to encode what data should be monitored, as well as what this data should be and when and how often to query it before raising some failure.

Figure 3.4: This depicts the requirements for the input sequences and describes what data the input must carry

3.4.2 Parsing the input

Since the input will be read from a file and parsing information from text has been solved already, I have made use of some libraries that do some of the work for me. Most notably,

that is the yaml package obtainable from opam[13]. This allows me to skip the entire parsing step and directly translate the YAML abstract syntax tree to a custom information tree.

My current implementation represents the parsed power sequence input as a a list of `power_command` according to the following type definitions.

```
type power_command =
  | Write of {
    device: string;
    control: string;
    value: int;
    }
  | Monitor of {
    device: string;
    input: string;
    delay: int;
    range: interval;
    timeout: int

type power_sequence = PowerSequence of
  power_command list
```

Listing 3.1: Type definition of power command and power sequence.

Since this is the frontend/backend interface described earlier, the requirements elaborated in definition 2 have to be satisfied. The write command can achieve this quite trivially, since it contains both the target control and the target value to be written, the backend is able to pass the value to the requested device. The monitor command is a bit less simple, since a part of its task is to halt until the value satisfies some condition. In the current state of my implementation this is a simple 'value inside of interval' check that is delayed until the value should have stabilized and and then periodically polled until either the value finally satisfies the condition, or the polling times out. The result of such a timeout is an error and should be reacted to.

While this does technically satisfy the requirements set in definition 2, there is room for improvement here, since not necessarily all values need to be monitored in quite the same way, the biggest difference being that not all situations necessarily can be met by such a simple interval check and may be more complex.

Since the frontend/backend interface does not include any information about the topology, this information has to be read from a file as well. As section 3.5 explained, this information is split into multiple files and consequently all of them pass through the same YAML parser and are also converted into a custom representation for quick and easy use. The following type definitions illustrate the structure of the topology in the compiler, which very closely resembles the listings and diagrams in section 3.5.

```
type topology = {
  device_types: (string * device_type) list;
  busses: (string * interface_bus) list;
  devices: (string * device) list;
  control_map: (string * (string * string)) list;
  monitor_map: (string * (string * string list) list) list;
  monitor_scale: monitor_scale list;
}

type device_type =
  | PMBus_Device_Type of pmbus_device_type

type pmbus_device_type = {
  data_formats: (string * pmbus_format) list;
  monitor_alias: (string * pmbus_monitor_alias) list;
  control_alias: (string * pmbus_control_alias) list;
  command_override:
    (string * (driver_action option * driver_action option)) list;
}

type interface_bus = {
  bus_name: string;
  bus_number: int;
  alerts: string list;
}

type device =
  | PMBus_Device of {
    device_type: pmbus_device_type;
    bus_name: string; address: int }


type monitor_scale = {
  device_name: string;
  scale: float;
}
```

Listing 3.2: Type definition of the topology and its dependencies.

Note that some types seem incomplete, and this is due to the fact that I only worked with PMBus devices and did not include the entire enzian platform, so some definitions might be present already, but not actually used yet, such as the `monitor_scale`. Also some supertypes are not strictly necessary in this implementation, since they're only subtyped once, such as the `device_type`. However, the inclusion of this subtype allows a simple extension for more diverse device types, again by reducing code coupling.

The conversion process first converts the device specific information and stores the result in an internal lookup table and then start converting the topology information. This specific order was chosen, because the strict separation between these two types of information is not necessary at this level, since this information must be easily accessible but not mutable, because it only persists for the running time of the compiler. To facilitate access to

the information, I decided to store both in a single `topology` datastructure and add the devicetype information to each and every device that is listed in the topology information.

3.5 Topology Information

The definitions and figures in section 3.4 above lack some very critical information about the system. As the frontend should not concern itself with the details of communicating with devices, this information is not included in the frontend/backend interface and thus will not be passed to the backend through the frontend. Consequently the backend has to get this information by some other means. This is why the backend takes a secondary input that describes the communication interfaces and capabilities that are present in the system.

This topology information must include all the details necessary to communicate any possible action to any possible device and thus must include details about the general architecture of the system, such as over which bus or with what address a device can be reached, but also device specific information, such as a detailed description of the communication interface. In case of the Enzian system, there are multiple devices that communicate over many different forms of protocols, from a simple GPIO interface to more complex PMBus capable devices[5]. For this reason, I have split the topology information into two distinct categories, which are represented by multiple files to improve maintainability. First, there is the category of General Topology, which is represented by a single file and captures the large scale architecture of the system. And secondly, there is the category of Device Type Specific Information, which is represented by a single file per device type, that captures the interface the device exposes.

To demonstrate the necessity of such an approach to split the information, lets think of a similar problem in cartography. Suppose we want to map an entire country, analogous to our attempt to describe an entire system. Of course we want to have a very detailed map of every city and village in the country to make sure that we can find our way to every single house. However if we look to travel from one city to another, such a detailed map is of little use, since it contains a great amount of information that does not concern any path to another city, thus making the map unnecessarily unwieldy and hard to use. If however we create another map that only contains information about the larger roads and leaves out many details like houses small villages, it will become much simpler to navigate even large distances.

3.5.1 General Topology

As explained above, the general topology information describes the system as a whole in limited detail.

The most basic part the general topology is the layout of the communication busses. This is done below the `busses` field in the topology file and supply the information needed to interact with that specific interface. A bus is defined as a name with corresponding `bus_number` and `alerts` fields. My current implementation does not use the `alerts` field

and uses the `bus_number` to select an I2C device under `/dev`. This functionality is not necessarily complete and may need to be extend, if communication protocols other than I2C are needed.

Of course the topology must also include information about what devices are available on the system. This is captured as a list of devices named `devices` field in the topology file. Every device that is entered into that list specifies some information about itself, some of which is type dependent. First of all, the device must in all cases specify its exact type identifier in the `type_name` field. This lets the compiler know how to interact with the device if it is required to perform some action. Also in all cases, the device must specify the bus, to which it is connected, in the `bus_name` field. This lets the compiler know where to interact with the device. Other than that, there are some interface specific settings that need to be set below the `interface_settings` fields in order to ensure that the compiler can produce interactions with the device. Currently, the only handled interface is an I2C bus, and thus the only setting that really is needed is the `address` setting, which, in combination with the `type_name` and `bus_name` fields, lets the compiler know exactly what code to produce for any interaction with the device. Of course, if other bus types are introduced to the system, such as for example some GPIO based interface, different settings will be necessary to be passed in `interface_settings`, such as the relevant pin numbers.

For actually interacting with the devices, the compiler needs to know about the controls that are available on the system. These available controls are listed in the `control_map` field. Every control is specified by its name, the name of a device in the `device` field and a device specific control name in the `control` field. This is basically mapping the device specific control to a globally unique name, hence the name `control_map`.

The monitoring commands are also aliased in a similar manner. However, instead of the `device` and `control` fields, the monitor mappings include a list of key value pairs, where the key is the name of a device, and the value is a list of device specific monitor names. This should reflect the asymmetry we can observe in control and monitoring systems: only one source can drive some rail at a time (conceptually speaking, of course the load could be distributed among multiple devices, but these would work together as a single unit and not target different voltages), while multiple monitors from multiple devices can monitor any value at any given time. These monitor mappings exist for any monitor that could be encountered, namely temperature monitors in `temp_monitor_map`, `voltage_monitor_map` and `fanspeed_monitor_map` in the current state of my implementation. Naturally, if the need arises, this can be easily extended to contain more possible monitors and although the current three different monitor classes behave identically, the separation should facilitate any possible future extensions that need to behave differently for some reason.

Additionally, the topology file contains a field `monitor_scale` that captures constant scaling factors for some voltage regulators. I have included this from the enzian power management repository because some voltage regulators on enzian are managing a voltage outside their specification and achieve this by passing the voltage through a resistor voltage divider and monitoring this reduced voltage[3]. In general, this can simplify the system because it reduces the need for special voltage regulators for higher voltage rails and thus is a useful addition to the topology information.

3.5.2 Devicetype specific information

The devicetype specific information is meant to capture all information that describes the interactions with a specific device. The assumption is, that all instances of the same device behave identically, down to some state that can be set completely over its communication interface.

The device file specifies some values that are shared among all possible device types, such as `device_type_name` and `device_interface` but also some interface or even device specific information, such as a list of PMBus formats for a PMBus capable device.

This file may also contain a list of overrides for certain controls and/or monitors. Such an override is abstracted as a list of other commands and serves to enable non standard compliant devices (like the MAX31785) to be controlled by the generated sequence. This problem with non compliant devices will be explored further at a later point.

3.6 Compiling the power sequence

Once the input has been prepared and converted into the format detailed above, the compiler can actually start its work and produce a action sequence. This still is not the definitive output of the backend, but rather another intermediate representation of the solution. This time however much more concrete and including all the information necessary to execute said sequence. I have encoded this stage of the solution into a list of `driver_action` that describe the interactions over various communication interfaces and also encode the monitoring of values and non standard behavior. The most general type, this `driver_action`, looks as follows:

```
1  type param =
2    | InternalVariable of string
3    | Variable of string
4    | Int of int
5    | Int64 of int64
6    | ParamList of param list
7    | Literal of string
8    | Empty
9
10 type driver_action =
11   | I2C_Action of param * i2c_action * param * param
12   | SMBus_Action of param * smbus_action * param * param
13   | PMBus_Action of param * pmbus_action * param * param
14   | Custom_Action of driver_action list
15   | Monitor_Action of
16     param * param * param * param * driver_action list * param * param
```

Listing 3.3: Type definitions of driver actions and parameters.

`I2C_Action`, `SMBus_Action` and `PMBus_Action` are the encodings of interactions through their respective protocols and include some meta information as instances of `param`. In order to facilitate this explanation, lets look at the following example instance of a `driver_action`:

```
I2C_Action (bus_name, action, return_value, return_length)
```

The `bus_name` parameter encodes the name that the bus object, that should be used to transmit the commands over the wire, has in the C code and is in general identical to the name that is specified in the `busses` field in the topology file.

The `return_value` and `return_length` parameters are used to indicate a return value of the command. If the action does not return a value, such as a simple write, then these parameters would be `Empty`, if however there is a return value, then `return_value` describes the name of the return value, and `return_length` describes the length of the returned value in bytes. The length is necessary, conceptually speaking, since some actions may return a block of data of unknown length, and be stored as a pointer to some structure in memory. Currently, this is not the case in my implementation however, since the fan controller does not expose any such commands

The `action` field in the above example must be of type `i2c_action`, according to the definition of `driver_action`, however, since the `SMBus_Action` and `PMBus_Action` constructors have the same signature, I will describe all of these here.

```
1 type i2c_action =
2   | I2C_Write of { address: param; values: param; }
3   | I2C_Read of
4     { address: param; return_value: param; return_length: param }
5   | I2C_Write_And_Read of
6     { address: param; values: param; return_value: param;
7     return_length: param }
```

Listing 3.4: Type definition of the `i2c_action`.

The `i2c_action` encodes any transactions over a standard I2C interface, without any protocols on top. The definition above, includes a simple write, a simple read and a more complex write-and-read transaction with no stop-condition between writing and reading. This is not necessarily complete yet for any possible case, but these are the types of transactions that are required for the protocol stack implemented.

Parameters like the target address or the data to be transmitted are encoded in constructor arguments such as `address` and `values` of type `param`.

```
type smbus_action =
  | SMBUS_QuickCommand of { address: param; value: param }
  | SMBUS_SendByte of { address: param; data: param }
  | SMBUS_ReceiveByte of { address: param; return_value: param }
  | SMBUS_WriteByte of { address: param; command: param; data: param }
  | SMBUS_WriteWord of { address: param; command: param; data: param }
  | SMBUS_WriteBlock of
    { address: param; command: param; data: param }
  | SMBUS_ReadByte of
    { address: param; command: param; return_value: param }
  | SMBUS_ReadWord of
    { address: param; command: param; return_value: param }
  | SMBUS_ReadBlock of
    { address: param; command: param; return_value: param;
    return_length: param }
  | SMBUS_ProcessCall of
    { address: param; command: param; data: param;
    return_value: param }
  | SMBUS_ProcessCallBlock of
    { address: param; command: param; data: param;
    return_value: param; return_length: param }
```

Listing 3.5: Type definition of the `smbus_action`.

The `smbus_action` is a little less brief than the `i2c_action` and encodes all possible (standard) SMBus commands. In contrast to the `I2C_Action`, this implementation is largely complete (except for the Packet Error Code (PEC) capability listed in the SMBus specification, which is not used in my case) and allows the implementation encoding of arbitrary SMBus transactions.

Again the parameters needed for executing the transactions are encoded using the constructor arguments of the type definitions.

```
type pmbus_action =
  | PMBUS_PAGE of
    { format: pmbus_format; address: param; page: param option;
    return_value: param option }
  | PMBUS_CLEAR_FAULTS of
    { format: pmbus_format; address: param; }
  | PMBUS_WRITE_PROTECT of
    { format: pmbus_format; address: param; mode: param option;
    return_value: param option }
  | PMBUS_STORE_DEFAULT_ALL of
    { format: pmbus_format; address: param; }
  | ...
```

Listing 3.6: Type definition of the `pmbus_action`.

In contrast to the other action types, the PMBus commands are only partially listed here, since the PMBus standard names over 200 commands (even though I have only implemented 42 of them so far). This still illustrates the concept behind the `pmbus_action` type. Similar

to the `i2c_action` and `smbus_action` types, there are some parameters that capture any information necessary to execute the action.

Unlike most other commands, the `format` parameter is not of type `param`, but of type `pmbus_format`. This is, because the format needs more information than can be encoded into a simple `param` type variable, since the format information includes some function calls with different signatures depending on the format used and the direction of data transfer. This distinction allows the code generator to directly translate a format into C code, much like the rest of the parameters that don't need any additional information.

Additionally, the value carrying parameters and the return values are marked as optional here. Since PMBus commands are generally readable and writeable, this flag allows the same constructor of `pmbus_action` to encode both a read and a write and is encoded into a mapping that is passed to the code generator (this will be explained more detailed in section 3.7).

The `Monitor_action` includes a lot of meta information as well. Again, lets look at a concrete example with named parameters.

```
Monitor_Action (min, max, delay, timeout, actions, return_value, return_length)
```

The parameters `min`, `max`, `delay` and `timeout` are simple integers, with `[min, max]` denoting the interval constraint that the monitored value must satisfy, `delay` denotes a delay before the value should be probed the first time and `timeout` denotes the timeout to keep retrying before aborting.

Again the parameters `return_value` and `return_length` denote the value that is returned by the actions and its length, and their function is the same as above.

The `actions` parameter describes a list of `driver_action`s that read some value that should be monitored. This list usually only needs to contain a single action that reads a value from a device, however, in some cases there may be a more complicated list of commands. This may be the case if the command may leave traces that need to be cleaned up before continuing, or if the command has been approximated by more than one action.

Such non standard commands are encoded by the `Custom_Action` constructor and are used to implement non standard compliant commands and controls of devices that otherwise mostly support some standardized interface or protocol and thus do not warrant the development of their own set of `driver_action`s. This concept works very well on the protocol stack that I am handling. For example any PMBus capable device may ignore some part of the PMBus standard and the compiler could fall back to using `SMBus_Action`s or even `I2C_Action`s to implement the behavior of said device instead. The obvious shortcoming of this approach is the possible lack of lower layers in the protocol stack. This problem will be explored in more detail in section 3.8

Now that we have had a look at the inputs and the datastructures the compiler handles, we can take a deeper dive into the compilation process itself. The compiler has a single entry-point that directly converts the inputs into the target output, which is done by a

single short function named `compile`. This `compile` function itself is very brief and simple to explain, but there are some interesting facts hidden in its signature.

```
1 let compile (t: topology) (s: power_sequence): driver_action list =
2   let PowerSequence power_cmds = s in
3   List.map (compile_power_command t) power_cmds |> List.concat
```

Listing 3.7: The implementation of the compile function.

Note that `compile` function is little more than the `List.map` function exposed by OCaml's standard library. This function applies a function `f` to every element of a list and returns the list of result, so the call `List.map f [a; b; c]` would result in the list `[f a; f b; f c]`[11]. Due to the functional nature of OCaml, the different calls to `f` do not influence each other and thus every command is compiled completely independent of any other command in the power sequence.

As explained before, the compiler takes additional information, to compile the input sequence, in form of `t` of type `topology`. `t` describes the system that the `power_sequence` passed in `s` should run on and includes all the information listed above. The sequence `s` itself carries a list of power commands and thus can simply be translated element by element using the `List.map` and `compile_power_command` functions.

The `compile_power_command` function really does the heavy lifting for the compiler. This function decides for every `power_command` independently whether it requires a read or a write `driver_action`, collects all necessary information about the communication interface and assembles the required list of `driver_action`s containing all information the code generator needs. The pseudo-ocaml code below illustrates the operation of this function.

```
let compile_power_command (topo: topology) (cmd: power_command) =
  match cmd with
    | Monitor ->
      get device_instance
      match device_instance with
        | PMBus_Device ->
          get necessary parameters
          if this is an overridden command
            expand the overrides variables
              (this returns a driver_action list)
            wrap the override in a Monitor_Action,
            prefix a PMBUS_PAGE command and return
          else
            construct PMBus_Action, wrap it in Monitor_Action,
            prefix PMBUS_PAGE command and return
        | Analogous for other device types
    | Write ->
      get device_instance
      match device instance with
        | PMBus_Device ->
        get necessary parameters
        if this is an overridden command
          expand the overrides variables
            (this returns a driver_action list)
          wrap the override in a PMBus_Action, prefix a
          PMBUS_PAGE command and return
        else
          construct PMBus_Action, wrap it in PMBus_Action,
          prefix PMBUS_PAGE command and return
        | Analogous for other device types
```

Listing 3.8: A simplified representation of the `compile_power_command` function.

Many lines in the above function are simplified or even compressed into less lines for brevity in the sample above. The concept behind this is that every `power_command`, whether write or monitor, carries its target and and the parameters for the action it should accomplish. The function then consults the topology information to find the targets devicetype and decides to communication interface to use subsequently. Once the interface has been selected, the command is checked for possible overrides, that could alter the function of said command. If the command is indeed overridden, then the override template is loaded, expanded with the the parameters of the command and packaged into the monitor or write action that will then be returned. If the command is not overridden, then the command template will be given the parameters, packaged as well and then returned.

3.7 Generating C code

As section 3.2 stated, my compiler should output C code as its target. The program that is produced should be complete, in the sense that can be compiled into a native executable

with only the necessary libraries linked to it by the use of a C compiler such as gcc or clang. This means that the output and the resources it depends upon need to contain all the functionality necessary to perform any action that could be demanded by a power sequence.

Once the `driver_action` list has been produced as explained in the sections above, the `driver_action` list is passed to the code generator of the compiler. As the name implies, the task of te code generator is to generate code, C code in this case. Usually the code generator is one of the last steps of the compilation before an executable is produced and produces the machine code fora specified target platform[2]. As I have already explained however, my compiler should not directly produce an executable, but C code instead. As section 3.2.1 has already mentioned, this is a convenient way of increasing the modularity of the compiler since it completely avoids any dependencies on the target platforms architecture and can move this into a separately supplied runtime library and greatly reduces the complexity of the compiler itself. The produced C code can be processed further by any C compiler that already can target any platform one could wish for.

3.7.1 The template approach

Since the compiler really does mostly just work with power sequences in various forms of abstraction without any intent of optimizing the resulting C code, the code generator does not really need a good understanding of what C actually is. By making use of a template driven approach, I could keep the complexity of the code generator quite low and in turn offer myself a high degree of control over the produced C code with little to no modification of the code generator itself.

To produce C code from the abstract representation of the sequence, I wrote a series of templates that already contain the required C code and some flagged variables that can be replaced with the actual necessary values. The big advantage of this approach is the flexibility of the compiler regarding its target. Since the entire structure of each and every `driver_action` is basically an invariant, this information can be brought out of the running time of the compiler completely and produced beforehand. This also allows the compiler to be very quickly and easily adjusted to a different target, without requiring any modification to the source of the compiler itself.

Below, the template for `I2C_Write_And_Read` is depicted to show the workings of this template system. The name of variables are wrapped in angle brackets and are replaced by string representations of the required values by the code generator. The resulting strings that are generated for all `driver_action`s are then concatenated and themselves written into a larger template for building the entire C program including any include directives and helper functions.

```
i2c_write_and_read:
    |
        uint8_t* <write_data_name> =
            pack_values_into_array(<write_length>, <write_data>);
        uint8_t* <read_data_name> = create_data_array();
        runtime_i2c_write_and_read_bytes(<bus_instance>, <address>,
            <write_length>, <write_data_name>, <address>,
            <read_length>, <read_data_name>);
```

Listing 3.9: A simple template showing the variables that are replaced by the code generator, as well as its representation as a multiline YAML string.

Recall that most of these values are already defined by the corresponding `driver_action`, as is illustrated in this instance by the listing below:

```
I2C_Write_And_Read of { address; values; return_value; return_length }
```

Listing 3.10: Simplified definition of the `I2C_Write_And_Read i2c_action`.

Fields like `<address>` only need to be replaced, without any further processing, while fields like `<write_data_name>` or `<write_length>` are not precisely known at this time in the compiler, so they have to be generated. Meta-information of known parameters, such as `<write_length>`, can easily be computed at this time however and temporary variable names can simply be generated by the compiler, even guaranteeing no duplicate generated variable names.

As is evident from this description, this method of code generation is rather simple to implement and provides some degree of functionality that is satisfactory for what I've defined earlier.

3.7.2 The runtime library

Similar to many other systems, the code that my compiler produces does have some dependencies. These include very basic items, such as the OS and the C standard library, but it also needs more concrete functionality specific to the C programs that represent a power sequence. Such specific functionality includes all the interfaces that are exposed by any communication standard the compiler can handle, such as, in my case, the I2C, SMBus and PMBus commands and is packaged into a runtime library. I have chosen to bundle the communication interfaces and some helper functions into a runtime library for two reasons.

Firstly, the approach I have taken with the non-standard compliant commands depends on some layer hierarchy in the interface, as I will explain in a later section, and thus, I can make use of this dependency here as well. A layered interface in general, such as in my case the I2c, SMBus and PMBus protocol stack, is much simpler to implement if the entire interface of the lower levels is readily accessible. Since I had to implement the SMBus and PMBus interfaces myself, as no readily accessible implementation seems to exist, I would not have to write the entire protocol stack into every single template, but rather

could implement it once, and call it from the templates, which in turn greatly reduces their complexity.

Secondly, a runtime library provides a modular piece of functionality that is very loosely coupled with any code that runs on top of it, with only the interface between the layers being a fixed point. For this reason, it is very easy to exchange a runtime library with either an updated version of the runtime library, or even another runtime library, providing different functionality through the same interface. This can be used to facilitate development and debugging of the rest of the compiler, independent of the target language and platform.

The templates above have shown calls to functions such as `pack_values_into_array` and `create_data_array`, in addition to the communication stack, which is in this case represented by `runtime_i2c_write_and_read_bytes`. These two functions are some of the helper functions I have mentioned earlier. In the case of `pack_values_into_array`, I need to provide an array of unknown length to the I2C layer, which is not possible using literals in C. As C provides an interface to handle variable argument, so-called varargs, this problem can be easily solved by a simple inlined function.

The protocol stack itself consist of an interface to the I2C driver, provided by the linux system below, at the lowest levels. It enables the two most basic I2C transactions, a simple read and a simple write of arbitrary data to the I2C bus, as well as a two-transfer write and read transaction, that is needed by the SMbus implementation that depends on it.

```
1  void runtime_i2c_write_bytes(struct bus_instance* bus_instance,
2      uint8_t address, uint16_t length, uint8_t* data)
3  {
4      PLD("I2C: Writing Data: 0x%02x, Max Length: %d bytes\n",
5          address, length);
6
7      struct i2c_msg i2c_msg0;
8      i2c_msg0.addr = address;
9      i2c_msg0.len = length;
10     i2c_msg0.flags = 0;
11     i2c_msg0.buf = data;
12
13     struct i2c_rdwr_ioctl_data ioctl_data;
14     ioctl_data.nmsgs = 1;
15     ioctl_data.msgs = &i2c_msg0;
16
17
18     IOCTL(((struct i2c_bus_instance*)bus_instance)->i2c_adapter_fd,
19         I2C_RDWR, &ioctl_data);
20 }
```

Listing 3.11: An example of the I2C related functions in the runtime library, specifically the `runtime_i2c_write_bytes` function for transmitting arbitrary bytes.

The SMBus and PMBus protocols respectively build on top of the infrastructure that is already provided by lower layers. Since there does not seem to be a viable implementation already in existence, provided either by the linux system or a third party, I had to resort to

implementing the SMBus and PMBus protocols myself. This was not a big issue thankfully however, since both protocols are rather straight forward, when it comes to the data transfer itself. The examples below illustrate how the stack is built and the different layers interact with one another.

```
1  void smbus_write_byte(struct bus_instance* bus_instance,
2      uint8_t address, uint8_t command, uint8_t data)
3  {
4      PLD("SMBUS: Writing '%d', '%d' at '0x%02x'.\n",
5          command, data, address);
6      uint8_t buf[2];
7      buf[0] = command;
8      buf[1] = data;
9      runtime_i2c_write_bytes(bus_instance, address, 2, buf);
10 }
```

Listing 3.12: An example of an SMBus function in the runtime library, specifically the `smbus_write_byte` function, used for sending a single byte as the parameter to a command byte over the SMBus protocol.

```
1  #define PMBUS_COMMAND_PAGE                    0x00
2
3  void pmbus_page_write(struct bus_instance* bus_instance,
4      uint8_t address, uint8_t page)
5  {
6      PLD("PMBUS: Writing: ADDRESS: %d, PAGE: %d\n",
7          address, page);
8      smbus_write_byte(bus_instance, address,
9          PMBUS_COMMAND_PAGE, page);
10 }
```

Listing 3.13: An example of a PMBus function in the runtime library, more specifically the `pmbus_page_write` function. Note the suffix `_write` for this function, which is a result of pmbus commands being bidirectional.

In addition to the above, I have also implemented a debugging version of this runtime library. As I have mentioned before, the runtime library can easily be swapped out and replaced by another one, as long as the interface matches. The debugging version that I have written actually developed as a side effect of a solution to another problem: Since the compiled power sequences are intended to run on an ARM based BMC (as the Enzian platform currently uses this), I was not able to test the functionality of any sequence on my local machine, which is an x86-64 based computer. To facilitate especially the early stages of testing, I decided to do something about that and implemented two `ifdef` wrapped preprocessor macros in the runtime library, called `PLD` (short for PrintLineDebugger) and `IOCTL` (a wrapper for standard library functions of the same name), which can already be seen in the examples provided above. The `IOCTL` macro excludes the code interacting with the linux I2C interface when the library is built for an x86-64 target and the `PLD` macro includes a printf statement if the library is built for an x64 target, while excluding it when building for the ARM target. The `PLD` macro was a great tool to test early outputs of the

code generator, since printing the values at multiple times through the protocol stack, up until it would have been sent over I2C, helped debug the first implementations of functions like the PMBus format conversions.

The attentive reader may have spotted, that this debugging version of my runtime does not actually leverage the fact that the runtime can easily be exchanged, but rather uses the very powerful preprocessor of C to include debugging code where needed and remove target specific code where appropriate. Of course this is correct, and in fact this is probably even the simpler way of doing it in my situation, however I still wanted to include this second advantage of runtime libraries, since going forward, the compiler does not necessarily need to compile to a target that offers the same tools that the C environment provides. Of course, when using C as the target, there still could be use of this functionality, if one were to dynamically link the runtime to the power sequence, although the power sequence would still need to be compiled once for every target architecture at least.

3.8 The suboptimal reality of non standard compliant devices

The attentive reader may again have spotted, that I have left the topic of non standard compliant commands largely untouched until now. I felt this a sensible thing to do, since the entire compiler works and can be explained well and reasonably representative, without taking standard violations into account. After all, the point of standardization is usually to avoid having to seek solutions for strange or unexpected behaviour and incompatibilities. Standardization usually loosens code coupling, improves modularity and simplifies development for accessories, but despite this, even with these clear advantages, such non compliances are apparently not avoidable in the context of power management.

To effectively enable the handling of such a command, I have made some assumptions during development that even these violating commands need to follow, namely:

1. The device in question does not violate a standard on the hardware level (e.g. it may not violate the physical layer of the I2C standard, since the I2C master device may not be physically able to communicate with it in that case), or the lowest software interface available to the developer.
2. The device in question is capable of using a communications stack of some kind, with lower levels able to fill in possible gaps in the higher levels.

The first assumption arises from the simple fact that on some level, there must be an interface to move data from one device to the other. This of course includes the electrical and physical compatibility of the involved devices, but it also includes the lowest available software interface as a less hard dependency (albeit a less hard one, since it may be possible to develop a special driver for that case). Obviously, failure to comply with even this basic a standard will render the device unusable and therefore this is likely not a common problem.

The second assumption is technically a limit imposed by my model of the input, since it would be possible to simply provide the source code of the lowest level to the compiler instead of declaratively specifying an implementation using other interfaces already known to the compiler. This would however make specifying the topology more complicated and error prone as well hinder the topology's ability to be reused on a slightly different target platform, since source code compatibility may not be guaranteed between platforms, especially for low level languages.

As an example lets take a look at the `MFR_ID` command exposed by the fan controller. According to the PMBus specification, this would translate into an SMBus Block read/write, but as illustrated in section 3.1 in figure 3.1 the device expects an SMBus read/write byte transaction in its place. Of course the compiler needs to handle this case, since a block transaction would likely disrupt the operation of the device.

Conceptually speaking, my solution consists of overriding the offending command with a `driver_actions` constructed by `Custom_Action`. Such a `Custom_Action` is defined by a list of normal `driver_actions` and can be defined in the topology with some fixed values and some variables that will be expanded during compilation.

```
1 MFR_ID:
2     read:
3         i2c_write_and_read:
4         <bus_name>: <bus_name>
5         <address>: <address>
6         <write_length>: 1
7         <write_value>: 0x99
8         <return_length>: 1
9         <return_value>: <return_value>
```

Listing 3.14: This listing depicts the override of a non standard compliant command, in particular the `MFR_ID` command exposed by the MAX31785

Listing 3.14 depicts the command override for the `MFR_ID` command that is exposed by the fan controller and defines an abstract solution to this problem which can be implemented by the compiler. As the listing shows, an override defines which lower level command should override standard implementation and how to define the fields it requires.

If the command were not overridden, then a read from `MFR_ID` would internally be represented by the following instance of the `driver_actions` type (the monitor action has been ignored in order to simplify the example):

```
1 PMBus_Action (bus_name, PMBUS_MFR_ID (address), retval, retlen)
```

Listing 3.15: Depicts the constructor of `driver_action` encoding a PMBus action.

If however the command were overridden according to listing 3.15, the very same read from `MFR_ID` would be represented by the following, different `driver_actions` instance (again, the monitor action is being ignored):

```
1 Custom_Action ([I2C_Action (bus_name, SMBUS_READ_BYTE (address, cmd),
2   retval, retlen)])
```

Listing 3.16: Depicts the constructor of `driver_action` encoding a custom action.

Needless to say, the latter would result in vastly different C code. In contrast to the first one however, the resulting C code of the latter variant would actually result in the desired behaviour by replacing the PMBus command that is being violated with the SMBus transaction that is given in the datasheet in section 3.1.

More severe cases of non compliance may result in deeper reaching overrides, such as the example of the `MFR_LOCATION` command demonstrates. The `MFR_LOCATION` command breaks the SMBus standard and cannot be reimplement using SMBus transactions. The result is that the compiler has to fall back to I2C to reimplement the functionality that the device expects for this command. The representation would look similar, with an I2C transaction taking the place of the SMBus transaction in the example above.

This approach clearly has its limits however, since there are no protocols to fall back on, when the lowest protocol in the stack is violated, such as for example the I2C protocol, although this example seems highly unlikely since non I2C compliant devices likely could not communicate with an I2C controller.

Chapter 4

Evaluation

I have evaluated my work through some tests, the results of which I will present in this chapter. In general however, the verification of a compiler is not a trivial task and the fact that every stage of the compiler chain described earlier is either not yet implemented or itself not yet verified to be correct does not help the matter.

I have not been able to formally verify any of my work, however, I have been able to create a set of test cases, that demonstrate the feature set of my compiler and can demonstrate the feasibility of such an approach. I would like to stress however, that a formal verification of the compiler chain should be considered, should it ever be fully completed and used for the deployment of hardware, since there are numerous safety and security concerns, that have been mentioned earlier, that may not be caught by test cases.

4.1 Topology

First of all, I want to mention that an integral part of the compilation process is reading, parsing and interaction with the topology information given to the compiler. Trivially, the compiler cannot function correctly without or with incomplete or corrupted topology information, which is why it is important to to test the input and to verify that the topology information is correctly read and interpreted. In addition to the correct working of the topology parsing, this test will ensure that the subsequent tests can reliably determine whether or not the compiler works as intended.

In order to evaluate the correctness of the topology parsing, I have created a handful of functions for the compiler, as well as the '–print-topology' input flag, which allows the compiler to simply read and interpret the topology information and printing it again, the way it is represented in memory. To this end, I have created a series of small topology files that can be manually inspected and compared with the output of the compiler to determine whether or not the interpretation of the topology done by the compiler was correct.

The output that the test produces is a textual representation of the topology information (both the general topology and the device specific information) that has not been translated back to the original input. It illustrates the exact structure that the information has taken in memory and makes the duplication of some information, as well as the general similarity between input and output visible. The output meets my expectations and represents the topology used as an input. As an example for the outputs in this category, I have included a simple test topology and it's output in appendix A.

This result is the desired one and no surprise to me, given that this test was mostly a sanity check. Never the less, this demonstrates that the compiler can correctly read and interpret the topology information and is an important milestone.

Now that this is out of the way, I can start to actually test the compilation of power sequences and demonstrate the abilities of the compiler in the following sections.

4.2 Simple Sequences

In order to demonstrate the functionality and feasibility of the entire compiler backend, it was a good place to start with the basics. Recall that the `compile` function is little more than a call to OCaml's `List.map` function on a list of `power_command`s (as depicted in listing 3.7) and as a result of this simple implementation, the compilation is done on a command by command basis and, as per the functional nature of OCaml, different commands are independent of one another as was explained in section 3.6. For this reason, testing single commands is a viable strategy to find errors in the compilation as it does both approximate the real use of the compiler, but also keeps the complexity of the test cases very low and thus is easy to work with. Accordingly, these tests will attempt to show the correctness of the compilation process.

The sequences are limited to a single, standard compliant and non overridden command of the fan controller. Both write and monitor commands have been tested by writing and monitoring multiple different commands, described in a simplified topology. The generated C code file is then visually inspected, further compiled for the BMC using gcc, for both the x86 and ARM targets. Recall that the x86 target includes some debugging utilities and does not actually interact with any hardware. The traces it provides depict the calls that were made to the standard library greatly facilitate manual evaluation. Finally the compiled executable for the ARM target is run on actual hardware and the result observed and evaluated.

```
-
    type: monitor
    device: fans
    input: CPU_FAN
    delay: 100
    range:
        from: 0x0000
        to: 0xffff
    timeout: 1000
```

Listing 4.1: Shows an input sequence describing a single simple PMBus `READ_FANSPEED_1` command to be executed.

Listing 4.1 depicts the typical input these tests provided and shall serve as the sample for all the tests run in this category. I have also reduced the topology used by this test in order to facilitate the manual examination of the output. The topology consists of a single device with only one control and two monitors and was reused for all tests in this category. The full topology that was used for these tests can be viewed in appendix B. For the remainder of this section, any further listings of outputs, traces and concrete examples of results will be produced by compiling the sequence of listing 4.1.

Listing 4.2 depicts the `power_sequence` function of the output produced by the compiler when compiling the input sequence described above. While the Listing below only depicts a single function, you can inspect the full output, as well as the corresponding trace in appendix C.

```
1  void power_sequence()
2  {
3      pmbus_page_write(pwr_fan, 82, 0);
4      int __mvar1;
5      runtime_sleep_ms(0 - 10);
6      struct timespec __start2;
7      clock_gettime(CLOCK_REALTIME, &__start2);
8      do {
9          runtime_sleep_ms(10);
10         uint16_t rv = pmbus_direct_format_inverse(
11             pmbus_read_fan_speed_1_read(pwr_fan, 82), 1, 0, 0);
12
13         __mvar1 = rv;
14         struct timespec current;
15         clock_gettime(CLOCK_REALTIME, &current);
16         struct timespec diff =  {
17             current.tv_sec - __start2.tv_sec,
18             current.tv_nsec - __start2.tv_nsec
19         };
20         long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
21         if (diff_long > 100) runtime_timeout();
22     } while (!(__mvar1 >= 0 && __mvar1 <= 65535));
23 }
```

Listing 4.2: This listing shows single function of the output of the compiler when given listing 4.1 asinput. This does not include any memory setup or cleanup performed nor the actual entry point function. A more complete output of the compiler can be seen in the appendix.

The code produced by the compiler for each of these sequences looks exactly as expected. Both the write and monitor commands follow the expected template and contain the correct expanded values for addresses and data, where applicable. Running the sequences on both x86 and ARM targets works as intended and does not show any obvious shortcomings and/or sever bugs. The traces that are produced by the x86 executable show that the correct calls are made and the protocol stack handles them as expected as depicted in appendix C. The trace depicts many reads, which indicate that the loop has run many times. This is an expected consequence of the debug version of the runtime explained earlier in section 3.7.2. Since the debug version does not interact with any hardware, it cannot return any data and as a solution generates a random response to fill in the absence of a device. The ARM targeted version does perform as expected as well and triggered the expected fanspeed changes as well was awaited correct fanspeed before terminating.

4.3 Complex Sequences

These sequences are intended to be larger than the single commands demonstrated in the previous section and could possibly even be integrated in a larger system with a manager program that runs the sequences. In order for this functionality to work, the sequences must still be correct when they are much longer than one command, and their correctness

is not only limited to the correct translation of single commands and adherence to the template, but also includes more interesting information such as the naming of temporary variables and memory management. The naming of temporary variables works by adding a number as a suffix to the variable name in order to make it unique, and must of course work through any sequence to avoid compilation problems and correct functioning of the sequence. The memory management is important, because the sequences must work correctly and not produce any errors, even for long lasting uptimes of the BMC. The following should demonstrate that the problems mentioned above do in fact not appear in the output of my compiler.

The sequences of this category consist of a collection of write and monitor commands that perform some setup, in order to move the fan controller to some desired state, and some write and monitor commands that set and monitor some values, in order to check the correctness of the preceding setup actions. In addition, all sequences of this category start with a soft reset to ensure equal starting conditions and prevent configurations from persisting through multiple tests. The commands do not however contain overridden commands, neither non standard compliant nor manufacturer defined, since these are handled in their own category. When a sequence is produced, it is again first manually inspected for obvious errors, before then being compiled for both the x86 and ARM targets, where the x86 executable will again output a trace of driver calls and the ARM executable will again be run on the BMC. Since the sequences of this chapter are rather long, I have included some samples of them in appendix D.

The manual inspection of the produced C code has turned out to be more difficult in this situation, since the amount of code produced is greatly increased over the last category of tests. Despite this, I have had a close look at the output and it seems to follow the template and the individual power commands are still translated as expected. The names of the temporary variables, generated by the compiler, are unique to their variable and references to variables seem to be resolved correctly, using the expected variable name to refer to some value as written in the template. Furthermore, the output allocates memory in two different ways: on the stack using alloca and on the heap using malloc. Stack allocation is used widely in the `power_sequence` function, whenever space needs to be allocated for a temporary variable, such as the data to be transmitted, or the values returned, in case they are not single bytes or words. Heap allocation on the other hand is used to allocate space for the structures that describe the busses present in the system. These allocations are made before the `power_sequence` function executes and are freed again thereafter.

The compilation of the code to both the x86 and ARM targets have worked without any errors. The trace produced by the x86 executable meets the expectations and shows that the correct data was being transmitted in the correct order. A sample of the produced traces and outputs can be found in the appendix D. When run on the BMC, the sequences produce the expected effects, that range from expected runtime of the sequence executable to some observable effect like the fans speeding up or slowing down.

Obviously, the fact that the code follows the template, even for larger sequences, is good and even necessary for the compiler to work correctly. However, based on the results from the simple test category, I was expecting this here as well. The same can be said for the correct translation of the individual commands. However, the variable naming and the memory management still provide valuable information about the workings of the compiler.

First of all, the uniqueness of the variable names produced is of course a necessity for the compiler to function correctly, not to mention the successful compilation of the produced C code. Since the variable names are unique over a sequence of multiple commands, the information needed to generate a unique name does get carried over to the following command and thus should not fail for any sequence thrown at it, and in addition it shows that the variables are expanded correctly.

The memory management of the produced sequences is of course another very important point to examine. Since the BMC's uptime could potentially be arbitrarily large, any power manager that aim to make use of such sequences not only needs to work correctly, but also needs to work for extended periods of time. As a result, slow failures such as memory leaks that would only show after weeks of use must also be out of the question. As stated above, the entirety of the `power_sequence` function allocates the memory for its temporary variables on the stack. This has several advantages with the most dominant being the cleanup needed to mitigate memory leaks. For stack allocations, this cleanup is reduced to none at all, since the stackframe is completely cleared once the function returns and the code necessary for this is automatically produced by the compiler. In addition to the cleanup necessary (or rather the lack thereof), stack allocations also have a speed and predictability advantage over heap allocations, since the malloc function must first find a suitable block of memory to allocate and must also from time to time coalesce the blocks it has already allocated and freed again, in order to prevent memory fragmentation. This not only is additional and more complex work, but it may also have a highly varying runtime, since the coalescing may run at any time. The heap allocations that happen none the less, all happen in the setup function and allocate space for a part of the topology information. This information is long-lived, since it needs to persist through the entire sequence, and may even be reused for other sequences, since the topology for any sequence on the same system should stay the same.

4.4 Non Standard Compliant Commands

While the correct compilation of standard compliant power sequences is of course a very important foundation of any compiler for power sequences, it is also necessary that non standard behaviour can be modeled and handled, as discussed in section 3.8. However, since the problems faced when handling non standard behaviour differ quite substantially from the problems faced when compiling standard compliant sequences, it was justified to take a deeper look at these in their own section. Since no other test has compiled a non standard command, these tests are the first ones to make use of the necessary features of the compiler and thus aim to demonstrate their functionality and correctness.

I have run 4 test for different commands that had to be overridden, either because they break the standard, or because they are manufacturer defined. The commands I tested were the `MFR_ID`, `MFR_LOCATION`, `MFR_FAN_CONFIG` and `MFR_FAN_LUT`. Similar to the earlier tests with power sequences, I have compiled the sequences, manually inspected the code, compiled it for x86 and ARM and then run it onn x86 and ARM. The input sequences again were a single command, which made use of the override feature of the compiler. As discussed in section 3.1 the `MFR_ID` command did only violate the PMBus standard and could

be reimplemented using the SMBus interface, however the `MFR_LOCATION` command breaks even the SMBus standard with its violating block format. The `MFR_FAN_CONFIG` command is a manufacturer specified command and does not break the PMBus or SMBus standard. Lastly, the `MFR_FAN_LUT` command not only is a manufacturer specified command, but it also breaks the SMBus standard, again by a non compliant format that lacks the length of the data at the beginning. Since the input sequences are very similar to the input sequences in section 4.3 above, I have not included the input here, however, if you wish to have a look at the inputs, the are listed in appendix E.

In all cases, the manual inspection of the code turned out with no error again. Before continuing and compiling the resulting C code, it had to be altered slightly, so as to output some references when running a monitor command on the BMC, or if the command returned a block. After the modifications illustrated by listing 4.3, the compilation using gcc went without any issue and the x86 executable ran without any issues and its trace meeting expectations. The modification were necessary, since the command returns a pointer to the return value in this case.

```
1  do {
2      runtime_sleep_ms(10);
3      uint8_t* __i2c_temp_data5 = pack_values_into_array(1, 153);
4      uint8_t* __custom_return_value3 = create_data_array();
5      runtime_i2c_write_and_read_bytes(pwr_fan, 82, 1, __i2c_temp_data5,
6              82, 1, __custom_return_value3);
7
8      __mvar6 = *__custom_return_value3;
9
10     <handling timing>
11
12 } while (!(__mvar6 >= 0 && __mvar6 <= 65535));
```

Listing 4.3: This depicts one of the problems my implementation still has, namely the unawareness of type information in the code generator, hence the dereference has to be made manually here.

The code run on the BMC seems to work with one exception: the `MFR_FAN_LUT` command, which crashes and the linux kernel reports an error -110, which, according to the `errno.h` on the BMC is a timeout error.

Most of this section is again a good point to demonstrate the functionality of my compiler, with the exception of the `MFR_FAN_LUT` command that seems to cause a problem with the Linux I2C interface. I have been unable to find the cause of this error, but I suspect it has something to do with the amount of data being transferred, since this is the only command that fails to be transmitted correctly, and it is also the only command that transmits more than 9 bytes of data over I2C (33 bytes to be exact). I am however highly suspicious of this error, since this error seems to only happen sporadically, or at least does not happen when stepping through the code with gdb. The diagnostic tools that I know of and know how to use (which are in this case mainly gdb, dmesg and journalctl as well as any traces I try to produce manually) have not been of great help as I am writing this, since all behaviour I

could observe matches my expectations and should not result in an error (again, which it sometimes doesn't with gdb attached, however the command still fails on the system).

Chapter 5

Some Disadvantages

Despite my efforts to make the compiler backend as functional as possible and to solve as many problems as possible, some problems and disadvantages of my solutions still exist. The following is a enumeration of the problems that I know of.

5.1 Standard violations

As I have mentioned multiple times already, there do exist devices that claim to follow some standard such as PMBus but do not follow the specification to the fullest. While there certainly are devices that are only little or not at all hindered by this, I have no guarantee that my approach is able to handle all devices, no matter how badly they behave. And after all, while it may be unlikely for a device to mishandle a very low standard in their communication stack, like I2C, so badly, this may be the case for higher level standards that could potentially broken in ways I have not even thought of. Of course there is also the possibility that a device does not adhere to an standard and defines its interface completely by itself, for example using some custom parallel input.

5.2 Complexity

Some power management devices are a lot more complex than the PMBus standard may lead you to believe, and PMBus devices are of course not representative for the entirety of power management devices. Although the PMBus standard is powerful enough to fully configure a device like the MAX31785, that can completely manage itself by programming a LUT that translates temperature to fanspeed. Other devices and controller however may contain microcontroller and/or CPLDs/FPGAs that can automate various tasks. The interaction with such a device, including its setup, would drastically increase in complexity and state

changes would almost certainly be hard or even impossible to encode in a power sequence as I have defined it.

5.3 The Template Approach

As I have explained in section section 3.7.1 I have implemented the code generator using a simple simple search and replace procedure that is used to translate some fields encoded in a snippet of C code into the actual values needed. This is a very simple approach that was easy to implement and as it turns out it (mostly) works reasonably well in my use case, however since this approach is completely oblivious to any structure in its output, the code generator has no knowledge of a possible type system that the output needs to follow. This is a problem I have faced when trying to monitor certain values that are represented by a `uint8_t` array in C. My simple inside-of-interval check is unable to handle such a pointer value and the entire compiler is currently unaware of any typing information that could be used to choose a different template. This is a problem especially for commands that return data larger than a word (16-bit) since these are, by the definition of the SMBus spec, done by block transfers.

Chapter 6

Future Work

6.1 Completeness of the implementation

As I have mentioned a few times already, I have not implemented the entire compiler to a point where one could call it complete. I have left out a majority of the commands that PMBus defines and I haven't had a look on any communication other than I2C. Enzian has both other PMBus devices and even non I2C devices that would need to be integrated into the compiler before it can be used productively. In addition there is at least one error in my implementation that disallows it to send certain commands, such as the `MFR_FAN_LUT` command, which must first be solved before the compiler is in any useable state. Also, currently my implementation does not handle any sort of interrupts or alerts that the devices may raise and transmit over the SMBus Alert Line.

6.2 Integration of the frontend compiler

The frontend to this compiler does not yet exist in an ideal form to be integrated into this compilation pipeline. Currently, the entire compiler is a single Python program that produces complete power sequences written in Python. For a reasonable integration with this backend, it would need to be restructured, so as to ignore any communication aspect of the problem that it currently takes into consideration, and produce an abstract power sequence as defined in definition 1.

6.3 Verification of the compiler

Since BMCs are a vital part of a systems safety and security, it is of great interest that whatever runs on a BMC is verified to behave as expected. My backend compiler is no

exception to that, however I have not formally verified my work in this project. Going forward however, the verification of both the backend compiler, as well as the runtime library should be considered before pursuing any practical applications of my work. Of course, the runtime could also be extended with Humbell's work[8], mentioned in section 2.5, so as to include a verified I2C implementation.

6.4 Dynamic Power Management

As Schult et. al. have stated in their work [15], their solution is not yet fit for dynamic power management due to its high bound for time complexity. However, dynamic power management could offer great advantages to the Enzian platform, as a versatile power manager certainly could be of great use in a research platform. A completed compiler chain may be able to improve their running make dynamic power management feasible in the future.

Chapter 7

Conclusion

As I have mentioned in chapter 1, the aim of thesis was to provide the basis for a compiler chain for power sequences and demonstrate its feasibility. In the past few sections, I have provided a a detailed description of my work, as well as its strengths and weaknesses as far as known at this point in time. The compiler I have developed is capable of generating C code representing the input sequence. It not only provides functionality to generate C code representing standard compliant PMBus commands in both read and write directions, but it also provides infrastructure to model non standard compliant behaviour and generate the according C code to implement said behaviour.

However, despite these achievements, there is still a lot of work to be done, as chapters 4 and 5, as well as the rest of this thesis has shown. While the completion of the compiler chain is obviously important, the verification of the system is arguably even more so, since the correctness of the synthesized sequences is absolutely necessary.

Bibliography

[1] Jasmin Schult Daniel Schwyn Michael Giardino David Cock Reto Achermann and Timothy Roscoe. *Declarative Power Sequencing*. paper. Systems Group, Departement of Computer Science, ETH Zürich, 2021.

[2] Jeffrey D. Ullman Alfred V. Aho Ravi Sethi. *Compilers. Principles, Techniques and Tools*. 2nd ed. Addison-Wesley Publishing Company, 1985. ISBN: 0-201-10088-6.

[3] *Enzian BMC Power Management Tools*. URL: `https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bmc-powermgmt` (visited on 10/12/2021).

[4] *Enzian is a research computer built by the Systems Group at ETH Zurich*. URL: `https://enzian.systems` (visited on 10/12/2021).

[5] Systems Group at ETH Zürich. *Enzian Documentation*. URL: `https://wiki.netos.ethz.ch/Enzian` (visited on 10/09/2021).

[6] System Management Interface Forum. *Power Management Bus (PMBus) Protocol Specification. Part II - Command Language*. Version 1.2. Sept. 6, 2010. URL: `https://wiki.netos.ethz.ch/HW/Devices/pmbus` (visited on 09/17/2021).

[7] System Management Interface Forum. *System Management Bus (SMBus) Specification*. Version 3.1. Mar. 19, 2019. URL: `https://wiki.netos.ethz.ch/HW/Devices/smbus` (visited on 09/17/2021).

[8] Lukas Humbel et al. "A Model-Checked I2C Specification". In: *Model Checking Software*. Ed. by Alfons Laarman and Ana Sokolova. Cham: Springer International Publishing, 2021, pp. 177–193. ISBN: 978-3-030-84629-9.

[9] Maxim Integrated. *MAX31785. 6-Channel Intelligent Fan Controller*. 3rd ed. Aug. 15, 2012. URL: `https://datasheets.maximintegrated.com/en/ds/MAX31785.pdf` (visited on 09/17/2021).

[10] João M. P. Cardoso João Bispo Luís Reis. *Multi-Target C Code Generation from MATLAB*. paper. Factulty of Engineering at the University of Porto, Portugal, 2014.

[11] *OCaml Documentation*. URL: `https://www.ocaml.org/docs/` (visited on 10/19/2021).

[12] *OCaml vs. Haskell benchmarks*. URL: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/ocaml-ghc.html`.

[13] *ocaml-yaml – parse and generate YAML 1.1/1.2 files*. URL: `https://github.com/avsm/ocaml-yaml`.

[14] Marek Paška. *An Approcach to Generating C Code with Proven LTL-based Properties*. paper. Departement of Computer Science and Engineering at the University of West Bohemia, Pilsen, Czeck Republic, 2011.

[15] Jasmin Schult. “A model-based approach to platform-level power and clock management”. bachelor. Systems Group, Departement of Computer Science, ETH Zürich, 2020.

[16] NXP Semiconductors. *I2C-bus specification and manual*. 6th ed. UM10204. Apr. 4, 2014. URL: `https://wiki.netos.ethz.ch/HW/Devices/i2c` (visited on 09/17/2021).

Appendix A

Topology Output

General Topology

```
busses:
    bus_1:
        bus_number: 42
        alerts: [ "THIS_IS_AN_ALERT" ]

devices:
    test:
        type_name: test_device_1
        bus_name: bus_1
        interface_settings:
            address: 66

control_map:
    CTRL_1:
        device: test
        control: test_ctrl_1

temperature_monitor_map:
    MON_1:
        test: [ test_mon_1 ]

fanspeed_monitor_map:
    MON_2:
        test: [ test_mon_3 ]
```

Listing A.1: This is a sample of the topologies that were used to test the parsing of topology information

Device Specific Information

```
device_type_name: test_device_1
device_interface: pmbus

pmbus_formats:
    identity:
        identity:
    format_1:
        direct:
            m: 1
            b: 2
            R: 3

control_aliases:
    test_ctrl_1:
        command: FAN_COMMAND_1
        format: format_1
        page: 0

monitor_aliases:
    test_mon_1:
        command: READ_FANSPEED_1
        format: format_1
        page: 0
    test_mon_2:
        command: READ_FANSPEED_1
        format: format_1
        page: 1

pmbus_command_overrides:
```

Listing A.2: This is a sample of the topologies that I used to test the parsing of topology information

The Output of this test

```
Topology Information:
    Device Types:
        test_device_1 (PMBus Device):
            PMBus Data Formats:
                identity: Identity
                format_1:
                    m: 1
                    b: 2
                    R: 3
            PMBus Monitor Aliases:
                test_mon_1:
                    Alias: test_mon_1
                    Command: READ_FANSPEED_1
                    Format: format_1
                    Page: 0
                test_mon_2:
```

```
17                      Alias: test_mon_2
18                      Command: READ_FANSPEED_1
19                      Format: format_1
20                      Page: 1
21              PMBus Control Aliases:
22                  test_ctrl_1:
23                      Alias: test_ctrl_1
24                      Control: FAN_COMMAND_1
25                      Format: format_1
26                      Page: 0
27              PMBus Command Overrides:
28      Busses:
29          bus_1:
30          Bus Number: 42
31          Alerts: [ THIS_IS_AN_ALERT ]
32      Devices:
33          test (PMBus Device):
34              Bus Name: bus_1
35              Address: 66
36              N/A (PMBus Device):
37                  PMBus Data Formats:
38                      identity: Identity
39                      format_1:
40                          m: 1
41                          b: 2
42                          R: 3
43                  PMBus Monitor Aliases:
44                      test_mon_1:
45                          Alias: test_mon_1
46                          Command: READ_FANSPEED_1
47                          Format: format_1
48                          Page: 0
49                      test_mon_2:
50                          Alias: test_mon_2
51                          Command: READ_FANSPEED_1
52                          Format: format_1
53                          Page: 1
54                  PMBus Control Aliases:
55                      test_ctrl_1:
56                          Alias: test_ctrl_1
57                          Control: FAN_COMMAND_1
58                          Format: format_1
59                          Page: 0
60                  PMBus Command Overrides:
61      Controls:
62          CTRL_1 -> (Device: test, Control: test_ctrl_1)
63      Monitors:
64          MON_1:
65              test: [ test_mon_1 ]
66          MON_2:
67              test: [ test_mon_3 ]
```

Listing A.3: This listing depicts the topology as it is represented internally by the compiler.

Appendix B

Simple Topology

General Topology

```
busses:
    pwr_fan:
        bus_number: 4
        alerts: []

devices:
    fans:
        type_name: MAX31785
        bus_name: pwr_fan
        interface_settings:
        address: 0x52

control_map:
    CPU_FAN_PWM:
        device: fans
        control: FAN_0_PWM
    CPU_FAN_RPM:
        device: fans
        control: FAN_0_RPM
    CPU_FAN_CONFIG:
        device: fans
        control: FAN_0_CONFIG
    FPGA_FAN_PWM:
        device: fans
        control: FAN_1_PWM
    FPGA_FAN_RPM:
        device: fans
        control: FAN_1_RPM
    FPGA_FAN_CONFIG:
        device: fans
        control: FAN_1_CONFIG

```

```
temp_monitor_map:
    CPU_TEMP:
        fans: [ T_DIODE_0 ]
    FPGA_TEMP:
        fans: [ T_DIODE_1 ]

fanspeed_monitor_map:
    CPU_FAN:
        fans: [ FAN_0 ]
    FPGA_FAN:
        fans: [ FAN_1 ]
```

Listing B.1: This is the general topology that I have used for the simple testcases

Device Specific Topology

```
device_type_name: MAX31785
device_interface: pmbus

pmbus_formats:
    identity:
        identity:
    pwm:
        direct:
            m: 1
            b: 0
            R: 2
    rpm:
        direct:
            m: 1
            b: 0
            R: 0
    temp:
        direct:
            m: 1
            b: 0
            R: 2

control_aliases:
    FAN_0_PWM:
        command: FAN_COMMAND_1
        format: pwm
        page: 0
    FAN_0_RPM:
        command: FAN_COMMAND_1
        format: rpm
        page: 0
    FAN_0_CONFIG:
        command: FAN_CONFIG_1_2
        format: identity
        page: 0
    FAN_1_PWM:
        command: FAN_COMMAND_1
```

```
38          format: pwm
39          page: 1
40      FAN_1_RPM:
41          command: FAN_COMMAND_1
42          format: rpm
43          page: 1
44      FAN_1_CONFIG:
45          command: FAN_CONFIG_1_2
46          format: identity
47          page: 1
48
49  monitor_aliases:
50      FAN_0:
51          command: READ_FAN_SPEED_1
52          format: rpm
53          page: 0
54      FAN_1:
55          command: READ_FAN_SPEED_1
56          format: rpm
57          page: 1
58      T_DIODE_0:
59          command: TEMPERATURE_1
60          format: temp
61          page: 6
62      T_DIODE_1:
63          command: TEMPERATURE_1
64          format: temp
65          page: 7
66
67  pmbus_command_overrides:
```

Listing B.2: This is the device specific topology that I have used for the simple testcases. It only consists of a single device which is depicted here.

Appendix C

Simple Test Output Sample

Generator C Code

```
#include <fcntl.h>
#include <runtime.h>
#include <malloc.h>

struct bus_instance* pwr_fan;

void setup()
{
    struct i2c_bus_instance* __tmp_bus3 = (struct
        i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
    __tmp_bus3->i2c_adapter_fd = open("/dev/i2c-4", O_RDWR);
    pwr_fan = (struct bus_instance*)__tmp_bus3;
}

void cleanup()
{
    free(pwr_fan);
}

void power_sequence()
{
    pmbus_page_write(pwr_fan, 82, 0);
    int __mvar1;
    runtime_sleep_ms(100 - 10);
    struct timespec __start2;
    clock_gettime(CLOCK_REALTIME, &__start2);
    do {
        runtime_sleep_ms(10);
        uint16_t rv =
            pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
            82), 1, 0, 0);

```

```
30          __mvar1 = rv;
31          struct timespec current;
32          clock_gettime(CLOCK_REALTIME, &current);
33          struct timespec diff = { current.tv_sec - __start2.tv_sec,
                current.tv_nsec - __start2.tv_nsec };
34          long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
35          if (diff_long > 1000) runtime_timeout();
36      } while (!(__mvar1 >= 900 && __mvar1 <= 1100));
37  }
38
39  int main()
40  {
41      setup();
42      power_sequence();
43      cleanup();
44      return 0;
45  }
```

Listing C.1: This listing shows the full output of one of the simple test cases

Trace

```
1  PMBUS: Writing: ADDRESS: 82, PAGE: 0
2  SMBUS: Writing command '0' and data byte '0' to address '0x52'.
3  I2C: Writing Data: 0x52, Max Length: 2 bytes
4  PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
5  SMBUS: Reading word from command '144' at address '0x52'.
6  I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xb8 0x0b
7  PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
8  SMBUS: Reading word from command '144' at address '0x52'.
9  I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x70 0x17
10 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
11 SMBUS: Reading word from command '144' at address '0x52'.
12 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x58 0x1b
13 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
14 SMBUS: Reading word from command '144' at address '0x52'.
15 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x88 0x13
16 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
17 SMBUS: Reading word from command '144' at address '0x52'.
18 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xb8 0x0b
19 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
20 SMBUS: Reading word from command '144' at address '0x52'.
21 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x88 0x13
22 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
23 SMBUS: Reading word from command '144' at address '0x52'.
24 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x70 0x17
25 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
26 SMBUS: Reading word from command '144' at address '0x52'.
27 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xd0 0x07
28 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
29 SMBUS: Reading word from command '144' at address '0x52'.
30 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x28 0x23
31 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
32 SMBUS: Reading word from command '144' at address '0x52'.
33 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xe8 0x03
```

Listing C.2: This listing shows the trace of one of the simple test cases

Appendix D

Complex Tests

Generated C Code

```c
#include <fcntl.h>
#include <runtime.h>
#include <malloc.h>

struct bus_instance* seq;
struct bus_instance* clk;
struct bus_instance* pwr_fan;

void setup()
{
    struct i2c_bus_instance* __tmp_bus11 = (struct
        i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
    __tmp_bus11->i2c_adapter_fd = open("/dev/i2c-0", O_RDWR);
    seq = (struct bus_instance*)__tmp_bus11;

    struct i2c_bus_instance* __tmp_bus12 = (struct
        i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
    __tmp_bus12->i2c_adapter_fd = open("/dev/i2c-2", O_RDWR);
    clk = (struct bus_instance*)__tmp_bus12;

    struct i2c_bus_instance* __tmp_bus13 = (struct
        i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
    __tmp_bus13->i2c_adapter_fd = open("/dev/i2c-4", O_RDWR);
    pwr_fan = (struct bus_instance*)__tmp_bus13;
}

void cleanup()
{
    free(seq);
    free(clk);
    free(pwr_fan);
}
```

```
30
31 void power_sequence()
32 {
33     pmbus_page_write(pwr_fan, 82, 0);
34     smbus_write_word(pwr_fan, 82, 209, 8192);
35     pmbus_page_write(pwr_fan, 82, 0);
36     smbus_write_word(pwr_fan, 82, 209, 10240);
37     pmbus_page_write(pwr_fan, 82, 0);
38     smbus_write_word(pwr_fan, 82, 209, 8192);
39     pmbus_page_write(pwr_fan, 82, 0);
40     int __mvar4;
41     runtime_sleep_ms(300 - 10);
42     struct timespec __start5;
43     clock_gettime(CLOCK_REALTIME, &__start5);
44     do {
45         runtime_sleep_ms(10);
46         uint16_t rv =
               pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
               82), 1, 0, 0);
47
48         __mvar4 = rv;
49         struct timespec current;
50         clock_gettime(CLOCK_REALTIME, &current);
51         struct timespec diff = { current.tv_sec - __start5.tv_sec,
               current.tv_nsec - __start5.tv_nsec };
52         long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
53         if (diff_long > 100) runtime_timeout();
54     } while (!(__mvar4 >= 0 && __mvar4 <= 65535));
55     pmbus_page_write(pwr_fan, 82, 0);
56     pmbus_fan_config_1_2_write(pwr_fan, 82, 80);
57     pmbus_page_write(pwr_fan, 82, 0);
58     pmbus_fan_command_1_write(pwr_fan, 82, pmbus_direct_format(1000,
           1, 0, 0));
59     pmbus_page_write(pwr_fan, 82, 0);
60     int __mvar7;
61     runtime_sleep_ms(250 - 10);
62     struct timespec __start8;
63     clock_gettime(CLOCK_REALTIME, &__start8);
64     do {
65         runtime_sleep_ms(10);
66         uint16_t rv =
               pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
               82), 1, 0, 0);
67
68         __mvar7 = rv;
69         struct timespec current;
70         clock_gettime(CLOCK_REALTIME, &current);
71         struct timespec diff = { current.tv_sec - __start8.tv_sec,
               current.tv_nsec - __start8.tv_nsec };
72         long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
73         if (diff_long > 15000) runtime_timeout();
74     } while (!(__mvar7 >= 900 && __mvar7 <= 1100));
```

```
75      pmbus_page_write(pwr_fan, 82, 0);
76      pmbus_fan_command_1_write(pwr_fan, 82, pmbus_direct_format(2000,
            1, 0, 0));
77      pmbus_page_write(pwr_fan, 82, 0);
78      int __mvar9;
79      runtime_sleep_ms(250 - 10);
80      struct timespec __start10;
81      clock_gettime(CLOCK_REALTIME, &__start10);
82      do {
83          runtime_sleep_ms(10);
84          uint16_t rv =
                pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
                82), 1, 0, 0);
85
86          __mvar9 = rv;
87          struct timespec current;
88          clock_gettime(CLOCK_REALTIME, &current);
89          struct timespec diff = { current.tv_sec - __start10.tv_sec,
                current.tv_nsec - __start10.tv_nsec };
90          long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
91          if (diff_long > 15000) runtime_timeout();
92      } while (!(__mvar9 >= 1900 && __mvar9 <= 2100));
93  }
94
95  int main()
96  {
97      setup();
98      power_sequence();
99      cleanup();
100     return 0;
101 }
```

Listing D.1: This listing depicts a sample output of the compiler, repersenting the complex test cases

Trace

```
1  PMBUS: Writing: ADDRESS: 82, PAGE: 0
2  SMBUS: Writing command '0' and data byte '0' to address '0x52'.
3  I2C: Writing Data: 0x52, Max Length: 2 bytes
4  SMBUS: Writing command '209' and data word '8192' to address '0x52'.
5  I2C: Writing Data: 0x52, Max Length: 3 bytes
6  PMBUS: Writing: ADDRESS: 82, PAGE: 0
7  SMBUS: Writing command '0' and data byte '0' to address '0x52'.
8  I2C: Writing Data: 0x52, Max Length: 2 bytes
9  SMBUS: Writing command '209' and data word '10240' to address '0x52'.
10 I2C: Writing Data: 0x52, Max Length: 3 bytes
11 PMBUS: Writing: ADDRESS: 82, PAGE: 0
12 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
13 I2C: Writing Data: 0x52, Max Length: 2 bytes
14 SMBUS: Writing command '209' and data word '8192' to address '0x52'.
15 I2C: Writing Data: 0x52, Max Length: 3 bytes
16 PMBUS: Writing: ADDRESS: 82, PAGE: 0
17 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
18 I2C: Writing Data: 0x52, Max Length: 2 bytes
19 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
20 SMBUS: Reading word from command '144' at address '0x52'.
21 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xb8 0x0b
22 PMBUS: Writing: ADDRESS: 82, PAGE: 0
23 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
24 I2C: Writing Data: 0x52, Max Length: 2 bytes
25 PMBUS: Writing: ADDRESS: 82, FAN_CONFIG_1_2: 80
26 SMBUS: Writing command '58' and data byte '80' to address '0x52'.
27 I2C: Writing Data: 0x52, Max Length: 2 bytes
28 PMBUS: Writing: ADDRESS: 82, PAGE: 0
29 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
30 I2C: Writing Data: 0x52, Max Length: 2 bytes
31 PMBUS: Writing: ADDRESS: 82, FAN_COMMAND_1: 1000
32 SMBUS: Writing command '59' and data word '1000' to address '0x52'.
33 I2C: Writing Data: 0x52, Max Length: 3 bytes
34 PMBUS: Writing: ADDRESS: 82, PAGE: 0
35 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
36 I2C: Writing Data: 0x52, Max Length: 2 bytes
37 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
38 SMBUS: Reading word from command '144' at address '0x52'.
39 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x70 0x17
40 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
41 SMBUS: Reading word from command '144' at address '0x52'.
42 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x58 0x1b
43 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
44 SMBUS: Reading word from command '144' at address '0x52'.
45 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x88 0x13
46 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
47 SMBUS: Reading word from command '144' at address '0x52'.
```

```
48 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xb8 0x0b
49 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
50 SMBUS: Reading word from command '144' at address '0x52'.
51 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x88 0x13
52 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
53 SMBUS: Reading word from command '144' at address '0x52'.
54 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x70 0x17
55 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
56 SMBUS: Reading word from command '144' at address '0x52'.
57 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xd0 0x07
58 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
59 SMBUS: Reading word from command '144' at address '0x52'.
60 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x28 0x23
61 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
62 SMBUS: Reading word from command '144' at address '0x52'.
63 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xe8 0x03
64 PMBUS: Writing: ADDRESS: 82, PAGE: 0
65 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
66 I2C: Writing Data: 0x52, Max Length: 2 bytes
67 PMBUS: Writing: ADDRESS: 82, FAN_COMMAND_1: 2000
68 SMBUS: Writing command '59' and data word '2000' to address '0x52'.
69 I2C: Writing Data: 0x52, Max Length: 3 bytes
70 PMBUS: Writing: ADDRESS: 82, PAGE: 0
71 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
72 I2C: Writing Data: 0x52, Max Length: 2 bytes
73 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
74 SMBUS: Reading word from command '144' at address '0x52'.
75 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xd0 0x07
```

Listing D.2: This listing depicts a sample trace of the complex test cases

Appendix E

Non standard compliant tests

Generated C Code

```
1  #include <fcntl.h>
2  #include <runtime.h>
3  #include <malloc.h>
4
5  struct bus_instance* seq;
6  struct bus_instance* clk;
7  struct bus_instance* pwr_fan;
8
9  void setup()
10 {
11     struct i2c_bus_instance* __tmp_bus13 = (struct
           i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
12     __tmp_bus13->i2c_adapter_fd = open("/dev/i2c-0", O_RDWR);
13     seq = (struct bus_instance*)__tmp_bus13;
14
15     struct i2c_bus_instance* __tmp_bus14 = (struct
           i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
16     __tmp_bus14->i2c_adapter_fd = open("/dev/i2c-2", O_RDWR);
17     clk = (struct bus_instance*)__tmp_bus14;
18
19     struct i2c_bus_instance* __tmp_bus15 = (struct
           i2c_bus_instance*)malloc(sizeof(struct i2c_bus_instance));
20     __tmp_bus15->i2c_adapter_fd = open("/dev/i2c-4", O_RDWR);
21     pwr_fan = (struct bus_instance*)__tmp_bus15;
22 }
23
24 void cleanup()
25 {
26     free(seq);
27     free(clk);
28     free(pwr_fan);
29 }
```

```
30
31 void power_sequence()
32 {
33     pmbus_page_write(pwr_fan, 82, 0);
34     smbus_write_word(pwr_fan, 82, 209, 8192);
35     pmbus_page_write(pwr_fan, 82, 0);
36     smbus_write_word(pwr_fan, 82, 209, 10240);
37     pmbus_page_write(pwr_fan, 82, 0);
38     smbus_write_word(pwr_fan, 82, 209, 8192);
39     pmbus_page_write(pwr_fan, 82, 0);
40     int __mvar5;
41     runtime_sleep_ms(300 - 10);
42     struct timespec __start6;
43     clock_gettime(CLOCK_REALTIME, &__start6);
44     do {
45         runtime_sleep_ms(10);
46         uint16_t rv =
               pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
               82), 1, 0, 0);
47
48         __mvar5 = rv;
49         struct timespec current;
50         clock_gettime(CLOCK_REALTIME, &current);
51         struct timespec diff = { current.tv_sec - __start6.tv_sec,
               current.tv_nsec - __start6.tv_nsec };
52         long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
53         if (diff_long > 100) runtime_timeout();
54     } while (!(__mvar5 >= 0 && __mvar5 <= 65535));
55     pmbus_page_write(pwr_fan, 82, 0);
56     pmbus_fan_config_1_2_write(pwr_fan, 82, 80);
57     pmbus_page_write(pwr_fan, 82, 0);
58     smbus_write_word(pwr_fan, 82, 241, 57730);
59     pmbus_page_write(pwr_fan, 82, 0);
60     pmbus_fan_config_1_2_write(pwr_fan, 82, 208);
61     pmbus_page_write(pwr_fan, 82, 0);
62     pmbus_fan_command_1_write(pwr_fan, 82, pmbus_direct_format(2000,
           1, 0, 0));
63     pmbus_page_write(pwr_fan, 82, 0);
64     int __mvar9;
65     runtime_sleep_ms(8000 - 10);
66     struct timespec __start10;
67     clock_gettime(CLOCK_REALTIME, &__start10);
68     do {
69         runtime_sleep_ms(10);
70         uint16_t rv =
               pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
               82), 1, 0, 0);
71
72         __mvar9 = rv;
73         struct timespec current;
74         clock_gettime(CLOCK_REALTIME, &current);
```

```
75          struct timespec diff = { current.tv_sec - __start10.tv_sec,
                current.tv_nsec - __start10.tv_nsec };
76          long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
77          if (diff_long > 100) runtime_timeout();
78      } while (!(__mvar9 >= 0 && __mvar9 <= 65535));
79      pmbus_page_write(pwr_fan, 82, 0);
80      pmbus_fan_command_1_write(pwr_fan, 82, pmbus_direct_format(2000,
            1, 0, 0));
81      pmbus_page_write(pwr_fan, 82, 0);
82      int __mvar11;
83      runtime_sleep_ms(8000 - 10);
84      struct timespec __start12;
85      clock_gettime(CLOCK_REALTIME, &__start12);
86      do {
87          runtime_sleep_ms(10);
88          uint16_t rv =
                pmbus_direct_format_inverse(pmbus_read_fan_speed_1_read(pwr_fan,
                82), 1, 0, 0);
89
90          __mvar11 = rv;
91          struct timespec current;
92          clock_gettime(CLOCK_REALTIME, &current);
93          struct timespec diff = { current.tv_sec - __start12.tv_sec,
                current.tv_nsec - __start12.tv_nsec };
94          long diff_long = diff.tv_sec * 1000 + diff.tv_nsec / 1000000;
95          if (diff_long > 100) runtime_timeout();
96      } while (!(__mvar11 >= 0 && __mvar11 <= 65535));
97      pmbus_page_write(pwr_fan, 82, 0);
98      pmbus_fan_command_1_write(pwr_fan, 82, pmbus_direct_format(2000,
            1, 0, 0));
99  }
100
101 int main()
102 {
103     setup();
104     power_sequence();
105     cleanup();
106     return 0;
107 }
```

Listing E.1: This listing shows a sample output of the non standard test cases

Trace

```
1  PMBUS: Writing: ADDRESS: 82, PAGE: 0
2  SMBUS: Writing command '0' and data byte '0' to address '0x52'.
3  I2C: Writing Data: 0x52, Max Length: 2 bytes
4  SMBUS: Writing command '209' and data word '8192' to address '0x52'.
5  I2C: Writing Data: 0x52, Max Length: 3 bytes
6  PMBUS: Writing: ADDRESS: 82, PAGE: 0
7  SMBUS: Writing command '0' and data byte '0' to address '0x52'.
8  I2C: Writing Data: 0x52, Max Length: 2 bytes
9  SMBUS: Writing command '209' and data word '10240' to address '0x52'.
10 I2C: Writing Data: 0x52, Max Length: 3 bytes
11 PMBUS: Writing: ADDRESS: 82, PAGE: 0
12 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
13 I2C: Writing Data: 0x52, Max Length: 2 bytes
14 SMBUS: Writing command '209' and data word '8192' to address '0x52'.
15 I2C: Writing Data: 0x52, Max Length: 3 bytes
16 PMBUS: Writing: ADDRESS: 82, PAGE: 0
17 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
18 I2C: Writing Data: 0x52, Max Length: 2 bytes
19 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
20 SMBUS: Reading word from command '144' at address '0x52'.
21 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0xb8 0x0b
22 PMBUS: Writing: ADDRESS: 82, PAGE: 0
23 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
24 I2C: Writing Data: 0x52, Max Length: 2 bytes
25 PMBUS: Writing: ADDRESS: 82, FAN_CONFIG_1_2: 80
26 SMBUS: Writing command '58' and data byte '80' to address '0x52'.
27 I2C: Writing Data: 0x52, Max Length: 2 bytes
28 PMBUS: Writing: ADDRESS: 82, PAGE: 0
29 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
30 I2C: Writing Data: 0x52, Max Length: 2 bytes
31 SMBUS: Writing command '241' and data word '57730' to address '0x52'.
32 I2C: Writing Data: 0x52, Max Length: 3 bytes
33 PMBUS: Writing: ADDRESS: 82, PAGE: 0
34 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
35 I2C: Writing Data: 0x52, Max Length: 2 bytes
36 PMBUS: Writing: ADDRESS: 82, FAN_CONFIG_1_2: 208
37 SMBUS: Writing command '58' and data byte '208' to address '0x52'.
38 I2C: Writing Data: 0x52, Max Length: 2 bytes
39 PMBUS: Writing: ADDRESS: 82, PAGE: 0
40 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
41 I2C: Writing Data: 0x52, Max Length: 2 bytes
42 PMBUS: Writing: ADDRESS: 82, FAN_COMMAND_1: 2000
43 SMBUS: Writing command '59' and data word '2000' to address '0x52'.
44 I2C: Writing Data: 0x52, Max Length: 3 bytes
45 PMBUS: Writing: ADDRESS: 82, PAGE: 0
46 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
47 I2C: Writing Data: 0x52, Max Length: 2 bytes
48 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
49 SMBUS: Reading word from command '144' at address '0x52'.
```

```
50 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x70 0x17
51 PMBUS: Writing: ADDRESS: 82, PAGE: 0
52 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
53 I2C: Writing Data: 0x52, Max Length: 2 bytes
54 PMBUS: Writing: ADDRESS: 82, FAN_COMMAND_1: 2000
55 SMBUS: Writing command '59' and data word '2000' to address '0x52'.
56 I2C: Writing Data: 0x52, Max Length: 3 bytes
57 PMBUS: Writing: ADDRESS: 82, PAGE: 0
58 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
59 I2C: Writing Data: 0x52, Max Length: 2 bytes
60 PMBUS: Reading: ADDRESS: 82, READ_FAN_SPEED_1
61 SMBUS: Reading word from command '144' at address '0x52'.
62 I2C: Writing 1 bytes of Data: 0x52, and Reading 2 bytes of Data:
       0x58 0x1b
63 PMBUS: Writing: ADDRESS: 82, PAGE: 0
64 SMBUS: Writing command '0' and data byte '0' to address '0x52'.
65 I2C: Writing Data: 0x52, Max Length: 2 bytes
66 PMBUS: Writing: ADDRESS: 82, FAN_COMMAND_1: 2000
67 SMBUS: Writing command '59' and data word '2000' to address '0x52'.
68 I2C: Writing Data: 0x52, Max Length: 3 bytes
```

Listing E.2: This listing shows a sample trace of the non standard test cases

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Generating Power Management Code from Declarative Descriptions

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Vogel	Linus Ulysses

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Effretikon, 21/10/2021	L.Vogel

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.