Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 291b

Systems Group, Department of Computer Science, ETH Zurich

Modeling the I2C Bus

by
Jan Schär

Supervised by
Prof. Timothy Roscoe
Lukas Humbel
David Cock

March–September 2020

inf | Informatik
Computer Science

1 Abstract

I^{2}C (Inter-Integrated Circuit) is a standard bus protocol which can for example connect sensors or voltage controllers to a processor. It is commonly used in computers to support critical functionality, hence it is essential that it works correctly. However, current implementations in devices relatively often violate the specification, and host side hardware interfaces may have limitations that prohibit even compliant operations. With a formally verified implementation of I^{2}C, these problems could be avoided.

In this work, I present a model implementation of both sides of the bus. The implementation is written in Haskell and executable, and consists of multiple layers. Ultimately, the goal is to enable formal verification of this implementation. But before attempting this, it should be thoroughly tested and evaluated to ensure that it is useful and works in practice, and is likely to be correct.

An end-to-end correctness property for the model has been formulated as a random test using the QuickCheck Haskell library, which is already a strong hint that it holds. Formal verification of the property is left to future work.

The model implementation was connected through GPIO (general purpose input output) pins to a physical memory IC, and compared with a software model of the memory IC through random testing. This shows that the model works in practice and is compatible with the hardware implementation of the protocol in the IC.

I also compared the interfaces of the model to the I^{2}C API of the Linux kernel, and to the interface of a hardware implementation of the host side. This helps to ensure that the model is practical and not too restrictive.

Additionally, different examples of non-compliant devices were studied.

Contents

2 Acknowledgments

I would like to thank Prof. Timothy Roscoe for giving me the opportunity to write my Bachelor's thesis in the Systems group. Special thanks to Lukas Humbel for supporting and encouraging me, answering my questions, and guiding me throughout the project in our weekly meetings, and for providing me with lots of suggestions and comments on my report. I also thank David Cock for his help in supervising my work, and Reto Achermann for giving me helpful feedback on drafts of the report.

3 Introduction

I^2C (Inter-Integrated Circuit) [1] is a standard bus for communication between integrated circuits. Devices on the bus are either so-called 'masters' or 'slaves'. A slave device could be e.g. a memory chip, a sensor or a voltage regulator. The master can issue commands to the slaves to read or write data, it could e.g. be software running on a CPU. I^2C buses are ubiquitous in today's computers, and often perform critical functions like power management (e.g. controlling and monitoring voltage regulators). Hence, it is important that these buses operate correctly.

However, existing hardware implementations in slave devices relatively often have various bugs or violations of the specification, which leads to incompatibility. To allow communication with such devices, some master implementations have multiple flags to enable non-compliant behavior. In addition, hardware master implementations can have restrictions which make certain compliant operations impossible. The result is that only some combinations of slave devices and hardware master interfaces are compatible.

Previous approaches to ensure correctness of I^2C implementations are based on testing. This is certainly useful for catching bugs. However, as a famous quote by Edsger W. Dijkstra says, "Program testing can be used to show the presence of bugs, but never to show their absence!" Hence, in the last decades, there have been efforts to create formally verified implementations of various parts of computer systems. Some examples are the VAMP (Verified Architecture Microprocessor) [21], the CompCert compiler [22] and the seL4 microkernel [23]. This provides strong guarantees that these systems work correctly.

With an end-to-end formally verified implementation of I^2C, the problems mentioned above could be avoided. This is especially important since hardware cannot simply be patched like software, it needs to be correct from the beginning.

Having such a model implementation will also allow proofs of bigger subsystems. We may want to limit the rights of device drivers, such that they can only access a subset of slave devices on the bus, or even just certain functionality of a device. A right is defined by what is allowed to happen on a slave device, but is implemented by restricting the transactions that the driver can execute. The formal model allows us to relate the transactions to what happens on the device, which allows to prove that the implementation of the right is correct (i.e. any allowed transaction does not violate the definition of the right).

Problem statement The goal of this work is the creation of a model I^2C implementation. It should be amenable to formal analysis, which means that it allows correctness properties to be stated, and that it should be relatively easy to prove these properties in a proof assistant. Ultimately, we want to be able to do an end-to-end formal verification.

At the same time, the model implementation should conform to the I^2C specification. The specification is written in English, not in a formal language, thus adherence of the model to the specification cannot be formally proved. Instead, this must be ensured by carefully reading the specification, and by testing or proving compatibility with existing implementations. Consequently, this compatibility is also a goal of the model.

To ensure that the model is useful, the interface that it exposes to device

drivers should provide sufficient flexibility.

It should also be possible to express other APIs (such as the I^2C API of Linux) using the model.

Thesis outline In this work, I present a model implementation of an I^2C master and slave interface, written in Haskell (section 6). The language choice means that the model is executable, but also relatively easy to reason about in proof assistants. The model is split into multiple layers, which makes it possible to reason at different levels of abstraction, and allows proofs to be built layer by layer.

In section 7, I analyze the I^2C master API used in the Linux kernel, and compare it to the model implementation. This allows us to see if the high level interface of the model is flexible enough in practice.

I look at several examples of non-compliant devices (section 8), and describe how they violate the specification. I discuss different approaches how masters can deal with this problem.

I compare the model interface to the interface of a hardware master interface (section 9). This allows us to find potential problems and limitations in both.

The model allows interesting properties to be defined. Two such properties were written in the form of QuickCheck tests (section 10). While actual proofs of the properties are left to future work, this already gives a strong hint that they are correct, which is useful to do before attempting to build a formal proof. Also, a significant part of the challenge in formal verification is to formally specify what it means to be 'correct', in other words how these correctness properties should be formulated.

One can define a high level abstraction of how the bus connects masters and slaves, and then state as a property that the complete protocol stack behaves the same as the much simpler abstraction from the the point of view of devices and device drivers (see section 10.2). In other words, this property means that the implementation works correctly (for a single master and slave) when one views the high level abstraction as a specification of how it should work.

If one then additionally models the device itself and the device driver, the I^2C bus which connects them could be replaced by the abstraction mentioned above, thus enabling proofs of bigger subsystems.

Finally, I present a model implementation of an EEPROM (Electrically Erasable Programmable Read-Only Memory), which has been checked through random testing against an actual EEPROM connected to the master side of the Haskell model via GPIOs (section 11). This shows that the model actually works in practice and is compatible with existing implementations.

4 Related work

In [18], the UVM (Universal Verification Methodology) methodology is applied to verify an existing I^2C master core from Opencores. It uses a testing approach: Up to three random bytes are written to one of three slaves, the slave stores them. Then, the bytes are read back from the slave and compared with the original bytes. During the testing, coverage information is collected.

The work has a similar goal to my work, which is to ensure correctness of an I^2C implementation. However, the approach is different. The UVM work only covers the hardware part of the master side, while my work also covers the slave side and the high level software interface. The ultimate goal of my work is to enable formal correctness proofs; random testing, while still valuable, cannot guarantee correctness.

In the memory protection model proposed by Achermann *et al.* [19], an EEPROM connected through I^2C could be understood as an additional address space. Since the EEPROM is accessed via I^2C transactions, a model of the behavior of the bus is necessary to ensure correctness. Such a model is provided by my work.

5 Background

This section provides a short summary of the I^{2}C, SMBus and PMbus specifications, and some discussion. It should be enough to understand the rest of my thesis. While the model itself only covers I^{2}C, and SMBus and PMBus would be implemented on top, it is still useful to know about how the later two use I^{2}C and what features they require from it.

5.1 I^{2}C bus

I^{2}C is a standard bus protocol for communication between ICs. The specification [1] was first released in 1982 by Philips Semiconductors, and it is now a de facto standard used by many companies. The bus is used in many electronic devices, e.g. computers, and also sometimes for communication between devices such as in HDMI (where it is used e.g. for reading out the resolution or changing brightness of a monitor) [24]. The bus is relatively low speed and mainly used for configuration. What follows is a short summary of the parts of the specification which are relevant to my thesis. Nearly everything in this section can be directly derived from the specification [1], for better readability I did not put a reference after each sentence.

The bus has two wires, clock (SCL) and data (SDA), which are pulled up to the supply voltage. ICs may only drive the lines low, not high.[1] Devices participating in the bus are either masters or slaves, there can be multiple of both on the same bus. But there can also be intermediary devices which connect different bus segments, like bus buffers or multiplexers. Each slave has a seven-bit address (or ten-bit, see below), which should be unique.[2] Communication is always started by a master talking to a slave, identified by its address.

If SDA transitions from high to low while SCL is high, this is a START condition. If SDA transitions from low to high while SCL is high, it is a STOP condition. If SDA stays stable during a complete high period of SCL, a bit is transmitted, 1 if SDA is high, 0 otherwise.

START and STOP conditions and the clock signal are generated by a master. Slaves can only transmit a 0 bit by driving SDA low, transmit a 1 bit by doing nothing, and perform *clock stretching*. Clock stretching means to drive SCL low during the clock low period, which prevents prevents the clock from rising and thus blocks the bus. It is necessary if the slave needs more time before it can receive or transmit further data.

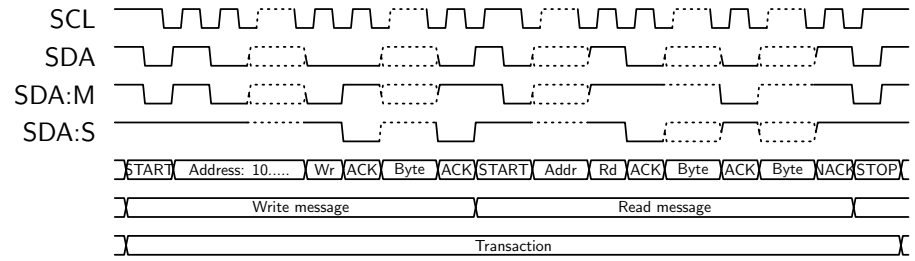
Communication happens in *transactions*, which contain one or more *messages*.[3] Figure 1 shows a complete example transaction. Each message begins with a START condition, and the transaction ends with a STOP condition. After each START, the master transmits a byte containing the 7-bit slave address and the read/write bit for this message. An acknowledge bit follows; if a slave is present at the specified address, it will transmit a 0 bit (ACK), otherwise the master will see a 1 bit (NACK). (Some slaves will not send an ACK if they are

[1] Under some conditions masters are allowed to drive SCL high. Also, this is not true for UFm mode, which I do not cover here.

[2] An exception is mentioned in [1, p. 14, note 6].

[3] This terminology is from the I^{2}C specification [1], but it is not clearly defined there, and often 'transfer' is used as a general term instead. Other works may use different terminology, e.g. Linux uses 'transfer'/'xfer' instead of 'transaction'.

Figure 1: This figure shows an example I^2C transaction. The SDA:M and SDA:S signals show the contributions of master and slave respectively to the SDA signal. The transfer of addresses (7 bits) and bytes (8 bits) have been abbreviated. Rd/Wr stands for Read/Write. The transaction has two messages, first one byte is written, then two bytes are read.



busy.) A message will not continue after a NACK. Next, the data bytes of the message follow.

If the message is a read (read bit set), the slave will transmit a sequence of bytes, which the master receives. After each byte, the master sends an ACK bit if the slave should continue sending data, else a NACK.

For write messages, the master transmits zero or more bytes, and the slave will send an ACK after each byte if it is able to process it.

When the bus is free, both lines (SDA and SCL) are high. During a transaction, the bus is busy, and other masters may not start a transaction. However, it is possible that multiple masters begin communicating at the same time. As soon as a device sees a 0 bit on the bus when it intended to transmit a 1 bit, it has detected *arbitration loss* and will stop transmitting. A master may retry the transaction in this case (after the current one is finished). The other devices will just continue with the ongoing transaction and not even notice that arbitration happened. It can also happen that there is an *undefined condition* [1, p. 12]. It occurs when a START and a STOP, a START and a bit, or a STOP and a bit are generated by different masters simultaneously.

The I^2C specification defines some special, reserved addresses [1, p. 17]:

The START byte is a read message at address 0, which will be NACKed. The actual transaction follows after that. It can be used to allow longer polling intervals in bit-banging software slaves.

There are High-Speed (Hs) Mode master codes. In Hs mode, the master drives SCL both high and low to achieve a higher clock frequency; if multiple masters were active simultaneously, it would cause short circuits. Thus, each master first transmits its unique master code at a lower speed before entering Hs mode, forcing arbitration to happen early.

Four addresses are reserved for ten-bit addressing. The remaining 8 bits are sent by the master in the first byte of a write message.

Other reserved addresses include general call and device ID.

It may happen that the SDA line becomes stuck low, e.g. if a slave misses a clock pulse due to noise. In this case a master may send up to nine clock pulses, until SDA goes high (this operation is called bus clear). If this does not help, or SCL is stuck low, the devices on the bus need to be reset.

The specification defines 5 different data rates: Standard mode (Sm), Fast-

mode (Fm), Fast-mode plus (Fm+), High-speed mode (Hs) and Ultra Fast-mode (UFm). Hs and UFm change the way the bus operates (Hs mode master codes were already mentioned), for this reason I do not handle these two modes in my work.

5.1.1 Observations

Read messages Note that for read messages, the slave has no way to tell the master that all bytes have been read; if it just stops transmitting, the master will receive dummy 0xff bytes. Also, the master has to read at least one byte in each read message; if it tried to read zero bytes, the following START or STOP condition could get overridden by a 0 bit transmitted by the slave.

Undefined conditions Assuming that we have a single master on the bus, all devices conform to the specification and there are no transmission errors (e.g. due to noise), there should be no arbitration losses or undefined conditions, i.e. transactions always succeed. In section 10.1, this is stated as a QuickCheck property for my model. However, if we have multiple masters, we need to be careful about undefined conditions. As the name indicates, the specification does not define what will happen if they occur, thus we should make sure they do not happen. They can occur in two situations: In the first, one master executes a transaction with messages $x_1, \ldots, x_n$ and another master simultaneously executes a transaction $x_1, \ldots, x_n, y_1, \ldots, y_m$ $(n, m \geq 1)$, i.e. the first list of messages is a strict prefix of the second list of messages. Here, a START and STOP condition occur simultaneously. (Remember that a START is sent before each message, and a STOP at the end of a transaction.) In the second case, the two message lists are $x_1, \ldots, x_n, w, y_1, \ldots, y_m$ and $x_1, \ldots, x_n, w', z_1, \ldots, z_k$ $(n, m, k \geq 0)$, where message w and w' have the same address and are both writes, and the contents of w is a strict prefix of w'. Here, a bit occurs simultaneously with a START or a STOP condition.

One way to prevent undefined conditions is the use of master codes like in Hs mode, where every master has a unique address (the master code) which is not assigned to a slave, and an empty message with this address is prepended to every transaction. This scheme forces arbitration to happen during transmission of the master code, such that only one master wins, and we are again in the single master case. (Unless Hs mode is actually used, the Hs mode master codes should not be used for this, since they additionally signal to slaves that transmission speed will change.)

5.2 SMBus

SMBus (System Management Bus) is a protocol that builds on top of I^2C, specified in [2]. It defines a fixed set of command formats, which helps standardize the interface of different devices, and allows to have a common higher level API than just reading and writing bytes.

SMBus has a slightly different electrical specification and stricter timing requirements than I^2C, but it is usually interoperable (the differences are detailed in [2, Appendix B]).

The simplest command in SMBus is Quick Command, which is just an empty read or write (i.e. the read bit contains the single bit of data which is trans-

ferred). The zero-length reads used here are problematic (if they are sent to a slave which is not a Quick Command slave, an undefined condition can occur, thus they are disallowed in the high level master part of my model); but I assume it should not be a problem to read one dummy byte. Most commands include a command byte, which can distinguish different commands. For variable length reads or writes, a length byte is prepended to the data (thus it can be at most 255 bytes long). SMBus has an optional Packet Error Checking (PEC) feature, which is a CRC-8 byte appended at the end.

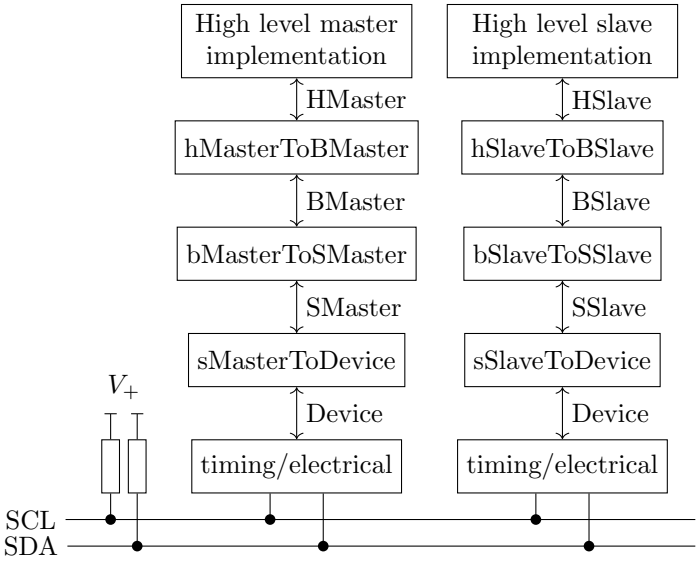
There can be a *host*, which is a slave with a special reserved address. It accepts SMBus Host Notify commands, where any device which supports this protocol can become a master to notify the host. Alternatively, devices can assert the optional SMBALERT# line. Then, the host, operating as a master, reads a byte from the Alert Response Address, to obtain the slave address of the device which caused the alert. Since multiple slaves could raise an alert at the same time, arbitration loss can also happen in slaves when using SMBus.

Additional features include the Address Resolution Protocol.

5.3 PMBus

PMBus [3] is a protocol for power management. It is built on top of an extended version of SMBus. The extensions to SMBus are the Group Command Protocol (multiple write commands to different devices in one big transaction, all are executed simultaneously at the STOP), and the Extended Command (which is used to obtain a second set of 256 command codes). PMBus is quite complex, it defines different data formats for temperatures/voltages/currents, defines the meaning of many commands and parameters/registers, and has extensive fault handling support.

Figure 2: An overview of the components of the model and the interfaces between them.



6 Modeling the I²C bus

In this section, I describe and discuss a model of the I²C bus. The model is implemented in Haskell and is runnable; there are sample slave and master implementations of an SMBus quick device and an EEPROM.

The model has three layers for both masters and slaves: The symbol layer, the byte layer and the high level layer. Note that the layers are not the same for masters and slaves, since only masters generate the clock signal and START/STOP signals.

Figure 2 shows an overview, the individual components will we explained in the following sections.

6.1 Timing/electrical layer

The timing/electrical layer is the interface between the model and the actual wires of the I²C bus. It is the same for both slaves and masters. This layer has to sample the state of the two wires (clock (SCL) and data (SDA)) at the right interval, and report it to the layer above, which is given as a `Device`. The `Device` then performs its functions and returns the next state to which the wires should be driven:

```
1 data BusState = BusState Bool Bool
2 type Device = DeviceState -> BusState -> (DeviceState,
      BusState)
```

(`DeviceState` contains the internal state of the `Device`.)

The timing/electrical layer abstracts over the analog behavior of the wires, which in the real world do not transition from one state to the other instantly. It is also responsible for ensuring that timing specifications are met, which are

defined in [1, p. 48-50]). For example, at any rising edge of SCL, it has to ensure that SDA is driven to its new value first, and SCL released only after the set-up time $t_{SU;DAT}$. This way, the layer also abstracts over the timing details, such that the layers above can operate in discrete time steps.

I created an implementation of this layer on top of a GPIO API (see section 11). Since it is not possible to guarantee a high enough sampling rate with the Haskell implementation, it does not support other masters attached to the bus externally. This is not an issue if only slaves are attached to the GPIOs, since the master generates the clock.

6.2 Symbol layer

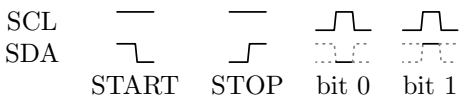
Moving up one level of abstraction, we have the symbol layer. There are four possible symbols: 0 bit, 1 bit, START and STOP. An additional idle symbol marks the absence of a symbol:

```
1 data BusSymbol = SymIdle | SymStart | SymStop | SymBit Bool
```

The master symbol layer is implemented in `sMasterToDevice`, its type is:

```
1 type SMaster = SMasterState -> BusSymbol -> (SMasterState,
      BusSymbol)
2 sMasterToDevice :: SMaster -> Device
```

The layer parses the symbol which appears on the bus lines:



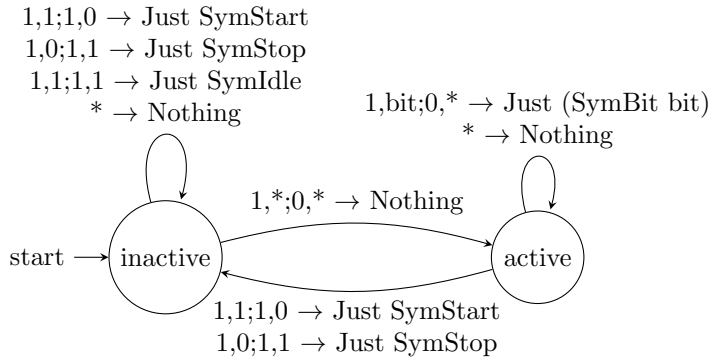
If one of these symbols is detected, the corresponding `BusSymbol` is given to the `SMaster`. If and only if the last symbol was STOP and both lines are high, the bus is idle and `SymIdle` is given. In any other case, the `SMaster` is not called.

The `SMaster` then returns the next symbol which should be generated on the bus.

There is some design freedom here, since the state of the SDA line can be anything during SCL low periods. The easiest thing to do is to just let SDA go high. One could argue that this uses less energy, since no current is flowing through the SDA pull-up resistor. On the other hand, the frequency at which SDA toggles is twice as high compared to other approaches, which makes it more prone to noise and interference problems. The alternative is to keep the SDA line at the state of the previous or next SCL high period. Using the previous state is still relatively simple. But for the next state, obviously we cannot look into the future, so we would have to generate the symbol to be given to the `SMaster` already before SCL goes low (which then gives us the next symbol to be generated). This complicates the design of the symbol parser. The advantage is that this aligns nicely with the non-zero setup time and zero hold time as specified in [1, p. 48].

There are certain rules that the `SMaster` has to follow, otherwise spurious symbols appear on the bus: If `SymStart` is followed by `SymStart` or `SymIdle`, an additional STOP appears. If `SymStop` or `SymIdle` is followed by `SymStop`, an additional START appears. If one or more `SymIdle` is sent between two

Figure 3: State machine diagram of the symbol reader. The symbol reader decides the next state and output value based on the combination of previous and current SCL,SDA values.



`SymBits`, an additional bit appears. The reason why this happens should be obvious: E.g. for the case of two START conditions, we have two transitions of SDA from high to low. This is only possible if there is a transition from low to high in-between, and that extra transition is detected as a STOP condition. In theory, it would be possible to always return SCL to low after each symbol, which would allow arbitrary sequences of all four symbols to be transmitted. However, this would not be I^{2}C anymore, since there, SCL must be high during idle periods.

The slave symbol layer has the following type:

```
type SSlave = SSlaveState -> BusSymbol -> (SSlaveState,
    Bool)
sSlaveToDevice :: SSlave -> Device
```

The symbol parsing side works the same as with the master. However, the transmitting side is much simpler, since the only thing a slave can do is turn the next 1 bit into a 0 bit by pulling SDA low. This is controlled by the `Bool` returned by the `SSlave`. When the slave is not transmitting, the value must be `True`.

I^{2}C slaves additionally have the ability to perform clock stretching when they need more time before processing further bytes. The generation of clock stretching is not yet implemented in the model, but it *accepts* clock stretching correctly.

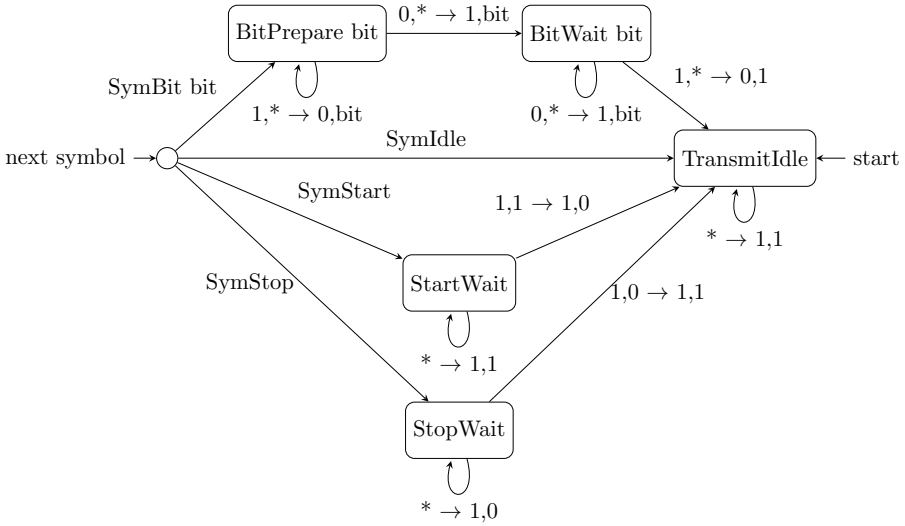
6.2.1 Implementation

Both the master and slave implementation use a common symbol reader function, which is depicted in figure 3:

```
data SymbolReaderState = SymbolReaderState Bool BusState
readSymbol :: SymbolReaderState -> BusState ->
    (SymbolReaderState, Maybe BusSymbol)
```

`sSlaveToDevice` simply calls the symbol reader, and sends it to the upper layer if there is a symbol. The upper layer then returns the SDA value. If there is no symbol, the last SDA value is used.

Figure 4: State machine diagram of `sMasterToDevice`. When the symbol reader sees a symbol, it is sent to the higher level, which returns the next symbol. Based on this, the next state is chosen. On the state transitions, the SCL,SDA value received from the lower level is on the left side of the →, and the returned value on the right side.



`sMasterToDevice` is more complex, since it has to be able to generate any bus symbol. It is displayed in figure 4. Each bus symbol is represented by a sequence of bus states, so the state machine generates these bus states, each time waiting until the desired bus state appears until proceeding to the next bus state. The symbol reader runs in parallel, and whenever it sees a bus symbol, it is sent to the higher level, which returns the next symbol to be generated.

6.3 Byte layer

In I^2C, bits always appear in groups of 8 (i.e. a byte) sent by the same device, followed by an acknowledge bit from the receiving side. The byte layer captures this aspect, by only allowing sending and receiving of bytes.

For the master, the interface is:

```
1 data BMasterAction = BActStart | BActStop | BActWrite Word8
      | BActRead | BActIdle
2 data BMasterResult = BResOk | BResNack | BResReadResult
      Word8 | BResArbitrationLost | BResUndefinedCondition
3 type BMaster = BMasterState -> BMasterResult ->
      (BMasterState, BMasterAction)
4 bMasterToSMaster :: BMaster -> SMaster
```

In each step, the `BMaster` gets the result of its previous action, and then returns the next action to be performed.

The byte layer detects arbitration losses (when 1 bit was sent but 0 bit received) and undefined conditions (any other unexpected case), in which case the result will be `BResArbitrationLost` or `BResUndefinedCondition` respectively.

Note that it is not always possible to detect undefined conditions, see section 5.1.1 for a discussion. The byte layer also keeps track of whether the bus is busy, and if so does not call the `BMaster`.

The following table shows an overview of possible results:

Action	Possible results
`BActStart`	`BResOk`, `BResUndefinedCondition`
`BActStop`	`BResOk`, `BResUndefinedCondition`
`BActWrite`	`BResOk`, `BResNack`, `BResArbitrationLost`, `BResUndefinedCondition`
`BActRead`	`BResReadResult`, `BResUndefinedCondition`
`BActIdle`	`BResOk`

There is a special case where any action directly after a `BActRead` may result in `BResArbitrationLost` or `BResUndefinedCondition`. This is because the byte layer first has to ask the `BMaster` whether more bytes should be read, before it can transmit the acknowledge bit, and that can have the above results.

The byte level master still has to follow the same sequencing rules as the symbol level master (with `SymBit` replaced by `BActRead` and `BActWrite`).

The slave byte layer looks as follows:

```
data BSlaveEvent = BEvStart | BEvStop | BEvReceive Word8 |
    BEvAck
data BSlaveReaction = BReactIdle | BReactTransmit Word8 |
    BReactReceive
type BSlave = BSlaveState -> BSlaveEvent -> (BSlaveState,
    BSlaveReaction)
bSlaveToSSlave :: BSlave -> SSlave
```

While the master gets back results for its actions, the slave receives events and then returns its reaction to the event.

There are three possible reactions: transmitting a byte, receiving a byte, and no reaction (`BReactIdle`). In the last case, the `BSlave` does not receive further events until the next START or STOP, and for `BEvReceive` the master will receive a NACK. `BEvAck` is emitted when a byte was transmitted, and the master acknowledged it.

When transmitting data, `bSlaveToSSlave` will detect arbitration loss and stop transmitting. This is needed for functionality such as the SMBus Alert Response Address. The model currently assumes that the slave does not need to know when arbitration loss happens, but this may not be the case for PMBus, due to its error handling functionality.

6.4 High level master layer

This is the high level master interface:

```
data MessageData = Read {
  size :: Integer,
  variable :: Bool
} | Write [Word8]
data Message = Message {
  msgAddress :: Address,
  msgData :: MessageData,
```

```
8     nonCritical :: Bool
9   }
10  data MessageDataReply = ReadReply [Word8] | WriteReply
        Integer
11  data MessageReply = MessageReply {
12    acked :: Bool,
13    msgDataReply :: MessageDataReply
14  }
15  type Transaction = [Message]
16  data TransactionReply = Success [MessageReply] |
        ArbitrationLost | UndefinedCondition
17
18  data HMaster = HMaster {
19    nextTransaction :: HMasterState -> (HMasterState, Maybe
          Transaction),
20    transactionReply :: HMasterState -> TransactionReply ->
          HMasterState
21  }
22  hMasterToBMaster :: HMaster -> BMaster
```

The high level master (e.g. a device driver) can issue transactions, and get back a transaction reply. In every step where the bus is free, `nextTransaction` is applied. Once it returns a transaction (rather than `Nothing`), it is executed, and at some point `transactionReply` will be applied; after that it repeats from the beginning.

A transaction is simply a list of read and write messages, which result in the corresponding list of read and write replies. A `WriteReply` indicates how many of the written bytes were acknowledged by a slave.

There are three additional requirements on transactions which are not expressed in the type system and are checked by `hMasterToBMaster`: The transaction cannot be empty [1, p. 14, note 5]. The `size` of read messages must be strictly positive (this is because we can only tell the device (via ACK/NACK) whether to transmit further bytes after having read the first byte). Finally, addresses must be between 0 and 127.

If the `variable` flag of a read message is set, the value of the first byte read is added to the read size. This is needed e.g. for SMBus Block Read; `size` would be set to 2 to enable the PEC byte, 1 otherwise.

`Message`s have a `nonCritical` flag. This influences how NACKs are handled. By default (`nonCritical=False`), if the address byte or any written byte is not acknowledged, the transaction is aborted and the rest of the messages are not executed. (The remaining un-executed messages will still get a corresponding `MessageReply` with `acked=False`.) This is important, as often later messages rely on earlier messages having executed successfully (e.g. first message sets address pointer, second message reads starting at this pointer). However, in some cases we need to override this default behavior. For example, the I^2C START byte [1, p. 19] is a read from address 0, which will not be acknowledged; but of course the rest of the transactions should still be executed. This can be done by setting `nonCritical=True`.

`hMasterToBMaster` does not do retries of transactions, this is the responsibility of the `HMaster`. It could also be implemented as an additional layer (e.g. `retryLayer :: Int -> HMaster -> HMaster`, the first parameter is the number of retries).

6.5 High level slave layer

A high level slave is defined as a set of functions, which are applied when certain events happen:

```
type Address = Integer

data HSlave = HSlave {
  slaveAddress :: HSlaveState -> Address -> Bool ->
      (HSlaveState, Bool),
  slaveRead :: HSlaveState -> (HSlaveState, Word8),
  slaveWrite :: HSlaveState -> Word8 -> (HSlaveState, Bool),
  slaveStop :: HSlaveState -> HSlaveState
}

hSlaveToDevice :: HSlave -> Device
```

The `Bool` passed to `slaveAddress` indicates whether the access is a read. The slave returns a `Bool` to indicate whether it wants to accept the transfer. If it does, an ACK is sent and either `slaveRead` or `slaveWrite` are called an arbitrary number of times. For `slaveWrite`, the slave can decide whether it wants to ACK the received byte. If at any point the slave does not ACK, or an arbitration loss is detected, no more `slaveRead` or `slaveWrite` events happen before the next `slaveAddress`.

6.6 Connecting the devices

So far we have seen the individual interfaces, and the functions which translate from higher to lower layers. It remains to actually connect different devices, so they can talk to each other. This connection can be done at any level.

At the lowest level, there is `globalStep`:

```
data GlobalState = GlobalState {
  devices :: [Device],
  deviceStates :: [DeviceState],
  busState :: BusState
}

globalStep :: GlobalState -> GlobalState
```

A `GlobalState` contains all the devices and their corresponding states, as well as the state of the two bus wires. The `globalStep` function advances the state by one step, by applying all devices with their current state and the bus state, and collecting the new states and output bus state. All bus states are then merged to one with a simple logical and operation. By iterating `globalStep` starting from an initial state, the bus can be simulated.

The connection can also be made at the symbol layer:

```
data SGlobalState = SGlobalState {
  sMasters :: [SMaster],
  sMasterStates :: [SMasterState],
  sSlaves :: [SSlave],
  sSlaveStates :: [SSlaveState],
  busSymbols :: [BusSymbol]
}
```

```
8
9 sGlobalStep :: SGlobalState -> SGlobalState
```

The principle is the same as before, but now masters and slaves are separate. However, the merging of bus symbols is more complicated. Depending on which combination of symbols from masters and SDA bits from slaves appears, there can be race conditions or even deadlocks. These situations are represented as a list of symbols with multiple elements and an empty list respectively. In the normal case, `busSymbols` contains just one element. A deadlock happens if a slave drives SDA low while no master sends a bit. A race condition occurs if one master sends a 1 bit, and another master sends a START or STOP symbol. This is called 'undefined condition' in the specification (see section 5.1.1). Currently `sGlobalStep` just picks the first element of the list. But the actual behavior is that this is chosen non-deterministically, it may even be that different devices see different symbols. Note that with a single master, undefined conditions are impossible, since the master cannot send two different symbols at the same time. However, with multiple masters, the model itself cannot prevent this; a possible solution (master codes) is discussed in section 5.1.1.

Finally, the function `hSlaveRunTransaction` allows to directly apply a transaction on a high level slave:

```
1 hSlaveRunTransaction :: HSlave -> HSlaveState ->
    Transaction -> (HSlaveState, TransactionReply)
```

Connecting the layers at the byte level should also be possible, however this was not implemented due to time constraints.

6.7 Design tradeoffs

The model consists of multiple layers, which has several advantages. First, it makes implementation easier (especially for the master side), as it is easier to implement several small parts than one big monolith. It creates the possibility to implement the lower layers in hardware and higher layers in software. Also, we gain flexibility, because a client which needs more freedom than the high level interface provides may use a lower level intermediary interface, rather than re-implementing everything up to the lowest level. It should also facilitate creating formal correctness proofs.

The high level master interface of the model for example restricts the ways a driver can react to device responses within the same transaction to what is possible with the available flags (such as `variable`). I hypothesize that the vast majority of device drivers can be implemented within these constraints. However, one could imagine scenarios such as atomic read-update-write, which are not possible with the high level master interface. Such drivers could either directly use the byte level interface together with locking. Alternatively, one could implement a separate layer on top of the byte layer with a much more flexible interface, e.g. using a bytecode language like eBPF to read and write individual bytes and perform computation.

The interfaces of the model are defined as function types, and thus the layers can be connected simply by function composition. This style has several advantages. It makes it relatively easy to implement the model, and to formally state properties about it. The function application implicitly provides synchronization and flow control, however only in one direction. For example, when the

byte level master layer is busy transmitting a byte, it simply does not call up to the higher level during this time, and thus the higher level cannot provide new bytes until the current byte is finished. This is not possible in the other direction, when the byte layer asks for the next byte, the upper layer is forced to return some value (see section 6.8 for how this could be supported). However, the function interface style also has some drawbacks. It puts the I^2C bus at the center of the whole system, which is not practical for implementations of real systems. Each layer has to keep track of the state of the layer above itself, this adds some complexity and is bug-prone, since it can happen that one forgets to replace the old state with the new state after applying the function of the higher layer.

6.8 Possible implementation of missing features

Some functionality has not been implemented in the model, this section discusses how these features could be implemented.

Clock stretching This feature allows devices to block the bus by holding SCL low. A device may need to do this if it needs more time before it can receive or transmit the next byte. For example, a memory chip may need multiple clock cycles to read a byte from non-volatile storage. It must stretch the clock until the byte arrives, if it did not, the master would read garbage data.

This could be implemented by allowing each layer (except `Device`) to return a 'clock stretch' value in place of an actual return value, e.g. by wrapping the return type with a `Maybe`. As an example, the symbol level slave interface would be changed to this:

```
1 type SSlave = SSlaveState -> BusSymbol -> (SSlaveState,
      Maybe Bool)
```

Each layer then has to pass this down, and at the lowest layer SCL has to be pulled low. For the high level master interface, this is not necessary, since there is no need to block the bus between transactions. It should not be very difficult to implement in the model, but may be a bit tedious, since it has to be supported in every layer.

Ten-bit addressing The I^2C specification defines a ten-bit addressing protocol, which can be useful if the seven-bit address space is too small [1, p. 15]. But this protocol just uses normal I^2C messages. The two most significant bits of the ten-bit address are encoded by choosing one of four seven-bit addresses reserved for this purpose. The remaining eight bits are then transmitted in the first data byte of a write message. For writes, the actual data bytes immediately follow. For reads, this one-byte read message is followed by a read message (in the same transaction), and this read message has the same seven-bit address as the write message had (thus it only carries the two most significant bits of the ten-bit address).

This can be implemented as a layer on top of the high level interface, simply by making the necessary transformation on the `Transaction`, and then again in the other direction for the `TransactionReply`. Since `Address` is defined as

`Integer`, the same data type could be used (however if one wants to mix seven-bit and ten-bit addresses, more thought needs to be put into how these should be distinguished).

Bus clear In the event that the SDA line gets stuck low, the I^2C specification recommends that the master should toggle SCL 9 times [1, p. 20].

This is an error recovery operation, if we need to perform it something has gone wrong already. It could for example be that a slave somehow missed a clock cycle during a write operation, and is now sending an ACK (pulling SDA low) while the master already tries to send a STOP. But since SDA is pulled low, the STOP condition is overriden. In this case, just toggling SCL 9 times would not actually help, since afterwards we would be in the exact same situation; the slave would think that a 0xff byte was written and would pull SDA low again for the ACK. Even worse, this byte that the slave thinks was written may overwrite important data or could cause arbitrary things to happen. So a better strategy would be to toggle SCL *up to* 9 times, until SDA goes high. Still, because something bad has already happened, it might be safer to reset all devices on the bus instead.

If one wanted to implement this feature anyway, it could be implemented as a separate `Device`. This bus recovery device would have to keep track of how long SDA has been low, and to generate the bus clear after a timeout.

7 Comparison to Linux API

In this section, the high level master interface of my model (section 6.4) is compared to the API implemented in Linux [7]. Similarities and differences are discussed, as well as limitations and additional features in Linux.

It turned out that two are very similar. In Linux, the kernel function `i2c_transfer`, and the `I2C_RDWR` ioctl for userspace, take a list of `i2c_msg` as an argument. Each `i2c_msg` has an address, a length, a data buffer, and a flags field. This corresponds to `Message` in the model, with the `msgAddress` field. For reads, the length is in `size`, and the data buffer maps to the contents of `ReadReply`. For writes, the data buffer corresponds to the content of `Write`, which also implicitly gives the length.

The following flags are available in Linux [5]:

`I2C_M_RD`: Defines the message as read or write. In the model, this is corresponds to a `Read` or `Write` in `msgData`.

`I2C_M_TEN`: The address is a 10 bit address. This can be implemented as a layer on top of the model, see section 6.8.

`I2C_M_DMA_SAFE`: Defines the buffer as DMA safe. Not relevant for the model.

`I2C_M_RECV_LEN`: Add the value of the first read byte to the length (only for reads). This maps directly to the `variable` flag in the model.

`I2C_M_NO_RD_ACK`: Skip the ACK/NACK bit after each read byte. This is used by just one driver for a non-I^{2}C-compliant device (KS0127).

`I2C_M_IGNORE_NAK`: Continue with a message even if NACKs are received.

`I2C_M_REV_DIR_ADDR`: Inverts the read/write bit.

`I2C_M_NOSTART`: Skips the repeated start and address byte. This can be used as a performance optimization to avoid copying multiple segments of a message into one big buffer. But it can also be used to talk to non-I^{2}C devices which require direction changes in the middle of a message.

`I2C_M_STOP`: Insert a STOP after this message, even when it is not the last one of the transaction. It is unclear to me why this flag exists, as it seems that in most cases, one could just send two transactions instead.

The Linux I^{2}C API does not have an equivalent of the `nonCritical` flag. When a written byte is not acknowledged, the bitbanging driver just returns an error (`EIO`) with no additional information, whereas in my model, the field in `WriteReply` specifies how many bytes were acknowledged. When an address byte is not acknowledged, it will be retried some number of times, sending STOP, START and the address byte again [8, around line 345]. This is problematic, as sending a STOP breaks the current transaction. For example, in PMBus, a transaction that starts with a read message is a Data Content Fault [3, section 10.9.1]. However, the address byte will still be acknowledged (and remember that the slave has no way to NACK the read bytes, as these ACK bits are sent by the master). Thus, assuming a transaction with a write and a read message is sent to a PMBus device, and the address byte of the read message is retried for some reason; then the read silently fails (producing all `0xff` bytes) with no error returned by the Linux API.

Linux also has an SMBus API, which is a subset of the I^{2}C API and recommended to be used whenever possible. This is because some bus master interfaces only support these commands, on the other hand there exists a function (`i2c_smbus_xfer_emulated`) which translates SMBus calls to I^{2}C calls [4].

The Linux SMBus API is limited to reading or writing at most 32 bytes. In SMBus 3.0, the limit was raised to 255 bytes [2, p. 85, D.3.6], but this change has not been implemented in Linux so far.

Linux also has support for I^2C multiplexers [6]. These multiplexers are basically switches which connect or disconnect bus segments, and can be operated either over I^2C itself or other means. On the software side in Linux, these bus segments appear as separate virtual I^2C master interfaces. When a driver accesses this interface, the multiplexer driver first operates the switches, and then redirects the I^2C transaction to the actual master interface (which is now connected to the corresponding bus segment).

8 Non-standard devices

Some slave devices do not conform to the specification. Driver developers may not be able to replace these components, and just have to deal with it.

One approach could be to also expose lower level APIs to driver developers (such as the byte level or even symbol level interface in the model).

The approach taken in Linux is to add a set of flags to the master interface which can be set to enable specific non-standard behaviors (see section 7). By combining the `I2C_M_REV_DIR_ADDR` and `I2C_M_NOSTART` flags, linux drivers basically gain the same flexibility as if they had access to the byte level API in the model, with the exception that the first byte after a START must be a write, and the limitations on reacting to device responses within a transaction still apply. The first byte after a START can be controlled fully by setting the address and `I2C_M_REV_DIR_ADDR` flags as necessary, and for following bytes the direction can be changed arbitrarily by adding messages with the `I2C_M_NOSTART` flag. An example of this is shown below (AS5011). Other flags allow some modifications at the symbol level too.

It may well be that this is the most practical solution, since hardware master interfaces also offer limited flexibility, and the kernel interface can only expose the features that the hardware interface has. But it is probably impossible to have a complete set of flags for every possible non-compliant behavior, and not all hardware interfaces support the available flags. If a device driver needs more flexibility, it would need to have its own bitbanging implementation, but this may not be possible if the bus is not available via GPIOs, and also creates problems if there are other devices on the same bus.

In the following, some examples are described and discussed:

24AA16 EEPROM This chip (datasheet at [13]) uses 8 addresses rather than just one. The three least significant bits of the bus address are the most significant bits of the data address. It is not entirely clear if this violates the specification, but it will be a problem if e.g. an access control layer assumes that each device has exactly one address.

AS5011 This hall sensor IC (datasheet at [14]) uses a non-compliant format for register read operations. After the address, the master writes the register address and then reads the data byte, without a repeated START in-between [14, p. 8]. This is only compatible up to the byte layer of the model. In Linux, the following code is used for this operation [9]:

```
uint8_t data[2] = { aregaddr };
struct i2c_msg msg_set[2] = {
  {
    .addr = client->addr,
    .flags = I2C_M_REV_DIR_ADDR,
    .len = 1,
    .buf = (uint8_t *)data
  },
  {
    .addr = client->addr,
```

```
        .flags = I2C_M_RD | I2C_M_NOSTART,
        .len = 1,
        .buf = (uint8_t *)data
      }
    };
```

KS0127 According to a comment in the Linux kernel source, this video decoder chip has a bug [10]:

```
/* We need to manually read because of a bug in the KS0127 chip.
 *
 * An explanation from kayork@mail.utexas.edu:
 *
 * During I2C reads, the KS0127 only samples for a stop condition
 * during the place where the acknowledge bit should be. Any standard
 * I2C implementation (correctly) throws in another clock transition
 * at the 9th bit, and the KS0127 will not recognize the stop condition
 * and will continue to clock out data.
 *
 * So we have to do the read ourself.  Big deal.
 *    workaround in i2c-algo-bit
 */
```

This bug affects the byte layer; it has to skip the NACK bit after the last read byte and immediately emit the STOP symbol. In Linux, the `I2C_M_NO_RD_ACK` flag is used.

CAT5259 This digital potentiometer (datasheet at [15]) uses 8-bit slave addresses instead of the 7-bit addresses defined in the I^2C specification. The second byte is always written by the master, and contains an opcode which indicates if following bytes are read or write. This is completely incompatible with the high level interface. But it still seems to be compatible with the byte level interface; data is transferred in 8-bit bytes and each byte is followed by an ACK bit from the side which received the byte.

However, there is also a special Increment/Decrement Command. With this command, the master sends an arbitrary number of 0 or 1 bits (used to fine-tune the wiper setting), and a STOP at the end. At this point, even the byte level interface is violated, only the symbol level remains compatible. Also note that it does not make sense to use this command in a transaction, since in order to effectively use it, the driver has to emit individual 0 or 1 bits and observe if the target has been reached.

Arguably, the interface of this device deviates so strongly from the I^2C specification that it should not be called I^2C, but the datasheet does it anyway [16].

9 Comparison to a hardware master interface

Commonly, hardware I^2C master interfaces are used, this frees up CPU cycles compared to a bitbanging implementation. By comparing the interface of the model implementation to an existing hardware interface, we can find potential problems and limitations in both.

Typically, these interfaces operate at the byte layer. In some cases, they have FIFO buffers which enable transfer of multiple bytes at once. With DMA, it could even be imaginable to implement the complete high level interface in hardware.

In this section, I describe how the interface of the Opencores I^2C master core [17] compares to the byte level interface of the model. This analysis is based only on the documentation of the core.

In general, the two interfaces correspond relatively closely.

The command register allows to generate START, STOP conditions, and to read and write bytes. This matches the `BMasterAction` type (see section 6.3). Note that `BActIdle` is not needed, because the high level master can simply choose not to send a command to the core (which is not the case in the model). Also, the core has to support clock stretching (since it cannot force the layer above to provide a command), which the model implementation currently does not support (see section 6.8).

The status register roughly corresponds to `BMasterResult` in the model. It is possible to detect NACKs and arbitration losses through the register. There seems to be no way to detect undefined conditions, but that is not essential for the operation of the bus; ideally these should not happen anyway.

The actual data bytes are read and written through the separate transmit and receive registers.

A minor difference is that the START command has to be sent at the same time as the first read or written byte, unlike in the model, where these are separate. But there is another, crucial difference: For read commands, the driver has to say whether the byte should be ACKed. In other words, it has to know *in advance* whether the next byte that it reads will be the last byte. This makes it impossible to implement variable length reads, since there we do not know that in advance: If the length byte is 0, it was the last byte, in any other case more bytes need to be read. Note that in the model, we do not have to provide this information; instead it is provided implicitly in whether the next command is again a byte read or a START or STOP condition. Consequently, the Linux driver for this master core [11] does not support the `I2C_M_RECV_LEN` flag, and the byte level interface of my model is not compatible with this master core interface.

In conclusion, we have found that the byte level interface of the model corresponds quite closely to an existing hardware interface, which indicates that it makes sense to use this model interface as the interface between hardware and software. But we also found a severe limitation in the hardware interface, which may not have been obvious just from reading the documentation.

10 QuickCheck properties

QuickCheck is a testing library for Haskell programs [20]. I have written two properties of the model in the form of QuickCheck tests. These properties generally have the form $\forall x_1 \in D_1, \ldots, x_n \in D_n : P(x_1, \ldots, x_n)$, where P is the boolean property which should be true for all parameters in the given domains D_i. QuickCheck provides a powerful way of specifying these domains, even generating arbitrary functions of a given type is possible. When running the tests, the QuickCheck library randomly samples elements from the parameter domains and evaluates the property for these parameters. By default, this is repeated 100 times.

10.1 No errors with a single master

This property states that, given a list of transactions which are executed sequentially by a single master, and a set of arbitrary slaves connected to the bus, no arbitration losses or undefined conditions occur while the transactions are executed. At the same time, the property also checks that all transactions complete within a certain number of device level steps (this number is calculated from the list of transactions).

This already provides quite strong security and liveness properties, but does not yet say that the data itself is transmitted correctly.

One caveat is that the model currently does not allow clock stretching. In a system where it is allowed, it would also be necessary to prove that all devices will always stop clock stretching within a finite or fixed amount of time in order to guarantee liveness.

Also, in a real system there is usually always some small probability of errors due to noise affecting the bus wires.

As already mentioned in section 5.1.1, the property does not hold in general for multiple masters connected to the same bus.

10.2 No observable difference when connecting at different layers

Building on the previous property, it still remains to show that the data itself is received correctly. However, it is not trivial to formally define this property. The approach which is taken here goes as follows: We take an arbitrary list of transactions and an arbitrary slave as input. The transactions are then executed twice: Once with the complete stack of layers and using `globalStep`, and once with `hSlaveRunTransaction`, which directly applies the transaction on the high level slave (see section 6.6). During both executions, logs of the transaction replies and everything the slave sees are collected, and these logs are then compared for equality. This way, `hSlaveRunTransaction` serves as a specification for what a slave should observe when a given transaction is applied, and what the transaction reply should be.

There is an additional property which compares the logs between connecting a single master and multiple slaves at the device and at the symbol level.

10.3 Results

By runnning the QuickCheck tests, a bug was found in the model implementation, more exactly in `bSlaveToSSlave`. The original code looked like this:

```
1     idleState = (SSlaveState bState SSlStIdle, True)
2     ...
3     handleReceiveEvent bEvent =
4       let (newBState, bReaction) = bSlave bState bEvent in
5       if bReaction == BReactIdle then idleState
6       else (SSlaveState newBState (SSlStReceiveAck
          bReaction), False)
```

The bug here was that if the call to `bSlave` returned `BReactIdle`, the code did not update the higher level state to `newBState`, but continued to use the old `bState` (which was obscured because this was referenced indirectly through the `idleState` binding). To fix it, line 5 was replaced by the following:

```
1       if bReaction == BReactIdle then (SSlaveState
          newBState SSlStIdle, True)
```

11 EEPROM model

I implemented a model of an I^{2}C EEPROM in Haskell, based on the 24AA256 datasheet [12].

The EEPROM can store 32 KB of data in non-volatile storage. It has a 15 bit address register, which can be set with a 2-byte write message. Read messages can be arbitrarily long. For each byte which is read, the byte pointed to by the address register is returned, and the address register is incremented by one (wrapping around at the end of the EEPROM). Writes do not go directly to the non-volatile storage, since that is only writable in whole 64-byte pages. Writes are done using write messages longer than 2 bytes, the first two set the address, the following bytes are written to a page buffer, wrapping around at 64-byte-aligned addresses. Only after the following STOP, the page buffer is copied to the non-volatile storage. This operation takes up to 5 ms, and the EEPROM does not respond to commands during this time.

The model is parameterized on the bus address, the data address size in bits (15 for the 24AA256), and the write page size in bytes (64 for the 24AA256).

The state is defined as follows:

```
1 data HSlaveState = EepromSlaveState Int
      EepromSlaveWriteState [Word8]
2 data EepromSlaveWriteState = EepromAddress Int |
      EepromWrite [Word8] | EepromRead
```

It contains the data address register (as an `Int`), the write state, and of course the contents of the EEPROM itself. The write state keeps track of where the next written byte will be stored. In the `EepromAddress` state, it will be shifted into the data address register. In the `EepromRead` state, when a byte is written, the page in which the address pointer lies is copied to the page buffer (represented by `EepromWrite`), and the byte is then written there. While in the `EepromWrite`, after each written byte the address is incremented, wrapping around at the page boundary.

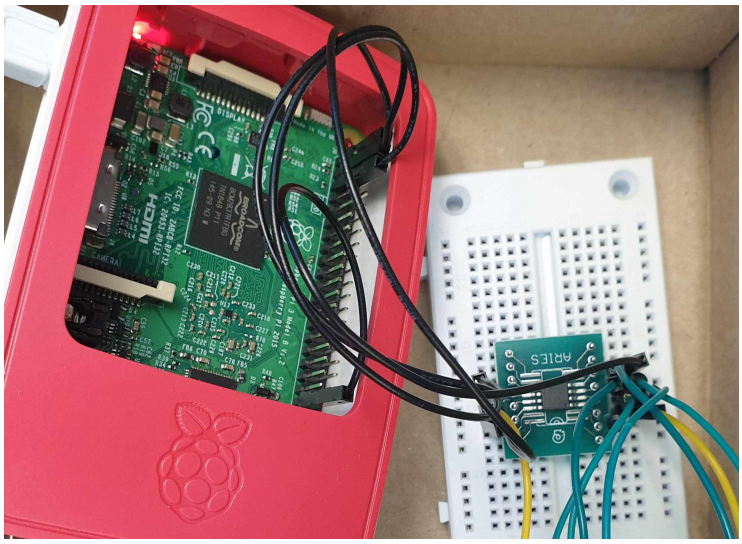
The write protect pin is not modeled. Also, the page write time of 5 ms, during which no commands are acknowledged, is not implemented in the model, which would be difficult since the model does not contain a notion of time.

The model was then tested against an actual EEPROM through random testing. First, the EEPROM was filled with random data, so that potential errors would be more likely to be detected. The initial contents of the model EEPROM were set to the same data. Then, using the QuickCheck library, random transactions of increasing size, but only to the bus address of the EEPROM, were generated and executed on both the real and the model EEPROM. The transaction replies were compared, and at the end the final contents of the EEPROM were read out and compared to the final contents of the model.

The EEPROM was connected to the GPIO pins of a Raspberry Pi (as shown in figure 5), and a Haskell module was written which connects the I^{2}C model to these GPIOs. To avoid generating spurious START or STOP conditions when both wires change at the same time, the module always sets SDA first is SCL is high and SCL first if it is low.

As a result of the testing, some differences were discovered and fixed in the EEPROM model. After testing again, no differences were observed anymore.

Figure 5: An EEPROM was connected to the GPIO pins of a Raspberry Pi for testing the model.



The first difference was that when a data write is followed by a repeat START instead of a STOP, the write page buffer is discarded rather than being written back to the EEPROM storage. Initially, the model did not contain a page buffer, but directly wrote bytes to the EEPROM contents.

The second difference was in the way incomplete writes to the address register are handled. Through experimentation, it was determined that the address register is a shift register. After an incomplete address write, an additional bit is shifted in, which is 1 if the write is followed by a repeat START, 0 for a STOP.

Both these differences only occur when using transactions different from the formats documented in the datasheet. Instead of adjusting the model, another option would be to restrict the domain of allowed transactions to the documented ones. Especially for more complex devices, this option may be preferable, since otherwise we are basically modeling unspecified or undefined behavior.

12 Access control on an EEPROM

Ultimately, we want to be able to give clients fine-grained rights to perform certain operations on devices attached to an I^2C bus as a slave. In general, which potential rights exist depends on the type of device. In this section we start by looking at the rights on an EEPROM.

We define a right not by what transactions it allows on the bus, but rather by what is allowed to happen on the EEPROM itself, which should be much simpler. This is called an "abstract right", and it is defined as a binary relation between "abstract states". We then additionally define a set of allowed transactions for the abstract right, and prove, using a process called refinement, that the effects of these transactions on the abstract state are contained in the abstract right. Since the transactions operate on the "concrete state", we need a "lifting" function which extracts the abstract state from the concrete state.

First, I define the abstract state of the EEPROM and the abstract rights. For now I only consider writes, for read rights one would have to additionally keep track of which bytes were read (or alternatively state that the transaction reply is independent of the initial value of the bytes for which reading is not allowed).

The abstract state is simply a vector of bytes, i.e. an element of $AS := \{0, \ldots, 255\}^N$ where N is the size in bytes of the EEPROM. An abstract write right is described as a set $WR \subseteq \{0, \ldots, N-1\}$, only those bytes whose index is contained in WR are allowed to be written to. Formally, the relation of allowed state transitions for a given WR is: $\{a \times b \in AS \times AS \mid a_i = b_i \; \forall i \in \{0, \ldots, N-1\}, i \notin WR\}$.

The concrete state on the other hand additionally contains the data address register and some state to parse bus messages, see section 11. For the concrete right, we need to know the bus address of the EEPROM and the page size P, in addition to N (which is always a power of 2 in the model). The set of allowed transactions contains all transactions with just a single write message. Such a write message will contain a data address a, followed by n data bytes. It must hold that $\{a, a-(a \mod P)+((a+1) \mod P), \ldots, a-(a \mod P)+((a+n-1) \mod P)\} \subseteq WR$ for the transaction to be allowed. This rather complicated formulation comes from the fact that writes wrap around at the page buffer boundary.

Unfortunately, due to time constraints this direction of the work was not continued. However, it may be useful as a starting point for future work.

13 Conclusion

In this thesis, we have seen a model implementation of the I^2C bus in Haskell. The model has four layers of interfaces, from the low level device interface which is shared between masters and slaves, to the symbol and byte level interfaces, up to the high level master and slave interfaces. State machines build the connection between lower and higher levels.

The model allows us to formally state end-to-end correctness properties between the high level master and slave interfaces, two such properties were presented.

The analysis of the Linux I^2C API shows that it is very similar to the high level master API of the model. This provides validation that the interface is useful in practice and not too restrictive.

The work of building a model of an EEPROM, connecting the master side of the I^2C model to an actual EEPROM chip, and comparing the two by random testing provides evidence that the model implementation works in practice and is compatible with the existing implementation in the EEPROM.

We have seen that there exist devices whose slave interfaces violate the I^2C specification in various ways. To be able to work with these devices, drivers either need to access the bus at lower levels, or the master interfaces have to expose the right set of flags to enable non-standard behavior. At the same time, existing hardware master interfaces may have limitations which break functionality that actually is compliant. As a result, we have no guarantees that a given combination of hardware master interface and slave device is compatible. If a formally verified HDL implementation of I^2C could be created, which guarantees compatibility thanks to end-to-end proofs, and hardware developers convinced to use it in their designs, this could help reduce the extent of this problem.

The model implementation has some limitations. Because it is implemented in Haskell, it is deterministic and thus does not encompass every correct implementation of the specification. Ideally, we would like to define correctness properties over all possible implementations, because the slave devices attached to the bus may use any such implementation. On the other hand, the specification does not actually leave much freedom, so this is maybe not as important.

14 Future work

It still remains to actually prove the properties which were presented in section 10. The property in section 10.2 only considers a single master and slave, this should be extended to multiple slaves, and potentially to multiple masters (but this would be quite tricky).

It is also necessary to prove that the behavior of the actual hardware and software implementation match the model. To this end, one could try to prove this for an existing implementation. This proof would also serve as evidence that the model is compatible with existing implementations, or could uncover bugs if it fails.

A different idea is to directly translate (the lower layers of) the model to a hardware description language (HDL). This was not investigated in detail, however it seems quite plausible that this is possible. There already exist compilers such as Clash or the FHW project [25] which can translate Haskell to HDLs. All parts of the model, with the exception of the high level master part, are state machines with fixed size states, inputs and outputs, and do not use features which are challenging for synthesis, like recursion. One challenge might however be the way the different layers are connected through function composition. This may need to be turned into some interface with valid/ready handshaking.

Related to this, clock stretching is not yet implemented in the model, I describe in section 6.8 how this could be done.

On a higher level, it should be possible to build access control functionality on top of the model, and to make refinement proofs which relate abstract rights defined on a slave device itself to the transactions allowed by the right. For example, a driver could be restricted to only writing certain bytes in an EEPROM by restricting which transactions it can issue. Work in this direction was started in section 12, which could be continued in future work.

References

[1] I^{2}C-bus specification and user manual, Rev. 6, 2014. https://www.nxp.com/docs/en/user-guide/UM10204.pdf Accessed: 2020-05-11.

[2] System Management Bus (SMBus) Specification, Version 3.1, 2018. http://www.smbus.org/specs/SMBus_3_1_20180319.pdf Accessed: 2020-07-28.

[3] PMBus - Power System Management Protocol Specification - Part II - Command Language, Rev. 1.2, 2010. https://pmbus.org/Assets/PDFS/Public/PMBus_Specification_Part_II_Rev_1-2_20100906.pdf Accessed: 2020-07-28.

[4] Linux kernel documentation: The SMBus Protocol. https://www.kernel.org/doc/html/latest/i2c/smbus-protocol.html Accessed: 2020-05-12.

[5] Linux kernel documentation: The I2C Protocol. https://www.kernel.org/doc/html/latest/i2c/i2c-protocol.html Accessed: 2020-07-28.

[6] Linux kernel documentation: I2C muxes and complex topologies. https://www.kernel.org/doc/html/latest/i2c/i2c-topology.html Accessed: 2020-08-24.

[7] Linux kernel: i2c.h - definitions for the i2c-bus interface. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/uapi/linux/i2c.h?h=v5.8.3 Accessed: 2020-08-24.

[8] Linux kernel: i2c-algo-bit.c - i2c driver algorithms for bit-shift adapters. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/i2c/algos/i2c-algo-bit.c?h=v5.8.3 Accessed: 2020-08-24.

[9] Linux kernel: as5011.c - Driver for Austria Microsystems joysticks AS5011. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/input/joystick/as5011.c?h=v5.8.3 Accessed: 2020-09-07.

[10] Linux kernel: ks0127.c - Video Capture Driver. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/media/i2c/ks0127.c?h=v5.8.3 Accessed: 2020-09-07.

[11] Linux kernel: i2c-ocores.c - I2C bus driver for OpenCores I2C controller. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/i2c/busses/i2c-ocores.c?h=v5.8.3 Accessed: 2020-09-07.

[12] Microchip 24AA256/24LC256/24FC256, 256K I^{2}C Serial EEPROM, 2019. http://ww1.microchip.com/downloads/en/DeviceDoc/24AA256-24LC256-24FC256-Data-Sheet-20001203W.pdf Accessed: 2020-06-29.

[13] Microchip 24AA16/24LC16B/24FC16, 16K I^{2}C Serial EEPROM, 2019. http://ww1.microchip.com/downloads/en/DeviceDoc/20001703M.pdf Accessed: 2020-09-07.

[14] Austria Microsystems AS5011, Low power Integrated Hall IC for human interface applications, 2009. http://www1.futureelectronics.com/doc/AUSTRIAMICROSYSTEMS/AS5011.pdf Accessed: 2020-09-07.

[15] ON Semiconductor CAT5259, Quad Digital Potentiometer (POT) with 256 Taps and I^2C Interface, 2013. https://www.onsemi.com/pub/Collateral/CAT5259-D.PDF Accessed: 2020-09-07.

[16] Arduino Forum: 8bit i2c - non standard slave device with no r/w bit. https://forum.arduino.cc/index.php?topic=664624.0 Accessed: 2020-09-07.

[17] Richard Herveille. I^2C-Master Core Specification, Rev. 0.9, 2003. https://opencores.org/websvn/filedetails?repname=i2c&path=%2Fi2c%2Ftrunk%2Fdoc%2Fi2c_specs.pdf Accessed: 2020-09-07.

[18] Shravani Balaraju. UVM verification of an I2C Master Core, 2019. Master's thesis, Rochester Institute of Technology. https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11223&context=theses

[19] Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, Timothy Roscoe. A Least-Privilege Memory Protection Model for Modern Hardware, 2019. https://arxiv.org/abs/1908.08707

[20] QuickCheck: Automatic testing of Haskell programs. https://hackage.haskell.org/package/QuickCheck Accessed: 2020-08-26.

[21] Christoph Berg, Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach. Formal Verification of the VAMP Microprocessor - Project Status, 2002. https://conferences.mpi-inf.mpg.de/elics02/report/berg.ps

[22] CompCert C verified compiler. http://compcert.inria.fr Accessed: 2020-09-05.

[23] The seL4 Microkernel. https://sel4.systems Accessed: 2020-09-05.

[24] Wikipedia: Display Data Channel. https://en.wikipedia.org/wiki/Display_Data_Channel Accessed: 2020-09-08.

[25] Stephen A. Edwards with Martha A. Kim, Richard Townsend, Kuangya Zhai, and Lianne Lairmore. The FHW Project: High-Level Hardware Synthesis from Haskell Programs, 2019. http://www1.cs.columbia.edu/~sedwards/papers/edwards2019fhw.pdf

A Haskell implementation of the model

```
1  -- I2C Bus Model
2  module Model where
3  import Data.Maybe
4  import Data.Word
5  import Data.Bits
6
7  ------------------------------------------------------------
8  -- Interface and type definitions
9  ------------------------------------------------------------
10
11 data BusState = BusState Bool Bool deriving (Eq, Show) --
       SCL, SDA
12 data BusSymbol = SymIdle | SymStart | SymStop | SymBit Bool
       deriving (Eq, Show)
13 type Address = Integer -- more correctly uint7 (but this
       allows to use the same data structures for 10-bit
       addressing)
14
15 -- Device interface
16 type Device = DeviceState -> BusState -> (DeviceState,
       BusState)
17 data DeviceState
18   = DirectSlaveDeviceState HSlaveState SlaveState BusState
19   | SlaveDeviceState SSlaveState Bool SymbolReaderState
20   | MasterDeviceState SMasterState LMasterTransmitState
       SymbolReaderState
21   deriving (Show)
22
23 -- Symbol level slave interface
24 type SSlave = SSlaveState -> BusSymbol -> (SSlaveState,
       Bool)
25 data SSlaveState = SSlaveState BSlaveState SSlaveImplState
       deriving (Show)
26
27 -- Byte level slave interface
28 type BSlave = BSlaveState -> BSlaveEvent -> (BSlaveState,
       BSlaveReaction)
29 data BSlaveState = BSlaveState HSlaveState Bool deriving
       (Show)
30 data BSlaveEvent = BEvStart | BEvStop | BEvReceive Word8 |
       BEvAck deriving (Show)
31 data BSlaveReaction = BReactIdle | BReactTransmit Word8 |
       BReactReceive deriving (Eq, Show)
32
33 -- High level slave interface
34 data HSlave = HSlave {
35   slaveAddress :: HSlaveState
36            -> Address
37            -> Bool -- is read
38            -> (HSlaveState, Bool), -- ack
39   slaveRead :: HSlaveState -> (HSlaveState, Word8),
```

```
40    slaveWrite :: HSlaveState
41          -> Word8
42          -> (HSlaveState, Bool), -- ack
43    slaveStop :: HSlaveState -> HSlaveState
44  }
45  data HSlaveState
46    = QuickSlaveState Bool
47    | EepromSlaveState Int EepromSlaveWriteState [Word8]
48    | CounterSlaveState Integer
49    | LoggingSlaveState HSlaveState [SlaveLogEntry]
50    deriving (Show)
51
52  -- Symbol level master interface
53  type SMaster = SMasterState -> BusSymbol -> (SMasterState,
        BusSymbol)
54  data SMasterState = SMasterState BMasterState
        SMasterImplState deriving (Show)
55
56  -- Byte level master interface
57  type BMaster = BMasterState -> BMasterResult ->
        (BMasterState, BMasterAction)
58  data BMasterState = BMasterState HMasterState (Maybe
        BMasterTransaction) deriving (Show)
59  data BMasterAction = BActStart | BActStop | BActWrite Word8
        | BActRead | BActIdle deriving (Eq, Show)
60  data BMasterResult = BResOk | BResNack | BResReadResult
        Word8 | BResArbitrationLost | BResUndefinedCondition
        deriving (Eq)
61
62  -- High level master interface
63  data HMaster = HMaster {
64    nextTransaction :: HMasterState -> (HMasterState, Maybe
        Transaction),
65    transactionReply :: HMasterState -> TransactionReply ->
        HMasterState
66  }
67  data HMasterState
68    = CounterMasterState Integer TransactionReply
69    | LoggingMasterState HMasterState [Maybe TransactionReply]
70    deriving (Show)
71
72
73  type Transaction = [Message]
74  data Message = Message {
75    msgAddress :: Address,
76    msgData :: MessageData,
77    nonCritical :: Bool
78  } deriving (Show, Read)
79  data MessageData = Read {
80    size :: Integer,
81    variable :: Bool
82  } | Write [Word8] deriving (Show, Read)
83
84  data TransactionReply
```

```
  = Success [MessageReply] | ArbitrationLost |
      UndefinedCondition
  deriving (Eq, Show)
data MessageReply = MessageReply {
  acked :: Bool,
  msgDataReply :: MessageDataReply
} deriving (Eq, Show)
data MessageDataReply = ReadReply [Word8] | WriteReply
    Integer deriving (Eq, Show)


------------------------------------------------------------
-- Model implementation, translating from higher to lower
    levels
------------------------------------------------------------

data SlaveState
  = SlStIdle
  | SlStStart
  | SlStRcvAddress Integer Address
  | SlStAck Bool
  | SlStReadAck
  | SlStRead Integer Word8
  | SlStWrite Integer Word8
  deriving (Show)

-- Translate a high level slave to a low level bus device
hSlaveToDeviceDirect :: HSlave -> Device
hSlaveToDeviceDirect hSlave (DirectSlaveDeviceState hState
    state prevBusState) busState = let
    busTrans = (prevBusState, busState)
    (newHState, newState) =
      if busTrans == busStop then (slaveStop hSlave hState,
          SlStIdle)
      else if busTrans == busStart then (hState, SlStStart)
      else if isClockDown busTrans then (
        let (BusState _ bit) = prevBusState in
        case state of
          SlStIdle -> (hState, SlStIdle)
          SlStStart -> (hState, SlStRcvAddress 0 0)
          SlStRcvAddress count addr ->
            if count == 7 then
              let (newHState, ack) = slaveAddress hSlave
                  hState addr bit in
              (newHState, if ack then SlStAck bit else
                  SlStIdle)
            else
              (hState, SlStRcvAddress (count + 1) (addr * 2
                  + (boolToInt bit)))
          SlStAck True ->
            let (newHState, byte) = slaveRead hSlave hState
                in
            (newHState, SlStRead 0 byte)
          SlStReadAck ->
```

```
            if bit then (hState, SlStIdle)
            else
              let (newHState, byte) = slaveRead hSlave
                  hState in
              (newHState, SlStRead 0 byte)
          SlStAck False -> (hState, SlStWrite 0 0)
          SlStRead count byte ->
            -- check for arbitration loss, required for
                SMBus ARP Get UDID
            -- Assumption: High level slaves don't need to
                know when arbitration loss happens.
            if (nthBit byte count) && not bit then (hState,
                SlStIdle)
            else if count == 7 then (hState, SlStReadAck)
            else (hState, SlStRead (count + 1) byte)
          SlStWrite count byte ->
            let newByte = byte * 2 + (boolToInt bit) in
            if count == 7 then
              let (newHState, ack) = slaveWrite hSlave
                  hState newByte in
              (newHState, if ack then SlStAck False else
                  SlStIdle)
            else (hState, SlStWrite (count + 1) newByte)
      ) else (hState, state)
    busOp = case newState of
      SlStAck _ -> busTransmit
      SlStRead count byte ->
        if not (nthBit byte count) then busTransmit else
            busIdle
      _ -> busIdle
  in
    (DirectSlaveDeviceState newHState newState busState,
       busOp)

busStart = (BusState True True, BusState True False)
busStop = (BusState True False, BusState True True)
isClockDown (BusState scl1 sda1, BusState scl2 sda2) = scl1
    && not scl2
busIdle = BusState True True
busTransmit = BusState True False

boolToInt False = 0
boolToInt True = 1

nthBit :: Word8 -> Integer -> Bool
nthBit byte n = testBit byte (7 - (fromInteger n))

data SymbolReaderState = SymbolReaderState Bool BusState
    deriving (Show)

-- Shared symbol reader
readSymbol :: SymbolReaderState -> BusState ->
    (SymbolReaderState, Maybe BusSymbol)
readSymbol (SymbolReaderState rcvActive prevBusState)
```

```
    busState = let
173   busTrans = (prevBusState, busState)
174   (busSymbol, nextRcvActive) =
175     if busTrans == busStop then (Just SymStop, False)
176     else if busTrans == busStart then (Just SymStart, False)
177     else if isClockDown busTrans then
178       let BusState _ bit = prevBusState in
179       (if rcvActive then Just (SymBit bit) else Nothing,
            True)
180     else if busState == busIdle && not rcvActive then (Just
          SymIdle, False)
181     else (Nothing, rcvActive)
182   in
183     (SymbolReaderState nextRcvActive busState, busSymbol)
184
185 -- Translate a symbol level slave to a low level bus device
186 sSlaveToDevice :: SSlave -> Device
187 sSlaveToDevice sSlave (SlaveDeviceState sState transmit
      readerState) busState = let
188   -- Parse incoming symbol
189   (nextReaderState, busSymbol) = readSymbol readerState
        busState
190   -- Send incoming symbol to symbol level slave and get
        back next symbol
191   (nextSState, nextTransmit) =
192     case busSymbol of
193       Just sym -> sSlave sState sym
194       Nothing -> (sState, transmit)
195   -- Transmit bit
196   nextBusState =
197     if nextTransmit then busIdle
198     else busTransmit
199   in
200     (SlaveDeviceState nextSState nextTransmit
          nextReaderState, nextBusState)
201
202
203 data SSlaveImplState
204   = SSlStIdle
205   | SSlStReceive Integer Word8 | SSlStReceiveAck
        BSlaveReaction
206   | SSlStTransmit Integer Word8 | SSlStTransmitAck
207   deriving (Show)
208
209 -- Translate a byte level slave to a symbol level slave
210 bSlaveToSSlave :: BSlave -> SSlave
211 bSlaveToSSlave bSlave (SSlaveState bState sState) symbol =
212   case symbol of
213     SymStart -> handleEvent BEvStart
214     SymStop -> handleEvent BEvStop
215     SymIdle -> idleState
216     SymBit bit ->
217       case sState of
218         SSlStIdle -> idleState
```

```
219         SSlStReceive count byte ->
220           let newByte = byte * 2 + (boolToInt bit) in
221           if count == 7 then handleReceiveEvent (BEvReceive
                  newByte)
222           else (SSlaveState bState (SSlStReceive (count +
                  1) newByte), True)
223         SSlStReceiveAck bReaction ->
224           handleReaction (SSlaveState bState) bReaction
225         SSlStTransmit count byte ->
226           if nthBit byte count && not bit then
227             idleState -- arbitration lost
228           else if count == 7 then
229             (SSlaveState bState SSlStTransmitAck, True)
230           else
231             (SSlaveState bState (SSlStTransmit (count + 1)
                    byte), nthBit byte (count + 1))
232         SSlStTransmitAck ->
233           if bit then idleState
234           else handleEvent BEvAck
235   where
236     idleState = (SSlaveState bState SSlStIdle, True)
237     handleEvent bEvent =
238       let (newBState, bReaction) = bSlave bState bEvent in
239       handleReaction (SSlaveState newBState) bReaction
240     handleReaction s bReaction =
241       case bReaction of
242         BReactIdle -> (s SSlStIdle, True)
243         BReactTransmit byte -> (s (SSlStTransmit 0 byte),
                nthBit byte 0)
244         BReactReceive -> (s (SSlStReceive 0 0), True)
245     handleReceiveEvent bEvent =
246       let (newBState, bReaction) = bSlave bState bEvent in
247       if bReaction == BReactIdle then (SSlaveState
              newBState SSlStIdle, True)
248       else (SSlaveState newBState (SSlStReceiveAck
              bReaction), False)
249
250
251 -- Translate a high level slave to a byte level slave
252 hSlaveToBSlave :: HSlave -> BSlave
253 hSlaveToBSlave hSlave (BSlaveState hState first) bEvent =
254   case bEvent of
255     BEvStart -> (BSlaveState hState True, BReactReceive)
256     BEvStop -> (BSlaveState (slaveStop hSlave hState)
            False, BReactIdle)
257     BEvReceive byte ->
258       if first then
259         let
260           isRead = (byte .&. 1 == 1)
261           (newHState, ack) = slaveAddress hSlave hState
                (toInteger (byte `shift` (-1))) isRead
262         in
263           if not ack then (BSlaveState newHState False,
                BReactIdle)
```

```
264           else if isRead then
265             let (newHStateR, byteR) = slaveRead hSlave
                    newHState in
266             (BSlaveState newHStateR False, BReactTransmit
                    byteR)
267           else (BSlaveState newHState False, BReactReceive)
268       else
269         let (newHState, ack) = slaveWrite hSlave hState
                byte in
270         (BSlaveState newHState False, if ack then
                BReactReceive else BReactIdle)
271     BEvAck ->
272       let (newHState, byte) = slaveRead hSlave hState in
273       (BSlaveState newHState False, BReactTransmit byte)
274
275
276 data LMasterTransmitState
277   = TransmitIdle
278   | BitPrepare Bool
279   -- Waiting for SCL to go high
280   | BitWait Bool
281   | StartWait
282   | StopWait
283   deriving (Show)
284
285 -- Translate a symbol level master to a low level bus device
286 sMasterToDevice :: SMaster -> Device
287 sMasterToDevice sMaster (MasterDeviceState sState state
        readerState) busState = let
288   -- Parse incoming symbol
289   (nextReaderState, busSymbol) = readSymbol readerState
          busState
290   -- Send incoming symbol to symbol level master and get
          back next symbol
291   (nextSState, tmpNextState) =
292     case busSymbol of
293       Just sym ->
294         let
295           (nextSState, nextSymbol) = sMaster sState sym
296           tmpNextState = case nextSymbol of
297             SymIdle -> TransmitIdle
298             SymStart -> StartWait
299             SymStop -> StopWait
300             SymBit bit -> BitPrepare bit
301         in
302           (nextSState, tmpNextState)
303       Nothing -> (sState, state)
304   -- Transmit symbol
305   BusState scl _ = busState
306   (nextState, nextBusState) = case tmpNextState of
307     TransmitIdle -> (TransmitIdle, busIdle)
308     StartWait ->
309       if busState == busIdle then (TransmitIdle, BusState
              True False)
```

```
      else (StartWait, busIdle)
    StopWait ->
      if busState == BusState True False then
          (TransmitIdle, busIdle)
      else (StopWait, BusState True False)
    BitPrepare bit ->
      if scl then (BitPrepare bit, BusState False bit)
      else (BitWait bit, BusState True bit)
    BitWait bit ->
      if scl then (TransmitIdle, BusState False True)
      else (BitWait bit, BusState True bit)
  in
    (MasterDeviceState nextSState nextState
       nextReaderState, nextBusState)


data SMasterImplState
  = SMStIdle | SMStBusBusy | SMStStart | SMStStop
  | SMStWrite Integer Word8 | SMStWriteAck
  | SMStRead Integer Word8 | SMStReadAck BMasterAction
  deriving (Show)

-- Translate a byte level master to a symbol level master
bMasterToSMaster :: BMaster -> SMaster
bMasterToSMaster bMaster (SMasterState bState sState)
    symbol =
  case sState of
    SMStIdle ->
      if symbol == SymStart then busyState
      else handleResult BResOk
    SMStBusBusy ->
      if symbol == SymStop then handleResult BResOk
      else busyState
    SMStStart ->
      handleResult (if symbol == SymStart then BResOk else
          BResUndefinedCondition)
    SMStStop ->
      handleResult (if symbol == SymStop then BResOk else
          BResUndefinedCondition)
    SMStWrite count byte ->
      case symbol of
        SymBit bit ->
          if nthBit byte count && not bit then
            handleResult BResArbitrationLost
          else if count == 7 then
            (SMasterState bState SMStWriteAck, SymBit True)
          else
            (SMasterState bState (SMStWrite (count + 1)
                byte), SymBit (nthBit byte (count + 1)))
        _ -> handleResult BResUndefinedCondition
    SMStWriteAck ->
      case symbol of
        SymBit bit -> handleResult (if bit then BResNack
            else BResOk)
```

```
357         _ -> handleResult BResUndefinedCondition
358     SMStRead count byte ->
359       case symbol of
360         SymBit bit ->
361           let newByte = byte * 2 + (boolToInt bit) in
362           if count == 7 then handleReadResult
                (BResReadResult newByte)
363           else (SMasterState bState (SMStRead (count + 1)
                newByte), SymBit True)
364         _ -> handleResult BResUndefinedCondition
365     SMStReadAck bAction ->
366       case symbol of
367         SymBit bit ->
368           if bAction /= BActRead && not bit then
                handleResult BResArbitrationLost
369           else handleAction (SMasterState bState) bAction
370         _ -> handleResult BResUndefinedCondition
371   where
372     busyState = (SMasterState bState SMStBusBusy, SymIdle)
373     handleResult bResult =
374       let (newBState, bAction) = bMaster bState bResult in
375       if bResult == BResArbitrationLost then
376         (SMasterState newBState SMStBusBusy, SymIdle)
377       else
378         handleAction (SMasterState newBState) bAction
379     handleAction s bAction =
380       case bAction of
381         BActIdle -> (s SMStIdle, SymIdle)
382         BActStart -> (s SMStStart, SymStart)
383         BActStop -> (s SMStStop, SymStop)
384         BActWrite byte -> (s (SMStWrite 0 byte), SymBit
              (nthBit byte 0))
385         BActRead -> (s (SMStRead 0 0), SymBit True)
386     handleReadResult bResult =
387       let (newBState, bAction) = bMaster bState bResult in
388       (SMasterState newBState (SMStReadAck bAction), SymBit
            (bAction /= BActRead))
389
390
391 data BMasterTransaction = BMasterTransaction {
392   msgs :: [Message],
393   replies :: [MessageReply],
394   state :: BMasterTransactionState
395 } deriving (Show)
396 data BMasterTransactionState = TrStStart | TrStAddress |
      TrStRead Integer [Word8] | TrStWrite [Word8] Integer
      deriving (Show)
397
398 -- Translate a high level master to a byte level master
399 hMasterToBMaster :: HMaster -> BMaster
400 hMasterToBMaster hMaster (BMasterState hState bState)
      result =
401   if result == BResArbitrationLost || result ==
        BResUndefinedCondition then let
```

```
    newHState =
      if isJust bState then transactionReply hMaster hState
        (case result of
          BResArbitrationLost -> ArbitrationLost
          BResUndefinedCondition -> UndefinedCondition)
      else hState
    in
      (BMasterState newHState Nothing, BActIdle)
  else
    case bState of
      Nothing ->
        doNextTransaction hState
      Just (BMasterTransaction [] replies _) ->
          doNextTransaction (transactionReply hMaster
              hState (Success (reverse replies)))
      Just (BMasterTransaction (msg:msgs) replies
          transState) ->
        let
          buildState state = BMasterState hState (Just
              (BMasterTransaction (msg:msgs) replies state))
          doNextMessage reply =
            if isSuccessful msg reply || nonCritical msg
                then
              (BMasterState hState (Just
                  (BMasterTransaction msgs (reply:replies)
                  TrStStart)),
                (if null msgs then BActStop else BActStart))
            else
              (BMasterState hState (Just
                  (BMasterTransaction []
                  ((reverse (map messageNackReply msgs)) ++
                      (reply:replies)) TrStStart)),
                BActStop)
        in
        case transState of
          TrStStart ->
            let addr = msgAddress msg * 2 + boolToInt
                (msgIsRead msg) in
            if addr < 0 || addr > 255 then error "Address␣
                out␣of␣range"
            else (buildState TrStAddress, BActWrite
                (fromInteger addr))
          TrStAddress ->
            if result == BResNack then
              doNextMessage (messageNackReply msg)
            else
              case msgData msg of
                Read {size = s} ->
                  if s <= 0 then error "Empty␣read"
                  else (buildState (TrStRead s []),
                      BActRead)
                Write [] -> doNextMessage (MessageReply
                    True (WriteReply 0))
                Write (b:bs) -> (buildState (TrStWrite bs
```

```
                    0), BActWrite b)
          TrStRead readLength bs ->
            let
              BResReadResult b = result
              newReadLen = readLength - 1 +
                (if null bs && variable (msgData msg) then
                    toInteger b else 0)
            in
              if newReadLen <= 0 then
                doNextMessage (MessageReply True (ReadReply
                    (reverse (b:bs))))
              else
                (buildState (TrStRead newReadLen (b:bs)),
                    BActRead)
          TrStWrite _ bytesAcked | result == BResNack ->
            doNextMessage (MessageReply True (WriteReply
                bytesAcked))
          TrStWrite [] bytesAcked ->
            doNextMessage (MessageReply True (WriteReply
                (bytesAcked + 1)))
          TrStWrite (b:bs) bytesAcked ->
            (buildState (TrStWrite bs (bytesAcked + 1)),
                BActWrite b)
  where
    doNextTransaction hState =
      let (newHState, maybeTrans) = nextTransaction hMaster
          hState in
      case maybeTrans of
        Nothing -> (BMasterState newHState Nothing,
            BActIdle)
        Just [] -> error "Empty␣transaction"
        Just trans ->
          (BMasterState newHState (Just (BMasterTransaction
              trans [] TrStStart)), BActStart)

msgIsRead :: Message -> Bool
msgIsRead msg =
  case msgData msg of
    Read {} -> True
    Write {} -> False

messageNackReply :: Message -> MessageReply
messageNackReply msg =
  MessageReply False
    (if msgIsRead msg then ReadReply [] else WriteReply 0)

isSuccessful :: Message -> MessageReply -> Bool
isSuccessful msg reply =
  acked reply &&
    case msgData msg of
      Read {} -> True
      Write bytes -> msgDataReply reply == WriteReply
          (toInteger (length bytes))
```

```
-- Directly run a high level transaction on a high level slave
hSlaveRunTransaction :: HSlave -> HSlaveState -> Transaction -> (HSlaveState, TransactionReply)
hSlaveRunTransaction _ _ [] = error "Empty transaction"
hSlaveRunTransaction hSlave hState tr =
    (slaveStop hSlave newHState, Success replies)
  where
    (newHState, replies) = doTransaction hState tr
    doTransaction hState [] = (hState, [])
    doTransaction hState (m:ms) = let
        (hStateM, reply) = doMessage hState m
        (hStateMs, replies) =
          if isSuccessful m reply || nonCritical m then doTransaction hStateM ms
          else (hStateM, map messageNackReply ms)
      in
        (hStateMs, (reply:replies))
    doMessage hState msg =
      let (hStateAddr, addrAck) = slaveAddress hSlave hState (msgAddress msg) (msgIsRead msg) in
      if not addrAck then
        (hStateAddr, messageNackReply msg)
      else let
        (hStateReply, dataReply) =
          case msgData msg of
            Read { size = s, variable = v } ->
              if s <= 0 then error "Empty read"
              else
                let
                  (_, firstByte) = slaveRead hSlave hStateAddr
                  finalSize = s + (if v then toInteger firstByte else 0)
                  (hStateRd, bytes) = doRead hStateAddr finalSize
                in (hStateRd, ReadReply bytes)
            Write bytes ->
              let (hStateWr, ackCount) = doWrite hStateAddr bytes in
              (hStateWr, WriteReply ackCount)
        in
          (hStateReply, MessageReply True dataReply)
    doWrite hState [] = (hState, 0)
    doWrite hState (b:bs) =
      let (hStateB, wrAck) = slaveWrite hSlave hState b in
      if not wrAck then
        (hStateB, 0)
      else
        let (hStateBs, ackCount) = doWrite hStateB bs in
        (hStateBs, ackCount + 1)
    doRead hState count =
      if count <= 0 then (hState, [])
      else
```

```
        let
          (hStateRd, b) = slaveRead hSlave hState
          (hStateRest, bs) = doRead hStateRd (count - 1)
        in (hStateRest, (b:bs))


------------------------------------------------------------
-- Global state and step functions
------------------------------------------------------------

data GlobalState = GlobalState {
  devices :: [Device],
  deviceStates :: [DeviceState],
  busState :: BusState
}

globalStep :: GlobalState -> GlobalState
globalStep state = let
    devicesWithState = zip (devices state) (deviceStates
        state)
    deviceBusStates = map (\(d, s) -> d s (busState state))
        devicesWithState
    (nextDeviceStates, busStates) = unzip deviceBusStates
  in
    state {
      deviceStates = nextDeviceStates,
      busState = mergeBusStates busStates
    }

mergeBusStates :: [BusState] -> BusState
mergeBusStates [] = BusState True True
mergeBusStates ((BusState scl sda):t) = let
    BusState sclt sdat = mergeBusStates t
  in
    BusState (scl && sclt) (sda && sdat)

data SGlobalState = SGlobalState {
  sMasters :: [SMaster],
  sMasterStates :: [SMasterState],
  sSlaves :: [SSlave],
  sSlaveStates :: [SSlaveState],
  busSymbols :: [BusSymbol]
}

sGlobalStep :: SGlobalState -> SGlobalState
sGlobalStep state =
  case busSymbols state of
    [] -> state -- deadlock
    (busSymbol:_) -> let
      mastersWithState = zip (sMasters state)
          (sMasterStates state)
      slavesWithState = zip (sSlaves state) (sSlaveStates
          state)
      (nextMasterStates, masterBusSymbols) = unzip
```

```
582         (map (\(master, masterState) -> master masterState
                busSymbol) mastersWithState)
583       (nextSlaveStates, slaveDrives) = unzip
584         (map (\(slave, slaveState) -> slave slaveState
                busSymbol) slavesWithState)
585       in
586         state {
587           sMasterStates = nextMasterStates,
588           sSlaveStates = nextSlaveStates,
589           busSymbols = mergeBusSymbols masterBusSymbols
                slaveDrives
590         }
591
592 -- If an empty list is returned, this indicates a deadlock.
593 -- If the returned list has multiple elements, this
    indicates a race condition,
594 -- each device sees one of the symbols in the list (not
    necessarily the same).
595 -- This is called "undefined condition" in the I2C spec.
    However, that
596 -- definition in the spec is more broad, and includes
    combinations which can
597 -- be correctly resolved by arbitration.
598 mergeBusSymbols :: [BusSymbol] -> [Bool] -> [BusSymbol]
599 mergeBusSymbols masterSyms slaveDrives = let
600   someoneSends = (`elem` masterSyms)
601   slaveDrive = and slaveDrives
602   in
603     if someoneSends (SymBit False) then [SymBit False] else
604     if someoneSends (SymBit True) then (
605       if not slaveDrive then [SymBit False]
606       else if someoneSends SymStop then [SymBit False,
            SymStop]
607       else if someoneSends SymStart then [SymBit True,
            SymStart]
608       else [SymBit True]
609     ) else
610     if not slaveDrive then [] else
611     if someoneSends SymStop then [SymStop] else
612     if someoneSends SymStart then [SymStart] else
613     [SymIdle]
```