Model Railway 4.0

Nis Börge Wechselberg

Master Thesis
submitted in March 2015

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl. Inf. Christian Motika
Dipl. Inf. Ass. jur. Insa Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

To provide a real-world example of an embedded real-time system in teaching, the Real-Time Embedded Systems Group operates a model railway installation. This model railway can not only be used in advanced, practical laboratory courses, but also in public presentations of the Institute for Computer Science at Kiel University.

This Thesis documents the design and development of a new controller infrastructure for the model railway. The history of the model railway and the previous implementations of controller hardware are discussed and the problems with the previous system are shown. The options for new controller hardware implementations are shown and rated and the selected hardware is discussed in detail. Using this hardware, the different software layers for the controller infrastructure are then designed and implemented. The new options for configuring and testing the hardware of the model railway are presented. Finally, an application programming interface (API) for usage in future controllers is presented.

Acknowledgements

I would like to thank Prof. Dr. Reinhard von Hanxleden for the unique opportunity to write this thesis at the group for Embedded and Real-Time Systems. Additionally, I would like to thank my advisors Insa Marie-Ann Fuhrmann and Christian Motika for their great support during the work on this thesis.

I thank Stephan Höhrmann for all the material on the history of the model railway and the great work he did as my predecessor. I have to thank Christoph Daniel Schulze for the support with photos of the model railway and my hardware components.

Finally, I want to thank my parents for all their support during my life and especially during my studies.

Contents

List of Figures

Abbreviations

8N1	eight data bits, no parity bit, and one stop bit
API	application programming interface
CAN	control area network
COTS	commercial off-the-shelf
CPC	Central Processing Unit
DIP	dual in-line package
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	general-purpose input/output
IC	Inner Circle
ICSP	In-Circuit Serial Programming
IDE	integrated development environment
IPC	inter-process communication
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JSON	JavaScript Object Notation
KH	Kicking Horse Pass
KIELER	Kiel Integrated Environment for Layout Eclipse Rich-Client
LED	light-emitting diode
MoC	Model of Computation
NAT	network address translation
NFS	Network File System
OC	Outer Circle
PLCC	plastic leaded chip carrier
PSK	*Platinen-Steckkontakt*

PWM	pulse-width modulation
RISC	reduced instruction set computing
SCCharts	Sequentially Constructive Charts
SoC	system on a chip
SOIC	Small Outline Integrated Circuit
SUD	system under development
TCP	Transmission Control Protocol
TQFP	Thin Quad Flat Package
TTP	Time-Triggered Protocol
UART	universal asynchronous receiver/transmitter
USB	Universal Serial Bus
VPN	virtual private network

Chapter 1

Introduction

"Reactive real-time embedded systems are pervasive in the electronics system industry. Applications include vehicle control, consumer electronics, communication systems, remote sensing, and household appliances." Edwards et. al. used these sentences in 1997 to introduce their paper about designing embedded systems [ELL+97]. Fourteen years later Lee and Seshia prefaced their book on embedded systems with the claim "The vast majority of computers in use, however, are much less visible. They run the engine, brakes, seatbelts, airbag, and audio system in your car. [...] These less visible computers are called embedded systems, and the software they run is called embedded software."[LS11]

While teaching computer science, it is common for students to have a personal computer at home to experiment with and develop software on. Far less common is owning a programmable embedded system to experiment with. While platforms like the LEGO Mindstorms robotics platforms[1] can be used on a small scale, the challenges of a larger system are rarely discussed and tackled. The Chair of Real-Time and Embedded Systems of the *Christian-Albrechts-Universität zu Kiel* operates a custom-built model railway installation to give students the option to gather these experiences.

A model railway can provide valuable insight and different research opportunities in the world of embedded and real-time systems as well as distributed systems. These works are distributed across different levels of abstraction and take individual views on the model railway each. Given the size and the complexity of the installation, it is advised to use several smaller computers to control parts of the whole system under development (SUD). These smaller computers are no general purpose systems but microcontrollers and dedicated electronic components. Developing software for these controllers, like in this thesis, probably forms the lowest abstraction level of work on the model railway.

On a slightly higher level, communication between parts of the SUD has to be managed. This is often done by specialized bus systems like control area network (CAN) or Time-Triggered Protocol (TTP) buses which could be used and tested on the model railway.

The most common task for students is developing a controller for the model railway installation [HFP+06]. This controller reads the generated values from the various sensors, like Reed contacts or voltage measurements, and reacts by setting the available actors, for example track voltage, signals or switch points. Obviously, the model railway is a real-time system which imposes tight deadlines on the computation of valid responses. In addition, when controlling multiple trains, problems regarding concurrency, mutual exclusion and fairness arise. These

[1] `http://mindstorms.lego.com`

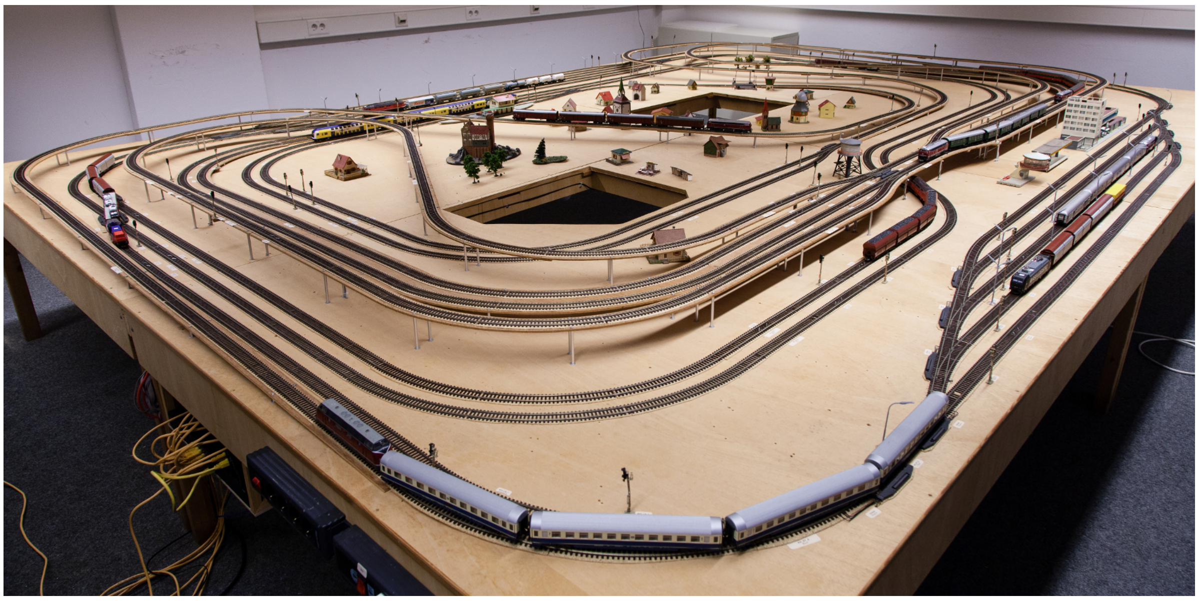


Figure 1.1. The model railway installation[2]

problems are often discussed in class using small examples or theoretical constructs. In contrast to this, the model railway provides a large-scale example to study and solve these problems.

Due to the open interfaces of the model railway, the options to implement a controller are widely spread. An API to use the railway is provided in C, but similar APIs could be implemented in almost every programming language. Using this kind of API, the students can either implement a controller in the target language directly or use higher level modeling tools. In advanced laboratory courses these controllers have been implemented with tools like Matlab/Simulink[3], SCADE[4] or KIELER SCCharts[5].

1.1 Goals

During the last 8 years the model railway installation has been used in multiple advanced laboratory projects and several public relations events. The needed hardware was developed by Stephan Höhrmann in 2005 and consists of a combination of PC/104 computing nodes and custom-built power electronics boards. The primary goal of this thesis is the replacement of this hardware with new parts. This new hardware should primarily consist of commercial off-the-shelf (COTS) hardware to provide the option of easily replacing defective parts in the future.

[2]For attributions for all figures please refer to page 73.

[3]`http://de.mathworks.com/products/matlab/`

[4]`http://www.esterel-technologies.com/products/scade-suite/`

[5]`https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/SCCharts`

The new hardware should, as far as possible, be backwards compatible to the previous generation of hardware. This would enable the controllers developed in previous courses to be used as test-cases for the new hardware.

The hardware should be open to future extensions like adding more track segments or new sensors. To further ease the extension, a simple configuration and testing interface should be provided.

The current focus in research at the Real-Time Systems group suggests more work on high-level controller development and model-driven development instead of low-level communication research. For this reason the option to extend the communication layer, e.g. with a CAN or TTP bus, is not in the focus of this thesis, even though the previous hardware provided this option.

1.2 Contributions

In this thesis the design and development of the new controlling hardware of the model railway is shown. The new hardware is constructed using Arduino Mega boards and Raspberry Pis Model B+ in combination with some peripheral hardware modules. The hardware is capable of controlling all parts of the model railway and can control multiple trains on the installation simultaneously. The Arduinos can be programmed, configured and tested through a web-based interface, provided by the Raspberry Pis.

The new hardware is mostly backwards compatible, allowing most legacy controllers to be run on the new hardware.

1.3 Outline

Chapter 2 first presents the history and the previous generations of the model railway. It then describes the railway hardware of the installation and the new testing circle. Options for low-level controlling hardware of the installation are described and discussed in Chapter 3. In Chapter 4 the communication between the controlling hardware is detailed and the valid messages are described. Chapter 5 shows the high-level communication, shows the new configuration and testing interface and shows the usage of the high-level API. The results, possible extensions and future work are shown and discussed in Chapter 6.

Chapter 2

The Model Railway Installation

Section 2.1 describes the inspiration of the model railway and the history which led to the current state of the model railway installation as well as the different approaches of the previous generations. Subsequently, Section 2.2 describes the railway hardware and the individual requirements of the railway parts. In Section 2.3 the testing circle, which was created during development, is detailed.

2.1 History and Previous Iterations

The plans for creating a model railway installation started back in 1995 when Prof. Kluge, the head of the group for computer organization and architecture at Kiel University, was researching scheduling problems. He chose a real-world example for his reasearch, the Kicking Horse Pass [Klu98]. This pass was the first transcontinental railway track through the Canadian Rocky Mountains and was first opened in 1885 [Pol95].

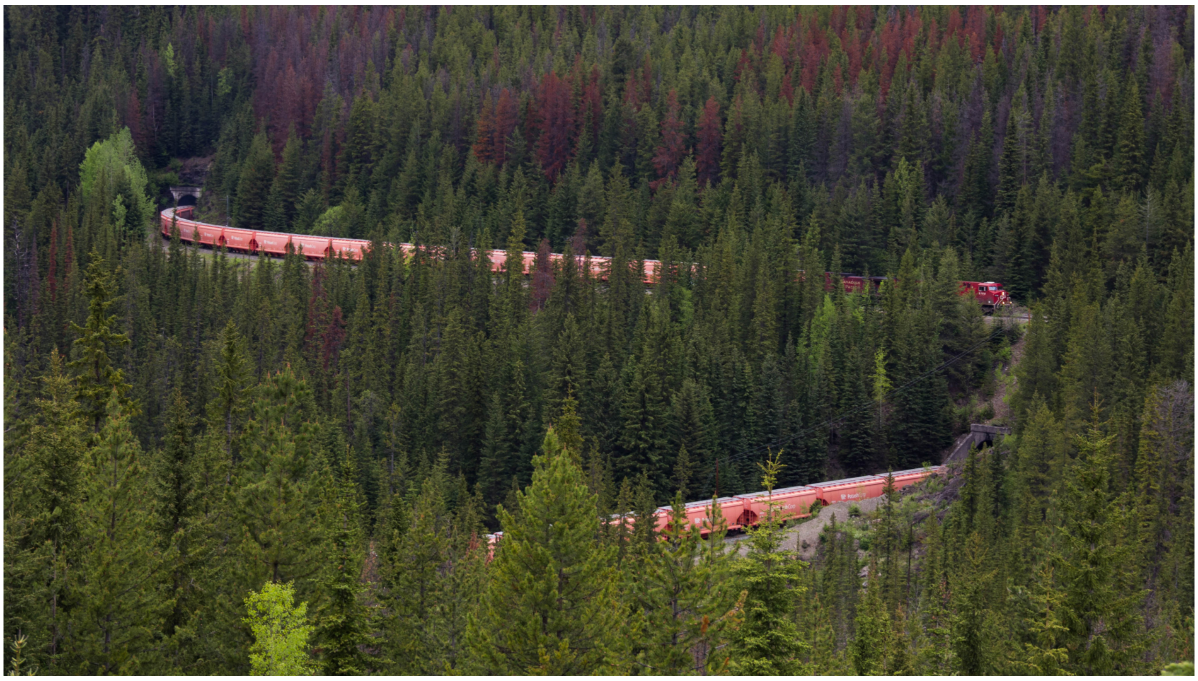


Figure 2.1. Freight train simultaneously entering and exiting Lower Spiral Tunnel

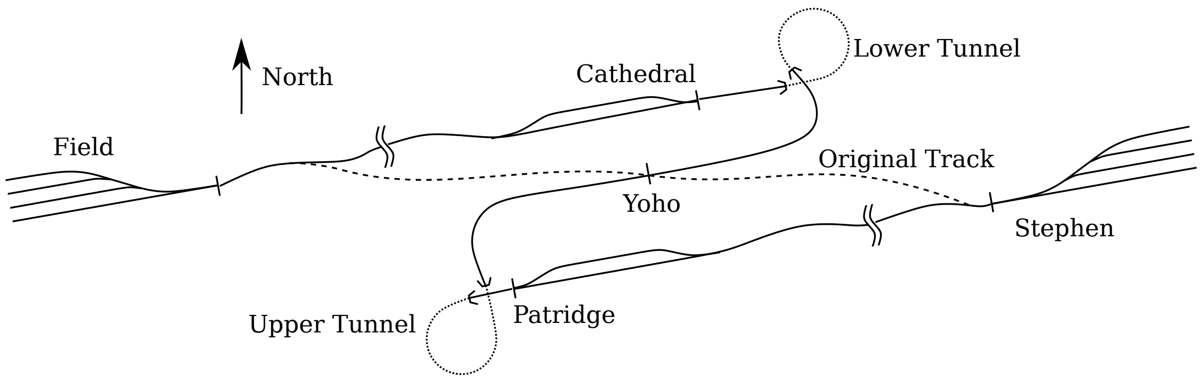


Figure 2.2. Schematic of the Kicking Horse Pass

Today, the region is part of the Yoho National Park. The first railway route through this pass was an almost direct route across the pass. This version was the steepest railroad track in North American railroad history with a ruling gradient of 4.5 percent. Due to this steep route two main problems were determined. Firstly the trains needed strong additional engines to climb to the summit. The second problem were frequent accidents with runaway trains on the descent. To reduce the ruling grade, two spiral tunnels were constructed, which reduced the incline to 2.2 percent at the cost of additional 12 km track length. The complete pass segment was built as a single track requiring coordination of trains in both directions. In 1995 a train needed about 1 hour to cross the complete pass and reach the next station with a turnout track. In these days the pass had a load of up to 30 trains a day, making some form of coordination unavoidable.

2.1.1 Model Railway 1.0

The first generation of the model railway installation was constructed between 1995 and 1997 by the research group for computer organization and architecture [The99]. The track layout was inspired by the spiral tunnels of the original Kicking Horse Pass and featured two spirals and an elevated pass, about 30cm above table level, between these two, as can be seen in Figure 2.3. A simplified schematic of the first track layout can be found in Appendix C.

The controlling infrastructure was designed and built by Jürgen Noss and consisted of a SUN Workstation which was equipped with a bidirectional interface at its parallel port. This interface was connected to multiple central circuit boards. Each of these boards was used to control specific hardware parts, one type of board controlling block segments and one other type controlling switch points. All of these boards used a specialized, self-developed bus system with 34 data lanes.

The primary research and teaching focus was developing a reliable controller with criteria like fairness and deadlock prevention to study the real world problems of the pass [Pol95]. The main modeling technique used in these courses were Petri net models. A complete Petri net to control the system was published by Hielscher et al. in 1998 [HUR+98]. The Petri net was

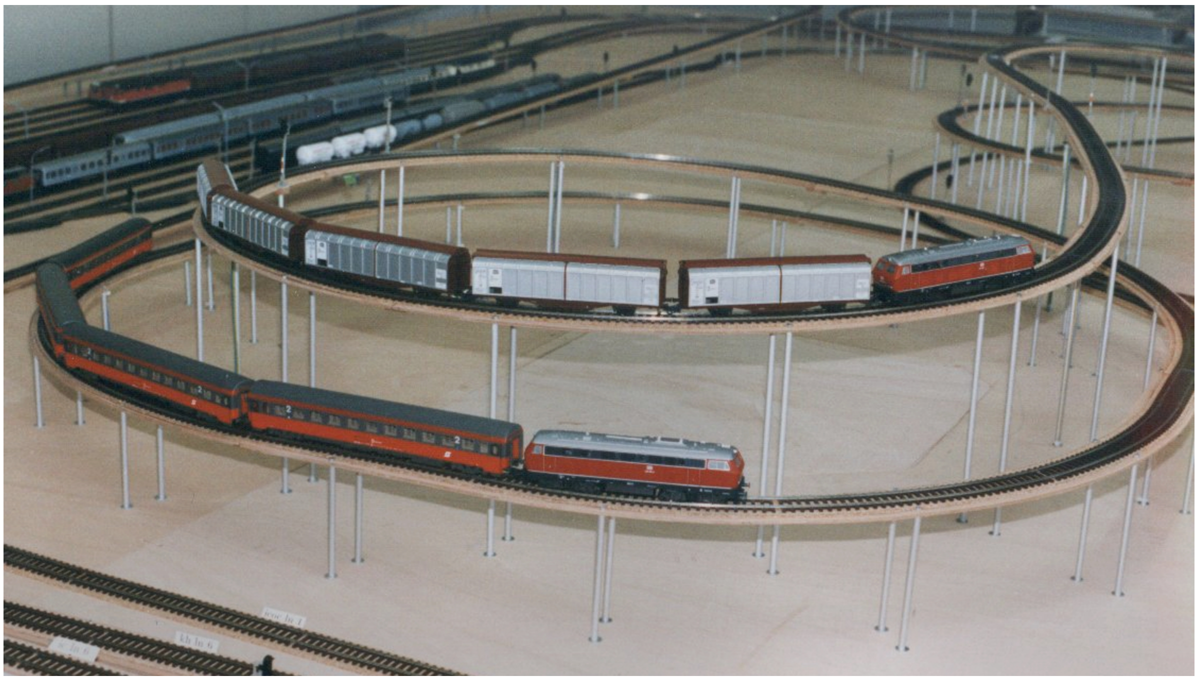


Figure 2.3. Trains climbing a spiral on the first installation

modeled using the Design/CPN tool of the Aarhus University[1], but the tool was determined hardly usable for teaching because of a very steep learning curve and frustrating long compile and simulation times.

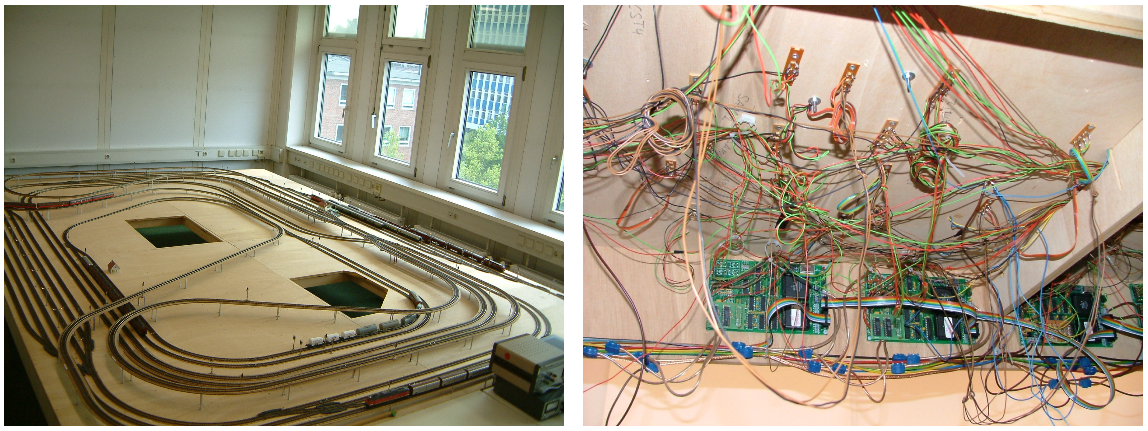
Ironically, as exposed by Höhrmann, the main problems with the first railway installation were very similar to the problems at the real Kicking Horse Pass [Höh06]. Trains frequently had problems climbing these steep tracks or fell from the elevated parts of the pass. In addition to these problems, electrical interferences on the central circuit boards caused malfunctions.

Furthermore, Reed contact events were unreliable. On the one hand, the system generated events without trains passing over a contact, on the other hand, contact events were not reliably created for each passing train. The installation was in use for some time until the research group relocated to new rooms at the main university campus. It then was not immediately reconstructed, laying dormant until the second generation was constructed.

2.1.2 Model Railway 2.0

In 1999 two students, Jochen Koberstein and Oliver Schmitz, started to reconstruct of the model railway in their spare time, leading up to a joint diploma thesis in 2001 [Kob01; The03]. They did a thorough reconstruction of the track layout incorporating the lessons learned from the flaws of the first generation. This mainly involved flattening the spiral tracks, while keeping the main spirit of the layout intact. The modifications resulted in slightly changed interconnections between the three main circles and a turnout track in the middle of the Kicking Horse Pass.

[1] `http://www.daimi.au.dk/designCPN/`



(a) Redesigned railway track layout **(b)** Interbus boards and wiring

Figure 2.4. Second generation of the model railway

For this reason, the model railway installation is no accurate model of the real pass anymore, but the name Kicking Horse Pass was kept nonetheless. The new track layout and the general design of the model railway can be seen in Figure 2.4a. A schematic view of the track layout can be seen in Figure C.2. The new turnout option allowed for a more "lively" train behavior, due to a faster scheduling of trains traveling in opposing directions on the Kicking Horse Pass.

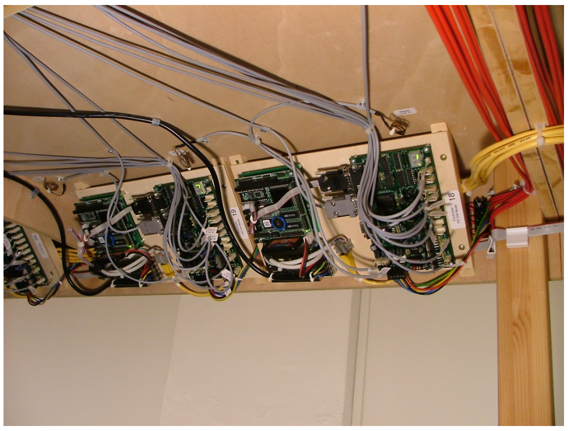
In addition to these changes, the controlling electronics was improved. The former central design was replaced with 24 controlling boards, connected by a common INTERBUS[2]. While this reduced the problems with electrical interferences, the new distributed system was not able to eliminate these problems completely. Koberstein and Schmitz also were able to locate problems with the Reed contacts which were either not detecting passing trains or creating sensor events without a train passing over them. They managed to isolate and fix problems with the placement of magnets at the trains, leading to a much improved rate of sensor detections. The additional events could be attributed to switch point operations. Strong interferences, generated at operating the switch point, and long parallel wires of contacts and switches lead to a strong inductive coupling between these parts.

The focus of the model railway installation was still placed on high-level controller design. Koberstein and Schmitz developed a Petri net for the new track layout and implemented a controlling library for controlling the model railway [Kob01]. In addition they developed the *Railway Control Center*, which allowed the user to define schedules for all trains. These schedules were then executed on the model railway while effectively preventing deadlocks.

The main problem of the second installation was the wiring of the whole system. The cables were mechanically unstable and, as can be seen in Figure 2.4b, hard to follow and maintain if some component broke.

Nevertheless multiple advanced laboratory courses were able to use the installation and develop their own controllers. After Prof. Kluge retired in 2002, the model railway was passed on to Prof. von Hanxleden and the chair for Embedded and Real-Time Systems. In 2004 the

[2]`http://www.interbus.de/`


(a) Power electronics and PC/104 nodes

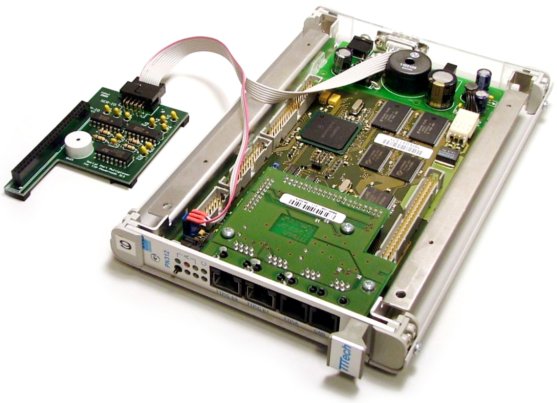

(b) TTP Powernode

Figure 2.5. Controlling hardware of the third generation

model railway had to be moved from the former rooms at Herman-Rodewald-Straße to their current room at Wilhelm-Seelig-Platz. Due to a different focus in research, a redesign of the controlling infrastructure was already planned. The plans were sped up when the INTERBUS boards were damaged while moving the installation. This led to the third generation of the model railway.

2.1.3 Model Railway 3.0

In the winter term 2004/2005 the student Stephan Höhrmann started the reconstruction of the installation in the new rooms as port of his diploma thesis [Höh06; The06]. The top-side of the tables was mostly left untouched, but the complete wiring was redesigned. The only addition to the track layout was the integration of a railroad crossing in the Kicking Horse Pass. The third generation was able to learn from the problems of the previous generations and to prevent electromagnetic interferences to a wide degree.

The new controlling infrastructure consisted of 24 custom-built power electronics boards as shown in Figure 2.5a. Each of these boards was able to control two track segments, four signals, four Reed contacts and four switch points. The new wiring consisted of dedicated cables for each component and a direct connection to the next free port on the power electronics board. Each of these cables has been individually documented and labeled to support possible debugging efforts.

Plans for future research and teaching shifted from high-level controllers to communication in the system. For this reason the controller should provide extension options to connect various different communication infrastructures like a CAN bus. The power electronics provided up to four serial connections to connect various hardware nodes. The first port was connected to a PC/104 computer each. These nodes consisted of a 386series processor and 32MB of flash memory and were able to communicate either via CAN or via Ethernet. The computers provided an interface between the serial communication of the power electronics and the controlling communication. As a parallel system 8 Time-Triggered Protocol Powernodes, as shown in

Figure 2.5b, were attached to 3 power electronics each. These nodes were able to communicate via CAN or via TTP bus.

In the following years the third generation of the model railway has been used for at least three diploma/master level laboratory courses as well as numerous public-relations presentations of the Department for Computer Science [HFP+06; Men13; Mot14b]. At these events high-school students were able to write their own simple controller and control a part of the installation with it. Roughly estimatated, in the 8 years of usage about 1000 students had the option to control the third generation with their own software.

2.1.4 Model Railway 4.0

In recent years the focus in research shifted back from inter-system communication to high-level controller modeling. Multiple recent laboratory courses managed developing controllers with up to 11 trains controlled concurrently on the model railway. In 2007 the course used SCADE[3] to model a controller for the TTP Powernodes[4]. Afterwards, the controller developed in this course has been used at presentations of the model railway as an example of an advanced controller. In the summer term of 2012, a similar task was solved by the students, this time using the PC/104 nodes for their controller[5].

The most recent course was held in the summer term of 2014[6]. The students used KIELER SCCharts, a new programming language and modeling tool, developed at the research group for Real-Time and Embedded Systems [HDM+14]. Sequentially Constructive Charts (SCCharts) are based on classical synchronous programming languages, like Esterel [Ber99] or Lustre [CPH+87], but extend the Model of Computation (MoC), e.g., by allowing sequential updates of variables during one tick, as long as race conditions can safely be circumvented. The students were able to create a controller using the PC/104 nodes, and managed to control up to 11 trains with dynamic schedules which could be updated through a smartphone application.

After several years of usage the PC/104 nodes started developing faults. Some nodes didn't start reliably anymore and needed several reboots until the node became available. Additionally the surrounding network infrastructure of network gateway systems and Network File Systems (NFSs) were breaking down and needed to be hotfixed for most presentations of the model railway. The power electronics didn't produce any faults during the whole time of operation, but in the event of a failure the only available replacement parts would be 5 bare, unpopulated circuit boards.

As mentioned, the 24 power electronics boards provided two ports for track segments each. These ports were fully utilized by the 48 track segments, requiring the addition of new power electronics boards and new PC/104 nodes to possibly extend the model railway installation.

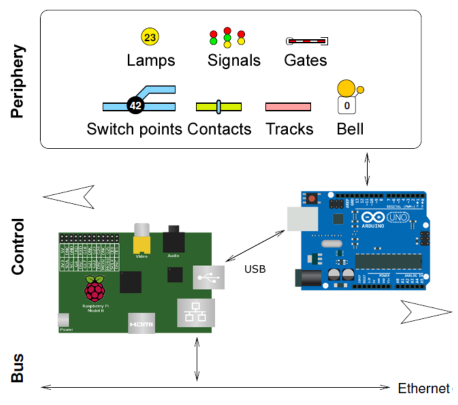
In April 2014 Christian Motika proposed a new design for the model railway controlling infrastructure [Mot14a]. The goal was the development of new hard- and software, mainly consisting of COTS components while reusing the existing model railway periphery and sticking to the established API of the third generation of the model railway.

[3] `http://www.esterel-technologies.com/products/scade-suite/`

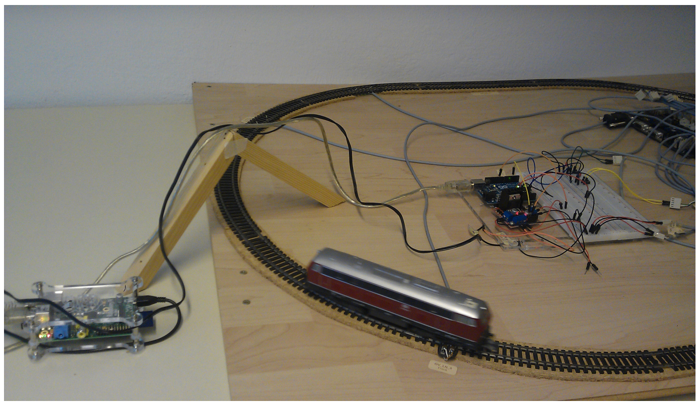
[4] `http://www.informatik.uni-kiel.de/rtsys/teaching/ss07/p-railway/`

[5] `http://www.informatik.uni-kiel.de/rtsys/teaching/ss12/p-railway/`

[6] `http://rtsys.informatik.uni-kiel.de/confluence/display/SS14Railway`



(a) Proposed system design



(b) Prototype implementation on the first testing circle

Figure 2.6. Proposed prototype for the fourth generation

The initially proposed design consisted of an Arduino board and an H bridge driver managing the low-level tasks of controlling railway periphery. This Arduino was controlled by a Raspberry Pi Model B. The proposed design could be used to control the testing circle which Stephan Höhrmann built during his diploma thesis. To accomodate future reconfigurations and extensions, a web-based configuration management was proposed. Some parts of the model railway periphery were not integrated in the initial test because the testing circle did not cover all elements of the model railway. This left open questions about the integration of switch points or lights into the proposed design. Additionally the scalability of the system needed to be validated by implementing the design on the full size model railway.

The evaluation of possible hardware platforms began in October 2014. The decision for using Raspberry Pis and Arduinos was not fixed and possible alternatives were investigated. To fully test the final design a bigger testing circle (detailed in Section 2.3) was constructed and the hardware design evaluated. Multiple adapter circuit boards were constructed to connect the new controller hardware to the existing model railway periphery. The implementation on the model railway installation was started in January 2015, due to delayed delivery of needed parts. In February the new hardware was installed and tested. Initial tests were able to control multiple simultaneous trains.

2.2 Railway Hardware

As ususal for reactive embedded systems, the model railway contains various sensors and actors. The next sections 2.2.1 to 2.2.4 describe the characteristics of the actors, like Tracks, Points, Signals and Lights. Subsequently, Section 2.2.5 describes the primary sensors, the Reed contacts. One type of hybrid system can be found on the installation, the railroad crossing, which consists of a combination of multiple actors and sensors simultaneously and is detailed in Section 2.2.6.

2.2.1 Tracks

The model railway installation uses standard railway material in size H0 from the company *Roco*[7]. In total, 127 meters of track are used in the complete installation. These tracks are split into 48 track blocks with lengths between 51 and 459 cm. The tracks form three primary circles, the Inner Circle (IC) which only allows counter-clockwise traffic, the Outer Circle (OC) which restricts trains to clockwise traveling and the Kicking Horse Pass (KH) which allows traffic in both directions. To allow trains to change from one circle to the other, four interconnections, called IO, OI and KIO[8], are provided. The simplified view on the track layout is shown in Figure C.2 in Appendix C.

The tracks are powered by a standard switching direct current power supply, providing 12V of voltage and up to 8A of current. By convention a train drives *forward*, in reference to the desired travel direction, when the right track has the higher electric potential. This means that turning the engine by 180° does not change the travel direction. The engines of the trains do not provide any means of digital control to individually manage the speed of a train. Instead each track block can individually be controlled by pulse-width modulation (PWM). By default, e.g., after resetting the hardware, all power to the tracks should be turned off.

2.2.2 Switching Points

Throughout the model railway 28 switch points are used. The majority of these are standard two-way switching points which enable a train to change from its main track to a side track. By convention these switching points are named *left* and *right* switches, depending on the direction a train can branch of. If the train can either travel straight or branch to the left side, as shown in Figure 2.7, the point is called a *left switching point*.

The switch points are operated by an external actuator, which can also be seen in Figure 2.7. This actuator primarily consists of two magnetic coils and a moving central core. This central core is connected to a small rocker which controls the needed actions. When current flows through one of the coils the core is pulled into the coil and the rocker is moved. The first resulting effect is the movement of a small piece of spring wire which moves the switch blades to the desired position. A secondary effect is the operation of a limit stop. This limit stop interrupts the electrical circuit of the recently activated coil and stops excessive current from flowing through the coils, preventing overheating and destruction of the actuator.

A third effect of the actuator action can be the polarization of the heart of the switching point, if desired. The heart is the central piece of track, right at the intersection of the branching and the straight track. This piece can be left unpolarized, meaning that it is not connected to any part of the power supply. This might cause train engines with very few contact wheels to loose their connection to the power supply and stop. This could especially happen with slow moving trains as a faster moving train would slide across the gap and reconnect at the other side. If the heart is polarized, it has to be connected to one of the clamps of the power supply, depending on the setting of the switching point. In Figure 2.7 the heart has to be connected to the right rail if the train is branching and to the left rail otherwise. These actions are performed

[7] `http://www.roco.cc`

[8] As shown in Figure C.2 the KIO interconnection is split in two parts, counting as two different interconnections.

by the actuator if the three small black wires at the right side of Figure 2.7 are connected. A minor drawback of polarized hearts is that trains can not be *burst open* the points anymore. When bursting a switch point open, the train travels against the branching direction while the switch point is not set to the correct setting. This can only be done because the switch blades are not fixed to the actuator but flexible. When the heart of the switch point is polarized, it causes a short circuit when the train engine passes over it. Throughout the model railway installation all switch points are using polarized hearts.

Two variants of switch points can be found on the installation. The interconnections between the Kicking Horse Pass and the other circles use curved versions of the normal two-way switch points. Apart from their different appearance these versions operate the same as the normal points. A second variant are the two single-slip diamond crossings, shown in Figure 2.8, which are used at the intersection of the Outer Circle and the KIO interconnections. These diamond crossings resemble two interweaved two-way points with two hearts and need two actuators to operate. The two actuators result in four possible combinations of setting. Out of these settings only three settings are viable because the fourth setting would always cause a short circuit.

Due to this restriction the two diamond crossings are not controlled symmetrically. The possible combinations of the actuators and the resulting paths are shown in Table 2.1. The diamond crossings are placed at the two intersection of IC, OC and KIO. They need to connect the KIO interconnections to both the Inner Circle as well as the Outer Circle but should not allow trains to change between IC and OC. If trains would change between IC and OC using this interconnection they would violate the restrictions on travel directions imposed on both circles.

In the first and second generation of the model railway the switch points were a major source of electromagnetic interferences. This is caused by the limit stop of the actuators which creates a strong spark when operated. This spark caused an impulse on the cables which could lead to an inductive coupling with adjacent cables. Stephan Höhrmann created simple filters, consisting

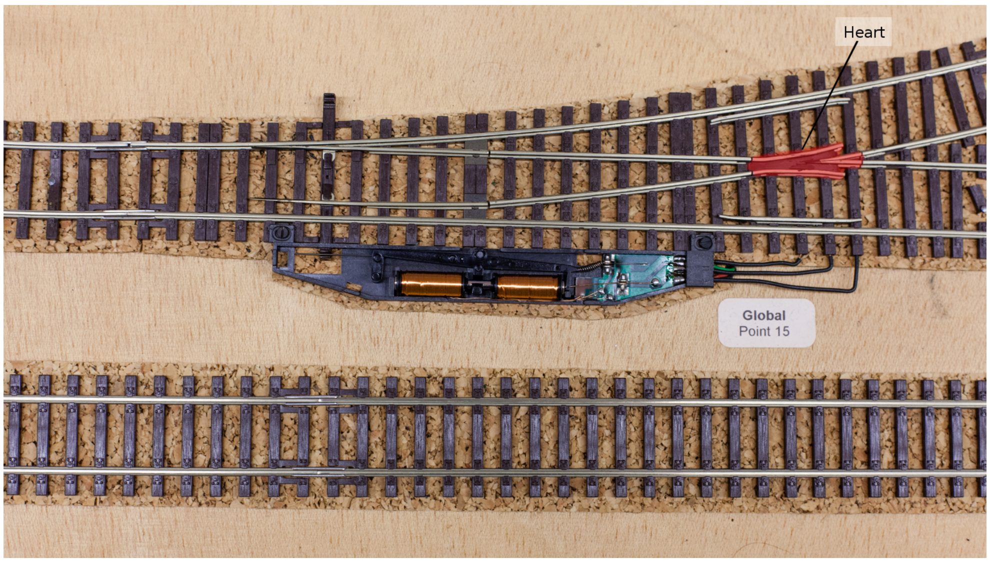


Figure 2.7. A left switching point with opened driver

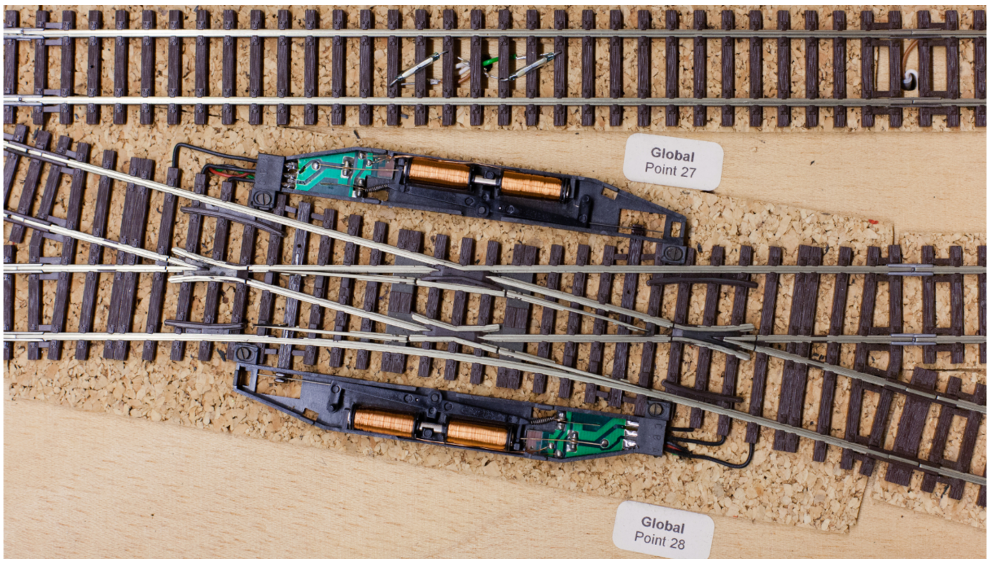


Figure 2.8. Single-slip diamond crossing with opened actuators

Table 2.1. Combination of actuators for single-slip diamond crossings (based on [Höh06])

Point 16	Point 17	Resulting connection
STRAIGHT	STRAIGHT	OC_ST_4 ↔ OC_LN_0
STRAIGHT	BRANCH	OC_ST_4 ↔ KIO_LN_0
BRANCH	STRAIGHT	Not allowed
BRANCH	BRANCH	IC_ST_0 ↔ KIO_LN_0
Point 27	Point 28	Resulting connection
STRAIGHT	STRAIGHT	IC_ST_4 ↔ KIO_LN_1
STRAIGHT	BRANCH	Not allowed
BRANCH	STRAIGHT	OC_ST_0 ↔ KIO_LN_1
BRANCH	BRANCH	0C_ST_0 ↔ OC_LN_5

of diodes and capacitors. Two of these filters are shown in Figure 2.9. These filters are used on each actuator of the installation. Additionally the cables for points are routed independently from other cables to further prevent inductive coupling and a galvanically isolated power supply is used to power the switch points.

2.2.3 Signals

As a visual addition to the model railway installation 56 signals are placed adjacent to the tracks. The signals consist of small light-emitting diodes (LEDs) and are either block signals, which show either red or green, or main signals which have an added yellow LED, as shown in Figure 2.10.

The block signals are used at the end of each block that only allows trains to continue straight. In this case a red signal means that an approaching train is not allowed to enter the next block while a green signal permits the continuation. The main signals are used at places

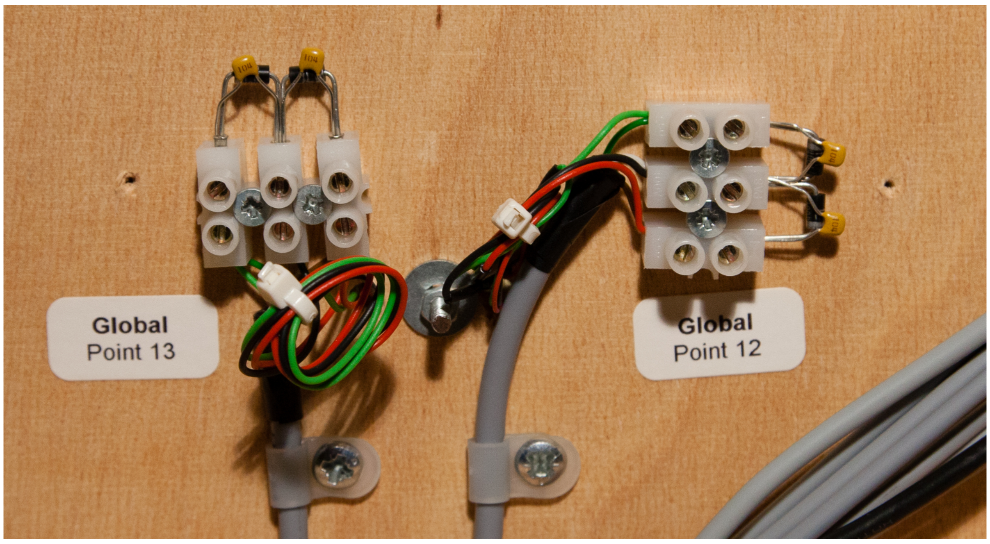


Figure 2.9. Filter for switching points

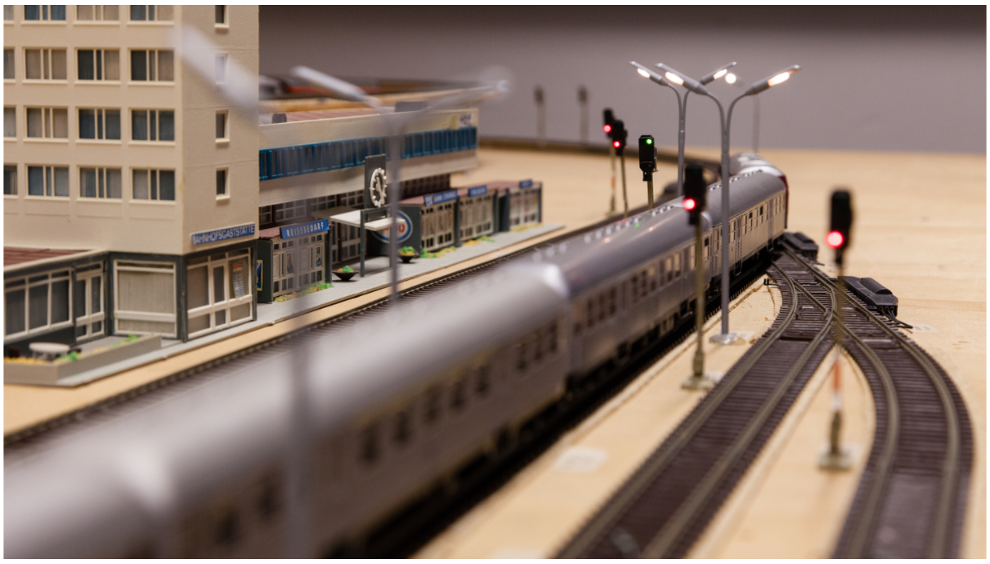


Figure 2.10. Main signals at exit of Kicking Horse Pass Station

which provide a switching option in the next block. These signals show a combination of green and yellow if the train wants to pass the point in branching configuration. These points should only be passed slowly. It should be noted that these signals don't interfere with the controlling in any way. The signals can be completely ignored and only be used as a decorative addition.

The LEDs of the signals are connected to a common anode and individual cathodes. The power electronics of the third generation powered the signals with a voltage of 12V and series resistors of 1 kΩ, resulting in a current of between 10 and 20 milliamps. The new electronics powers the signals with a voltage of 5V and series resistors of 220 Ω in all cathodes.

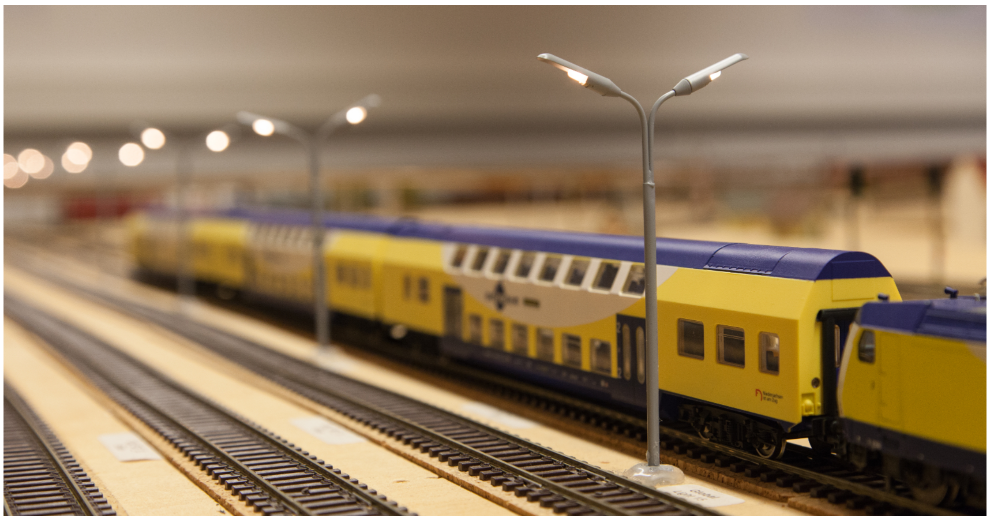


Figure 2.11. Lamp posts at Inner Circle Station

2.2.4 Lights

Primarily placed at the station areas are 24 lights. These lights consist of an arced metal post and one or two small light bulbs, as shown in Figure 2.11. The metal posts double-functions as the ground connection of the light bulbs.

The bulb is powered by a voltage of 12V and takes a current between 20 and 35 milliamps. The function of the lights is mainly aesthetic but they can also be used to show some internal state of the controller. For example the controller of the last laboratory course used the lamps to visualize the waiting state of each station track.

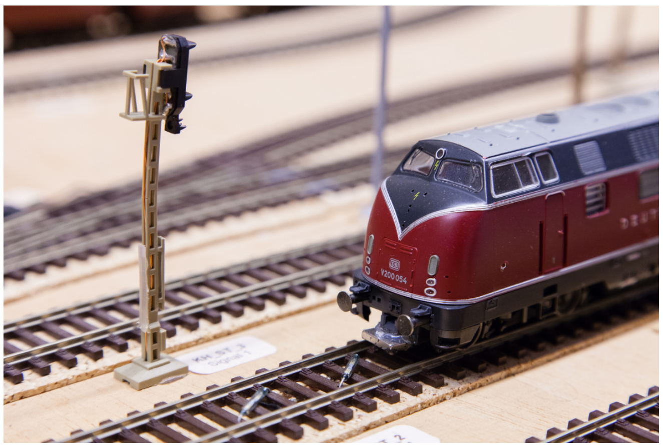
2.2.5 Reed Contacts

The only way of feedback to the controller are Reed sensors which are integrated in the tracks. Each track block which is able to contain a complete train is equipped with a pair of Reed sensors at each end. Figure 2.12a shows the two sensors of a pair, mounted with a spacing of 30 millimeters. These pairs are called a *Reed contact*. In total 160 Reed sensors are used in the installation, forming 80 contacts.

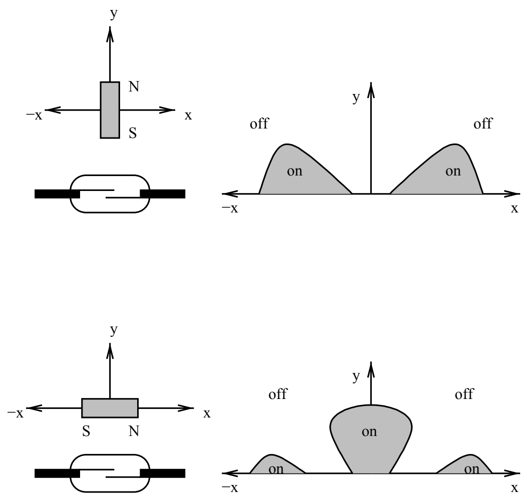
Each train carries a magnet at the front of the engine and at the end of the last rail car. When passing over a contact these magnets close the electric circuit and signal an event to the controller. Each of the sensors can individually be read by the controller hardware. Based on the order of events the hardware can then determine the direction of the passing train.

The sensors consist each of a small glass tube with two metal plates inside which normally are not connected. When a magnet is near, both metal plates are pulled against the tube, closing the electric circuit. For optimal events the tubes have to be properly aligned when mounted so that the plates are pulled straight up.

Koberstein and Schmitz determined some problems with the quality of contact events [Kob01]. As shown in Figure 2.12b the generated pattern of a Reed sensor is heavily influenced by the



(a) Reed sensors at the end of a station track



(b) Influence of permanent magnet alignment on event quality

Figure 2.12. Reed contacts at the model railway

alignment of the permanent magnet due to the orientation of the magnetic field around the magnet. In addition weak magnets tend to produce only unreliable events.

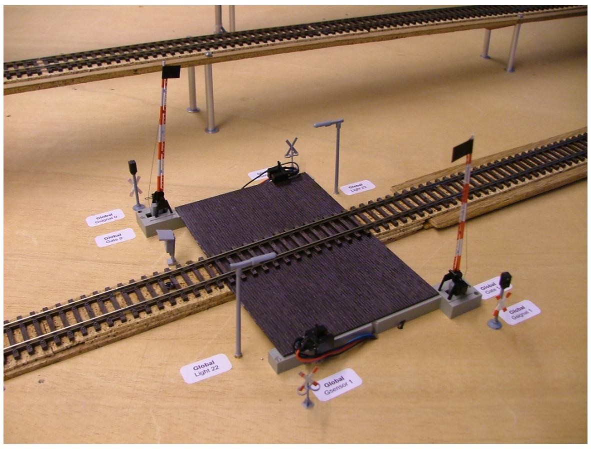
The sensors and the attached cables are susceptible to electromagnetic interferences. Additionally the generated sensor events rely on a mechanical process which might create an unstable connection. This makes careful filtering and debouncing of generated events in the controlling hardware unavoidable.

2.2.6 Railroad Crossing

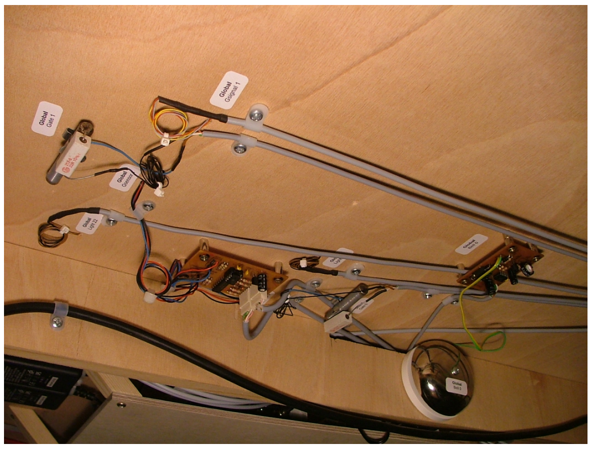
The railroad crossing is the most recent addition to the railway hardware. It forms a system of several sensors and actors on its own. The most obvious feature, easily visible in Figure 2.13a, are two booms which can be raised and lowered individually. Each of the booms is moved by a piece of memory alloy wire. When the wire is heated, e.g. by sending current through it, the wire contracts and pulls down the boom. When the current is turned off and the wire cools down again, the wire expands and the boom is raised by a small spring.

To detect the current position of the booms, each boom is equipped with a small piece of black adhesive tape. This tape is lowered into a photo sensor at the boom receptacle. These photo sensors are attached to a small circuit board beneath the installation. This circuit simulates the behavior of a Reed contact and generates a *forward* event when the photo sensor is triggered by the lowered boom. The circuit then waits for the boom to raise again and generates a *reverse* event. This circuit enables the controlling hardware to handle the photo sensors like the Reed sensors. The differentiation between Reed sensors and the railroad crossing can then be performed on a higher level of the controller.

Attached next to the circuits for the photo sensors is a door bell. This bell can be sounded to realistically signal the closing of the booms when a train approaches. The bell is controlled



(a) Railroad crossing with booms, signals, lamps and photo sensors



(b) Circuits for boom control and bell below the installation

Figure 2.13. Railroad crossing

by a driver board which generates the impulses to repeatedly sound the bell with an adjustable frequency. It can be controlled by the controlling hardware like a light post or a switching point. The hardware related to the railroad crossing is shown in Figure 2.13b.

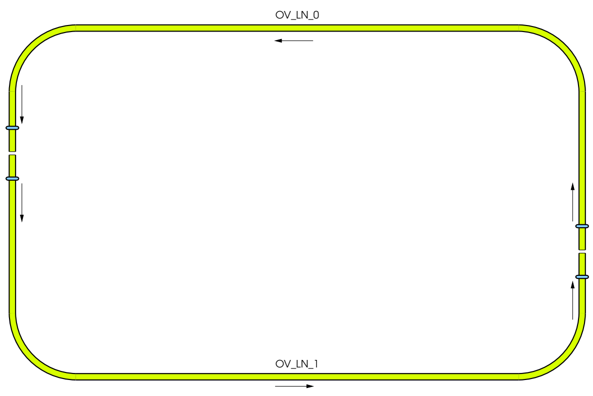
For a realistic appearance two St Andrew's Crosses are placed at the crossing. These crosses are equipped with small LEDs, colored yellow and red, and can be controlled like normal signals. Additionally two light posts are placed in the vicinity of the railroad crossing.

2.3 Testing Circle

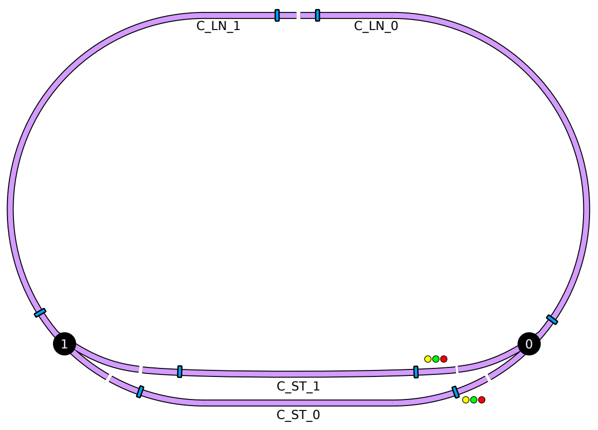
To evaluate the developed hardware, a new testing circle was created. Stephan Höhrmann already created a smaller testing circle during his diploma thesis but his circle was lacking some features present at the railway installation. Namely the first testing circle consisted only of two track segments with contacts at each end, as shown in Figure 2.14a. This meant that neither switch points nor signals could be tested with this testing circle.

The new testing circle, schematically shown in Figure 2.14b, consists of four track segments. Two of these segments form the majority of the circle while the other two form a small station area. At the end of the two station tracks block signals are placed. Two curved two-way switch points connect the station area to the circle. This new circle forms a valid testbed, providing all primary parts of the bigger installation. The switch points are filtered like the switch points at the installation.

The tracks form an oval of about 4.2 meters length. The whole testing circle is mounted on two separable wooden base plates, measuring 80cm x 120cm each. The electrical connections are provided by a central pin header which can easily be detached. This allows for easier transportation and provides an option to use the testing circle as a smaller, mobile demonstrator. Figure 2.15 shows the complete setup with an early prototype version of the controlling hardware.



(a) Third generation testcircle



(b) New testcircle with switch points and signals

Figure 2.14. Comparison of testcircles

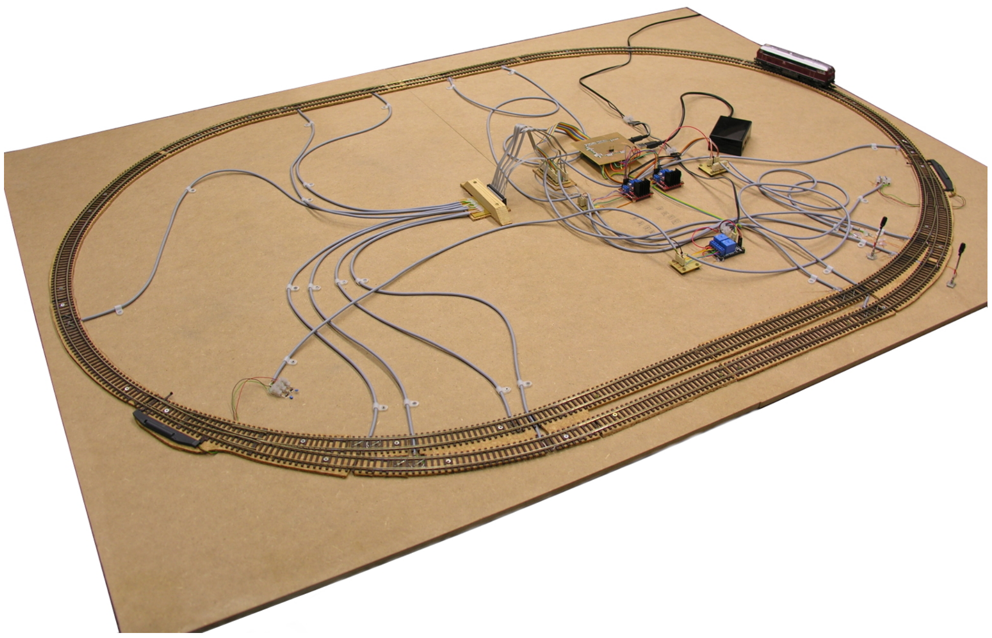


Figure 2.15. Model Railway testing circle with prototype hardware

Chapter 3

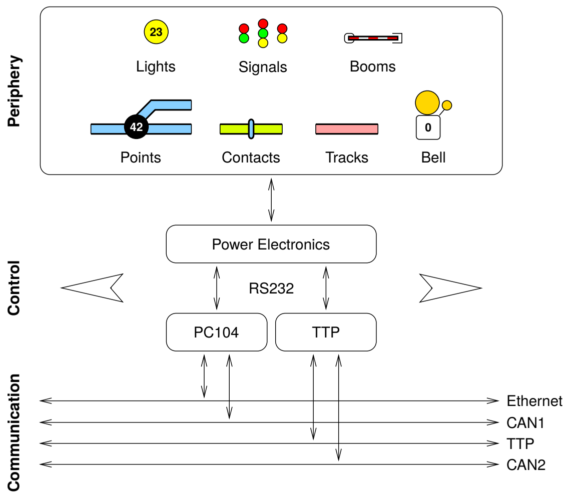
Controlling Hardware and Infrastructure

The general approach of the controlling hardware is a layered design of hardware components and communication paths. This approach is detailed in Section 3.1. The following Section 3.2 describes the various options for usage in these layers. Subsequently, in Section 3.3 the used hardware is selected. In addition to the main hardware, some smaller components are used throughout the installation, which are presented in Section 3.4. The mounting of the hardware and the network infrastructure are described in Section 3.5.

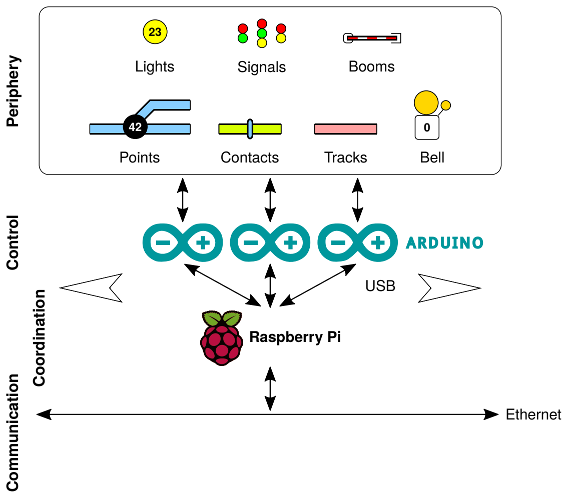
3.1 Approach

The third generation of the model railway introduced a layer based approach of controller hardware.

Figure 3.1a shows the approach of Stephan Höhrmann in the third generation of the model railway. The bottom layer shows various communication techniques used by the control infrastructure. Above the communication layer, different computing nodes are arranged. These connect to one or multiple communication paths and utilize the Power Electronics which is placed above the computing nodes and connected via RS232 connections. The power electronics then controls the periphery of the model railway through dedicated data lines for each component.



(a) Third generation hardware components



(b) Fourth generation hardware components

Figure 3.1. Comparison of layered controller hardware approach

The new approach, shown in Figure 3.1b, keeps the general layered structure. Yet, the initial implementation does not provide as much variety in the different layers. The periphery of the railway system is connected to Arduino microcontroller platforms using the same dedicated data lines as the previous system. The Arduinos provides the low-level interface to the model railway periphery. Multiple Arduino controllers are then connected to one Raspberry Pi using a USB connection and a universal asynchronous receiver/transmitter (UART) interface on this connection. The Raspberry Pis form the coordination layer, which manages the interaction between high-level controller a low-level Arduino interface. The Raspberry Pis are then connected to each other using normal fast Ethernet connections.

3.2 Technology Overview

As described in the previous section, two layers of controlling hardware are developed. In Section 3.2.2 possible alternatives for the microcontroller platform are described. The options for small embeddable computing nodes, which can be used in the coordination layer, are presented and evaluated in Section 5.4.2.

3.2.1 Coordination Systems

The coordination systems are used to manage the communication between the high-level controller, which is usually executed on a separate machine, and the microcontrollers at the model railway. Additionally the option to implement and execute a distributed high-level controller on these coordination nodes should be evaluated.

The computers should be small, energy efficient and provide the option to connect to multiple microcontrollers at the same time. This should be done via an USB connection per microcontroller board. The computers should be usable with a standard operating system. New developed software should be able to be compiled on the systems directly and not require a cross-compiler chain.

In the light of these restrictions several single-board computers are discussed in the next parts.

Beaglebone Black

The BeagleBone Black is a part of the BeagleBoard series, designed by Texas Instruments. The primary target audience was the maker community, resulting in the option to connect arbitrary hardware. The board is designed around a Sitara ARM Cortex-A8 processor, which is running at a clock speed of 1GHz. It is equipped with 512MB of DDR3 memory and 4GB of eMMC onboard storage. In terms of connectivity the board is equipped with one standard USB host port, Fast Ethernet and a Micro-HDMI connector. Additionally the BeagleBone Black is equipped with two embedded microcontrollers and two 46-pin connectors. These connectors can be used to attach different low-level peripheral, like UART, PWM or CAN-buses to the board. The complete system measures 86.40 mm x 53.3 mm and is available for about 50 Euros.

Raspberry Pi

The Raspberry Pi, or RasPi for short, was designed mainly for educational purposes. It should function as a platform to teach computer science at school and provide each student with the option to use a dedicated machine for this. For this reason the board was designed to be small and cheap while still providing enough computational power to be usable in teaching.

The Raspberry Pi is based on a Broadcom BCM2835 system on a chip (SoC) which bundles an ARMv6 processor with a decent graphics unit and low-level peripherals for USB, audio and mass storage. The processor is a 32-bit reduced instruction set computing (RISC) system and runs at 700MHz clock speed. Three versions of the Raspberry Pi are currently sold. The smaller model *A+* does not provide an Ethernet port and is equipped with 256MB of SDRAM. Additionally the board has a headphone jack and one USB port. It comes in a smaller form factor than the other boards, measuring 65 mm in length and 56 mm in width and costs about 20 Euros at retailers. The model *B+* is equipped with 512MB SDRAM, a Fast Ethernet port and four USB ports instead of one. It costs about 30 Euros. Both models provide a HDMI connector and are run from a microSD card as mass storage. Another shared feature is the general-purpose input/output (GPIO) connector which provides 40 pins for own extensions.

Recently, an improved version of the model B+ has been released, the Raspberry Pi 2 Model B. The new model is equipped with a more powerful CPU, using a 900MHz quad-core ARM Cortex-A7 CPU, and 1GB of SDRAM. The other features are similar to the model B+. Due to the improved hardware, and a high demand among the maker community, the price is currently at almost 40 Euros.

Despite the original intent, the Raspberry Pi primary use-case seems not to be the educational sector but the maker community. The low pricepoint, the extendability due to GPIO and the small formfactor lead to a common use of Raspberry Pi as "universal glue" between Ethernet communication and low-level periphery. As of time of writing, the maker blog *Hackaday*[1] lists 403 featured Raspberry Pi projects.

Raspberry Pi variants/Banana Pi/Cubieboard

Several other boards have been released after the success of the Raspberry Pi. These boards are usually trying to provide better hardware for certain circumstances. Most of the boards try to implement the same form factor as the Raspberrry Pi, or at least a very close variant. The most common feature is a more powerful CPU than the first Raspberry Pi versions. Since the release of the Raspberry Pi 2, the differences between the CPCs are much smaller than before.

The Banana Pi is produced by a development team in china. Though no official connection between the developers and the Raspberry Pi Foundation exis, the Banana Pi is clearly inspired by the Raspberry Pi. Compared to the RasPi B+, the Banana Pi is equipped with a more powerful CPU, an ARM Cortex-A7 Dual-core SoC with 1 GHz system clock speed. It has 1GB of SDRAM, a SATA connector and uses Gigabit Ethernet instead of Fast Ethernet. This board is available online for roughly 50 Euros.

Another small microcomputer is the Cubieboard2. This board is based on a similar CPU as the Banana Pi but provides a more powerful graphics unit. Additionally the Cubieboard is

[1] `http://hackaday.com` - Last visited: 2014-03-16 18:36

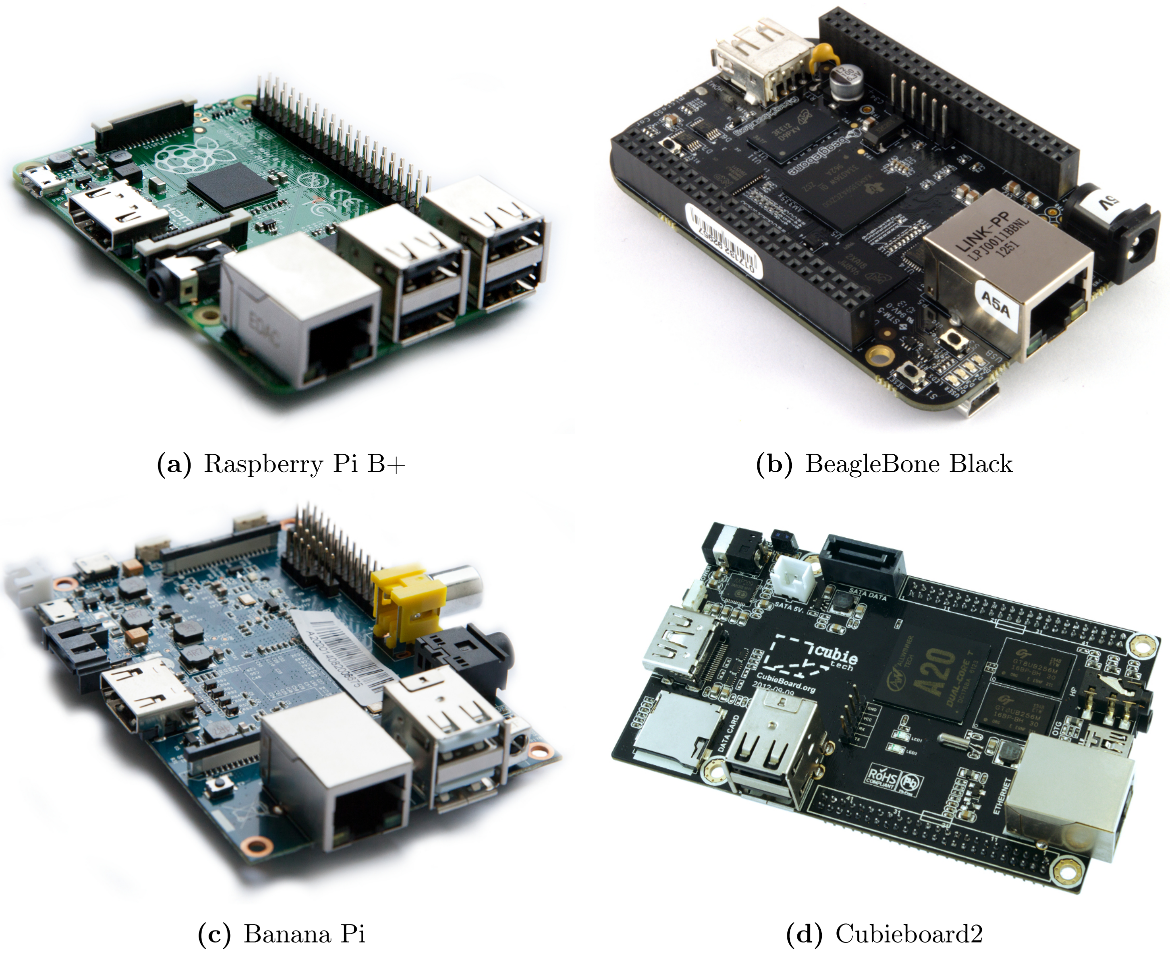
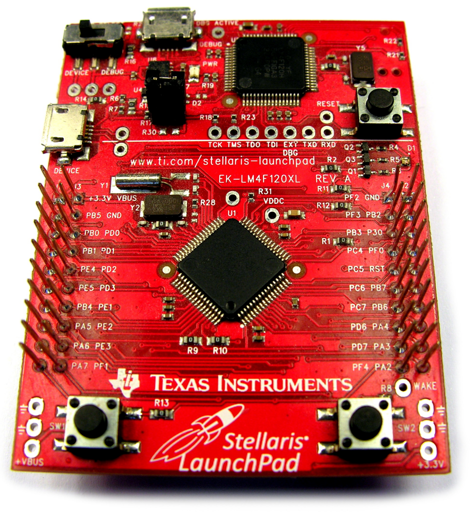

(a) Raspberry Pi B+ **(b)** BeagleBone Black **(c)** Banana Pi **(d)** Cubieboard2

Figure 3.2. Single-board computers

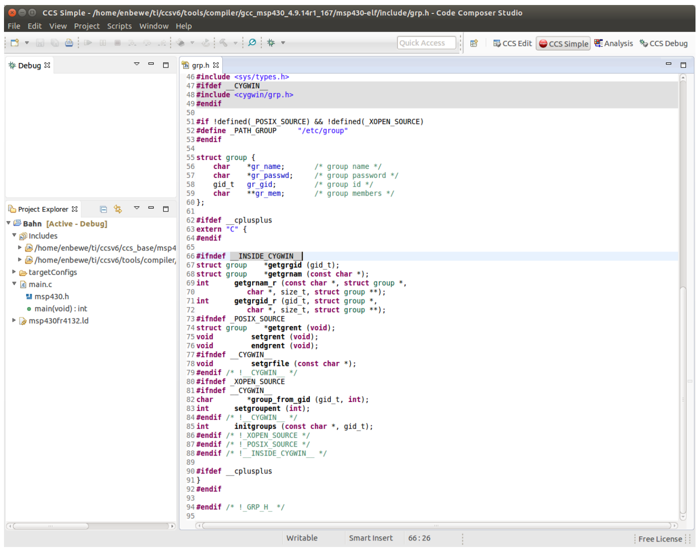
equipped with 3.4GB of onboard flash memory to be used for the operating system. The other capabilities are very similar to the Banana Pi or the Raspberry Pi.

3.2.2 Microcontroller Platforms

Several different microcontroller platforms are available on the market. Most of these platforms consist of a combination of hardware components and an IDE specially prepared for the hardware platform. To use the system at the model railway, the requirements on the IDE are fairly limited. The software should be usable on different operating systems and should provide a fairly stable development platform. Common tasks should be provided by a prepared or community-generated library. The requirements on the hardware are more detailed. The microcontroller has to provide several PWM generators, multiple analog inputs for Reed contacts and the ability to power LEDs directly through the chip. In addition, the microcontroller should provide many GPIO pins, to connect a microcontroller to many railway components. The platform should be cheap and widely available and replacement parts should be available for a reasonable period of time. This



(a) TI LaunchPad with ARM Cortex-M4



```
#include <sys/types.h>
#ifdef __CYGWIN__
#include <cygwin/grp.h>
#endif

#if !defined(_POSIX_SOURCE) && !defined(_XOPEN_SOURCE)
#define _PATH_GROUP     "/etc/group"
#endif

struct group {
    char    *gr_name;       /* group name */
    char    *gr_passwd;     /* group password */
    gid_t   gr_gid;         /* group id */
    char    **gr_mem;       /* group members */
};

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __INSIDE_CYGWIN__
struct group    *getgrgid (gid_t);
struct group    *getgrnam (const char *);
int      getgrnam_r (const char *, struct group *,
            char *, size_t, struct group **);
int      getgrgid_r (gid_t, struct group *,
            char *, size_t, struct group **);
#ifndef _POSIX_SOURCE
struct group    *getgrent (void);
void         setgrent (void);
void         endgrent (void);
#ifndef __CYGWIN__
void         setgrfile (const char *);
#endif /* !__CYGWIN__ */
#ifndef _XOPEN_SOURCE
#ifndef __CYGWIN__
char        *group_from_gid (gid_t, int);
int      setgroupent (int);
#endif /* !__CYGWIN__ */
int      initgroups (const char *, gid_t);
#endif /* !_XOPEN_SOURCE */
#endif /* !_POSIX_SOURCE */
#endif /* !__INSIDE_CYGWIN__ */

#ifdef __cplusplus
}
#endif

#endif /* !_GRP_H_ */

```

(b) Code Composer Studio

Figure 3.3. TI LaunchPad board and IDE

list tries to show some of the most common systems. Allan published an article with a wider comparison of available platforms including platforms for special purposes like wearables [All13].

TI LaunchPad

TI LaunchPad[2] is a collection of four different series of development boards from Texas Instruments. The boards run on a 3.3V power supply but differ widely in their size and capabilities [Tex14].

The smallest series are the MSP430 boards, which are mainly targeted at ultra low-power embedded systems. The microcontroller on these boards is a 16-bit RISC chip, running at 16MHz. The different boards of the series provide a common interface of 20 pins and are able to use up to 5 timers for PWM generation. Chrrently, these boards are sold for about 10 to 15 Euros for the smallest version.

The boards of the C2000 series are primarily targeted at real-time motor control applications. They use a micorcontroller from the F2802x Piccolo family, which are 32-bit chips, clocked at 60MHz. The boards feature 8 PWM channels and a 40-pin *BoosterPack* connector out of which 22 can be used as GPIO. These boards cost about 20 Euros but are rarely found at retailers.

The third series of boards are the Tiva-C Series, which are equipped with ARM Cortex-M4 microcontrollers. These chips run at 80MHz and provide features like integrated Wi-Fi network and embedded crypto engines. The boards provide the same 40-pin connector as the C2000 boards but can provide up to 24 PWM channels or 40 analog channels. The prices for the boards range between 20 and 60 Euros.

[2]`http://www.ti.com/Launchpad`

The last series consists of the Hercules boards. These boards are aimed at safety-critical automotive systems. The central microcontroller of these boards is a 32-bit ARM Cortex-R4 with 80MHz or 100 Mhz. The boards provide embedded CAN-interfaces and multiple analog channels or PWM-timers. The microcontroller is connected to the same 40-pin header as the previous series. The boards are priced at about 20 Euro.

All boards are extendable with BoosterPacks. These standardized circuit boards connect to the pin headers of the microcontroller boards and provide features like capacitive touch interfaces, battery packs, external sensors, or stepper motor control interfaces.

To program the TI LaunchPad two different IDEs are available. The *Energia* IDE was developed as a community project to provide the same user experience as the Arduino IDE while developing code for TI LaunchPads. In fact it heavily relies on the same APIs as Arduino and can generate programs for LaunchPads as well as Arduino. Most recent microcontrollers sold by Texas Instruments are compatible with Energia, but especially the Hercules series is currently not supported. Energia is available for all major operating systems, Microsoft Windows, Mac OS X and Linux. If the IDE doesn't suit the user, Energia can also function as a toolchain in the background and be embedded into Eclipse[3], Visual Studio[4] or Xcode[5]. The official IDE is the Code Composer Studio, which is an Eclipse-based software. It provides a feature-rich environment including live debugging and supports the complete portfolio of Texas Instruments microcontrollers. The Code Composer Studio is available for Microsoft Windows or as a Linux version.

Atmel AVR

The Atmel AVR series is a family of microcontrollers developed by Atmel since 1997. It consists of lots of different models and variants, grouped into several smaller families. The most common families, ATtiny and ATmega, are 8-bit microcontrollers. In addition a family of 32-bit controllers, called AVR32, is available.

The chips are not commonly sold as ready-to-use development boards but instead the bare chips are used. These chips can then be inserted into custom designed circuit boards. Some development boards are available from Atmel but are mostly only used to burn the programs to the microcontrollers. The boards are unusually expensive, costing nearly 100 Euros.

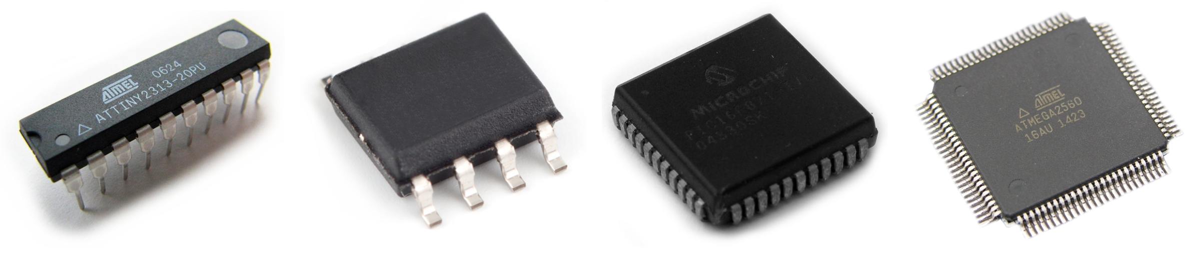
The ATtiny family form the lower end of the AVR microcontrollers with 1 to 8 kB of program memory and up to 256 bytes of RAM. The chips are available as either dual in-line package (DIP) or Small Outline Integrated Circuit (SOIC) with 8, 14 or 20 pins. The chips vary greatly in their capabilities, providing analog/digital converters, PWM-capabilities, Brown Out Detectors and Watchdogs or integrated UART. A common feature of the ATtiny family is the missing hardware multiplication unit, requiring every multiplication to be implemented in software. The chips are sold at prices between 1 and 2 Euros.

The bigger brothers of the ATtiny family are called ATmega. These chips come in a even wider variety than the ATtiny. The smallest ATmega, the ATmega8, comes with 8kB of Flash program memory and 1kB of RAM. It is capable of providing 3 PWM-channels, 8 analog/digital

[3]https://eclipse.org/

[4]https://www.visualstudio.com/

[5]https://developer.apple.com/xcode/



(a) ATtiny2313 in DIP **(b)** Unmarked SOIC chip **(c)** PIC16F in PLCC **(d)** ATmega in TQFP

Figure 3.4. Microcontroller in different packages

channels, Brown Out, a real time counter and different serial interfaces including UART. The biggest chip of the series, the ATmega2560, combines 256kB of flash memory, 8kB of RAM, 86 GPIO lines, real time counter, 16 PWM-channels, 4 UART-interfaces and a JTAG interface for on-chip debugging, as well as other features. Due to the higher pin-count of the ATmega chips, these chips are not sold as SOIC packages. The smaller chips are still available as DIP with up to 40 pins. The bigger chips are available as plastic leaded chip carrier (PLCC) or Thin Quad Flat Package (TQFP) with up to 100 pins. The chips are sold at prices between 2 and 10 Euros.

The chips could be used to implement a new custom power electronics board, like the one designed by Stephan Höhrmann for the third generation of the model railway. Stephan Höhrmann did not use Atmel AVR microcontrollers for his power electronics, but used Microchip PIC controllers instead. One of these microcontrollers is shown in the chip comparison in Figure 3.4c. Due to the focus on COTS components this is no preferred option. The Atmel AVR chips are used in various other products, the most prominent being the Arduino platform.

As an official IDE, the Atmel Studio[6] is available for free download. It is based on Visual Studio Shell by Microsoft and is only compatible with the Microsoft Windows operating system. As an alternative several free IDEs are available for various platforms. A free compiler, called AVR-GCC, is available and a toolchain for compiling, linking and flashing of the chip is built around this compiler. IDEs based on Eclipse, Netbeans[7] or Code::Blocks[8] are then implemented using the AVR-GCC toolchain.

Arduino

The Arduino project[9] started producing the first development boards in 2005. Since the beginning it produces a combination of microcontroller development boards and an easy-to-use IDE.

Most of the boards are based on 8-bit Atmel AVR chips from the ATmega series, most commonly the ATmega328. Around these chips additional parts to provide a quick start into development are provided. The Arduino Policy describes the essential parts of an Arduino board:

[6]https://www.atmel.com/tools/atmelstudio.aspx

[7]https://netbeans.org/

[8]https://www.codeblocks.org/

[9]http://www.arduino.cc

> “the original design includes all the electronic parts necessary to power and communicate with the microcontroller: regulator, clock crystal, USB-to-serial interface, and SPI programming interface for replacing the bootloader.” [Ard15]

An outstanding feature of the Arduino platform is the fact that all board designs are published under the *Creative Commons Attribution-ShareAlike 2.5 (CC-BY-SA) license*. This means that all designs and schematics are available for free download and everybody is allowed to produce (almost)[10] exact copies, so-called *clones*, or modified versions of the original boards, as long as these boards are also published under a CC-BY-SA license and proper attribution to the Arduino project is given.

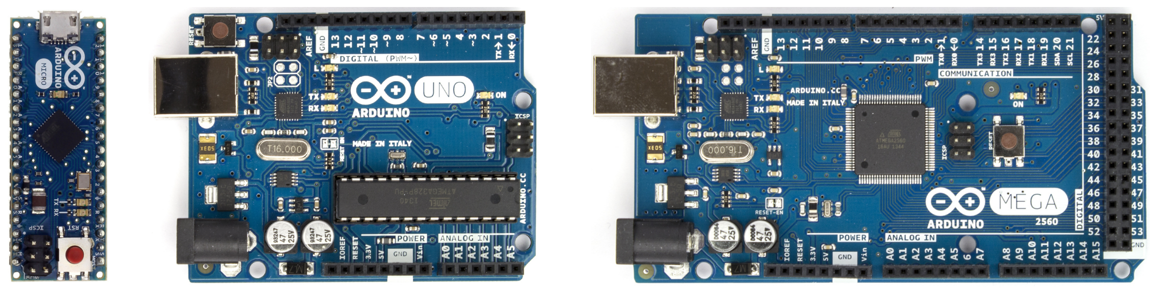
Several boards share a common form factor of roughly 53mm x 69mm board size. The first board, and the reference design for the Arduino platform, is the *Arduino Uno*. The Arduino Uno uses either a DIP or a TQFP-version of the ATmega328. The board provides an integrated In-Circuit Serial Programming (ICSP) and a UART-interface, which is provided over USB-connection. In addition to these components the board provides 14 digital GPIO pins (of which 6 can be used as PWM outputs) and 6 analog inputs. The pin mapping of the Arduino Uno is used for several other boards. These boards can be extendend by so-called shields, extension boards that can be stacked on top of the Arduino board and provide new capabilities. Some examples for official shields are GSM, Ethernet or WiFi boards or a USB Host shield.

Some boards are specially designed to provide a smaller form factor. These boards, called mini, micro and nano, are only 18 mm wide and between 30mm and 45mm long. This form factor enables the boards to be used directly on a breadboard during prototyping phase. The boards provide up to 20 GPIO pins with 6 or 7 PWM signals and between 6 and 12 analog inputs. The Arduino Mini does not provide a USB-connection and needs a separate adapter to be programmed. The two other boards provide a mini or micro USB-connector.

The biggest board is the Arduino Mega 2560. The board has 54 digital GPIO pins, of which 15 can be used as PWM outputs, 16 analog inputs and 4 UARTs.

The official Arduino boards are widely available for between 20 and 40 Euros, depending on the board size. Due to the open license several manufacturers provide exact, cheaper, clones of the official boards. These clones are available for about 10 to 20 Euros.

[10]The name Arduino is registered as a trademark, so third parties have to vary the name. Mostly names like *SomethingDuino* or *ArduSomething* are used.



(a) Micro **(b)** Uno **(c)** Mega 2560

Figure 3.5. Arduino Boards

3.3 Technology Conclusion

Matching the presented microcontroller boards and the coordination nodes against the individual requirements of the model railway led to the selection of Arduino Mega 2560 and Raspberry Pi B+ as the hardware of choice.

The Arduino Mega 2560 was primarily chosen due to easy availability, good library and documentation support. Additionally, replacement parts should be easily available in the future, either as official boards or third-party products. Due to the high pin-count of the Arduino Mega 2560, one board can replace two power electronics boards of the third generation. The new nodes are capable of controlling up to 6 track segments, 8 Reed contacts, 6 signals and 6 switch points. The Arduino boards provide a simple line of female circuit board connectors. To establish the compatibility to the current PSK print connectors a circuit board is designed which functions as an adapter board. For cost reduction the initial setup of the fourth generation uses Arduino clones from SainSmart[11]. These clones are identical to the official boards in almost every regard, apart from trademark protected logos on the circuit boards. If a board breaks in the future it can easily be replaced by official Arduino boards or other third-party boards.

The Raspberry Pi has been chosen because the hardware is available from multiple retailers and personally owned boards were already present at the staff of the Embedded Systems Group. These personal units could be used for first test installations and did not cause any problems in these tests. The BeagleBone Black is harder to obtain and would require the usage of a dedicated USB hub, due to the single USB port on the board. The two microcontrollers and lots of GPIO would serve as a nice addition to the design but currently no use case requires these options. The other variants provide no benefit over the Raspberry Pi. The higher system performance should be no factor, regarding the fact that the third generation of the model railway was using 386-series boards. Additionally most of the variants are providing only two USB ports at most, requiring a USB hub like the BeagleBone. The Raspberry Pi B+ provides 4 USB ports and can therefore be connected to multiple Arduino boards. The Raspberry Pi 2 has been released after the hardware for the model railway was chosen and ordered. Due to the same form-factor and similar specifications the new Raspberry Pi 2 could be used if one of the Model B+ nodes needs to be replaced.

It needs to be said that none of the presented hardware provides a guarantee for future availability. The chosen products are the most popular representatives of each group. This should lead to a good supply of replacement boards, or at least compatible boards, in the future. Also, given the low pricepoint of the individual components, replacement parts can easily be stocked before actually needed. Ultimately, if the Arduino boards are not available anymore, the hardware design is available and could be reproduced by a circuit board manufacturer.

Some functions need to be implemented by additional hardware components. Many railway parts need to be powered by a voltage of 12V and also take a substantial amount of current. The microcontroller of the Arduino board (as well as all other options) cannot provide the required amount of current or voltage. Instead some dedicated components are used for these railway parts.

[11] `https://www.sainsmart.com/`

3.4 Additional Controlling Periphery

To interface the model railway with the selected hardware, a custom circuit board needed to be designed and built. This circuit board is described in Section 3.4.1. Some components were not custom built, but purchased as complete modules. These modules are described in Section 3.4.2 and Section 3.4.3.

3.4.1 Arduino Railway Interface Boards

The Arduino boards provide a set of female pin headers, to connect to almost arbitrary hardware. This can be useful while prototyping a circuit on a breadboard and while the connection is usually created with jumper wires. To connect the Arduino to the model railway, a more durable interface is needed.

Stephan Höhrmann redesigned the wiring of the model railway periphery as part of his diploma thesis. Each component is equipped with a dedicated data line which is connected to the controlling infrastructure. The data lines are equipped with *Platinen-Steckkontakt* (PSK)-connectors with two, three or five pins, depending on the type of component. These connectors are shown in Figure 3.6.

To connect the model railway peripery to the new Arduino controller boards, an interface board for each Arduino was designed and built. The combination of the microcontroller board and the interface board is shown in Figure 3.7a.

The interface boards are hand-soldered on standard veroboard. The components used are all passive components, as in sockets for the PSK connectors, pin headers for connecting the additional controller periphery and some resistors. The schematic of the interface board is shown in Figure 3.7b.

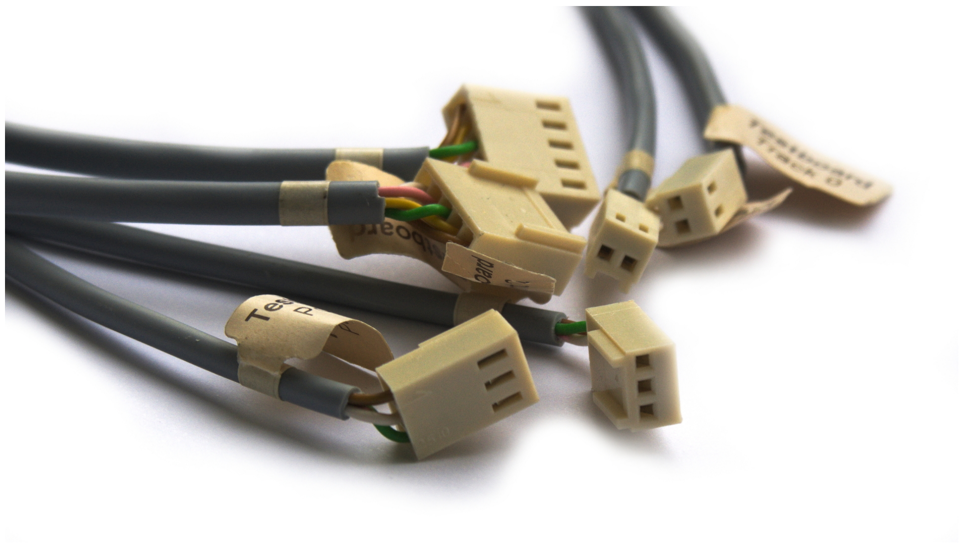
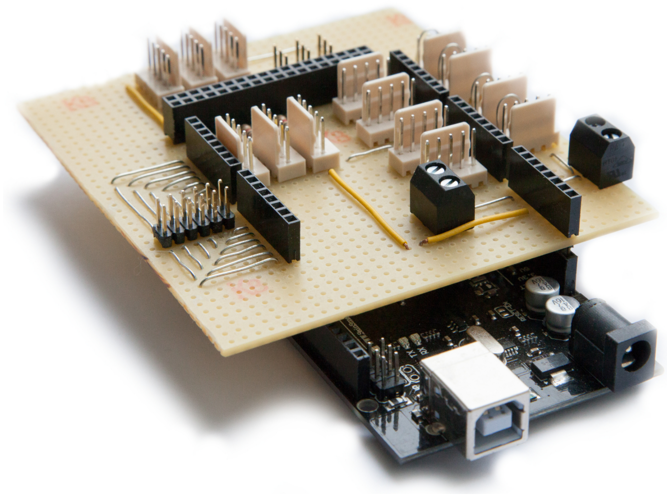
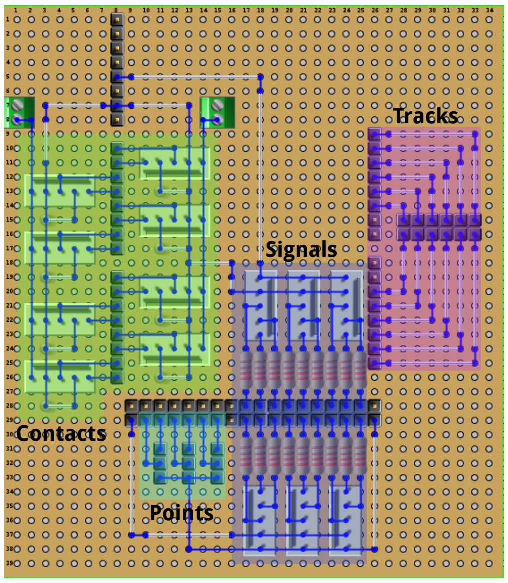


Figure 3.6. Various PSK connectors for model railway components



(a) Sainsmart Mega 2560 with interface board



(b) Schematic design of the interface board

Figure 3.7. Schematic design and implemented interface board

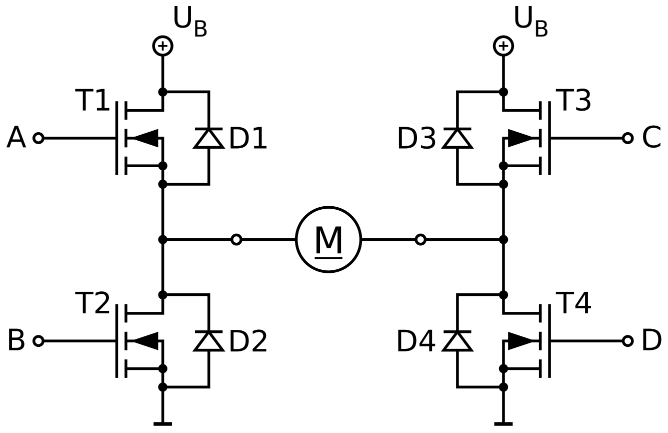
The interface boards use stackable pin headers to connect to the Arduino boards. These pin headers provide a male pin connector at the bottom of the board which plugs into the pin headers of the Arduino. On the top side of the interface board the same female headers are provided as on the Arduino. These pin headers can be used to connect additional hardware to free pins of the Arduino or to connect measuring equipment, e.g., when debugging the installation.

In the schematic view of the interface board four regions are highlighted. The leftmost regions, highlighted in green, marks the connectors for the contact inputs. These inputs are connected to the analog input of the Arduino board which are configured as analog input with an internal pullup resistor enabled. The second pin of each socket is connected to groud with a pulldown resistor of 1 kiloohm.

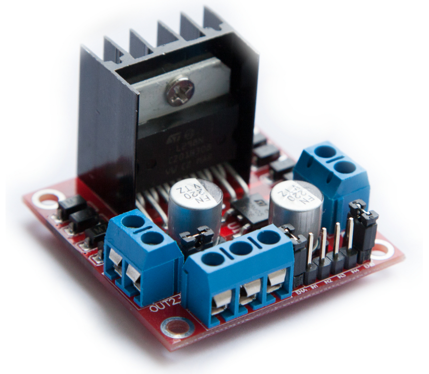
In the small region at the bottom, highlighted in cyan, pin headers for the relay modules, described in Section 3.4.3, are placed. These pin headers are connected to digital output pins of the Arduino board.

In the next region, highlighted in blue, PSK connectors for signal outputs are placed. Each of the signal connectors is equipped with three pins for the individual cathodes of the signal LEDs. These pins are connected to digital output pins of the Arduino board through a series resistor of 220 ohms. The other pins are connected to ground and 5V supply voltage respectively.

The region on the right side, highlighted in pink, provides a pin header for connecting the H bridge driver modules, described in Section 3.4.2. The top row of the pin header is connected to PWM pins of the Arduino board, controlling the speed of the trains. The bottom row is connected to digital output pins which function as reference pins for the PWM signal and manage the traveling direction of the trains.



(a) H bridge transistor setup



(b) Dual H bridge driver module

Figure 3.8. Track segment driver module

One interface board provides connectors for up to 8 Reed contacts, 6 switch points, 6 signals and 6 tracks.

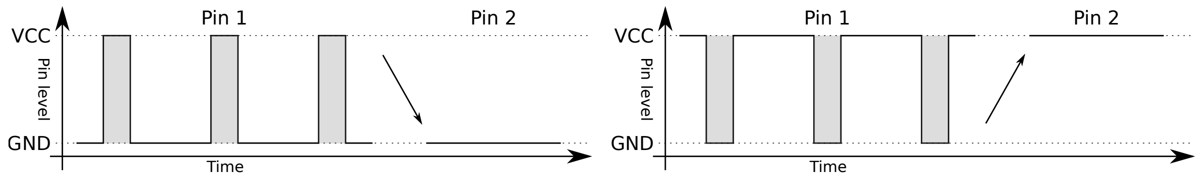
3.4.2 Track Segment Drivers

As already noted in Section 2.2.1, the tracks are powered by a dedicated 12V power supply. The microcontroller generates a PWM-signal at a voltage of 5V and with very low current. This low-voltage PWM-signal needs to be replicated with higher voltage and current.

To accomplish this, a dual H bridge driver module, based on a L298N chip, is used. This module combines the general setup of two H bridges, shown in Figure 3.8a, with a dedicated control logic to prevent short circuits. This control logic consists of an inverter, which prevents the opening of transistors T1 and T2 (or T3 and T4 respectively) at the same time. Additionally the inputs are regulated by *turn-on delay* circuits which prevent short circuits during the switching, due to the delay until the transistor blocks after the control input was turned off.

The H bridge module provides two separate channels which can be used to either control two independent motors or drive one four-phase step motor. Each of the channels is equipped with a dedicated enable signal which can be used to enable or disable the complete channel. These enable signals are permanently set to a high logic level by a jumper on the board. Each channel is controlled by two input pins. The first pin controls the inputs A and B shown in Figure 3.8a while the second pin controls the inputs C and D. If these pins are set to a high logic value, the top transistors T1 or T3 are opened and if the pins are set to a low logic value, the bottom transistors T2 or T4 are opened.

Three modes of operating can be distinguished. The first mode is supplying a PWM-signal to the first pin and setting the second pin to a low level. In this mode the powered train will drive forward, according to the PWM duty cycle. Figure 3.9a shows this operating mode and the corresponding active duty cycle. The second operating mode is used to drive the trains backwards. For this mode the first pin is set to a PWM-signal and the second pin is set to high logic level as shown in Figure 3.9b. In this mode the PWM duty cycle is inverted due to the



(a) Pin setup during forward operation **(b)** Pin setup during backward operation

Figure 3.9. Pulse-width modulation duty cycles

comparison against the high reference pin. The third operating mode is implemented by setting both pins to the same logic value. In this mode the motor of the train is braking, because it is short circuited. The rotating motor of the train creates a magnetic field near a conductive circuit, which, according to Lenz's law, results in an inverse magnetic field, slowing the train down. This process creates a substantial amount of heat but all components in this circuit are able to deal with this heat.

The H bridge driver needs a 5V supply to power its logic components. The module is equipped with a 5V voltage regulator to provide this. In this mode of operation the power supply of the module should be lower than 14V, because the voltage regulator would get too hot otherwise. Additionally the H bridge module and the Arduino have to share a common ground level. For this reason the grounds of central power supply of the logic parts and the power supply for the tracks are connected directly at the power supplies. Alternatively this 5V voltage can be supplied from the Arduino by connecting a corresponding pin of the Arduino to the 5V clamp of the module. In this mode the jumper on the module, which enables the voltage regulator, needs to be removed.

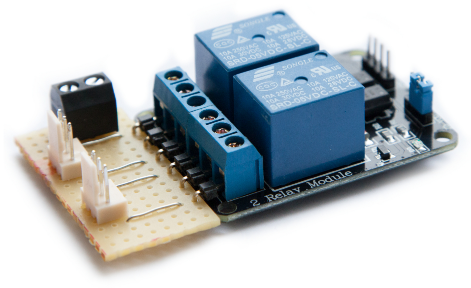
To form compatibility between the screw clamps of the module and the PSK print connectors of the model railway, a small hand-soldered circuit board is used.

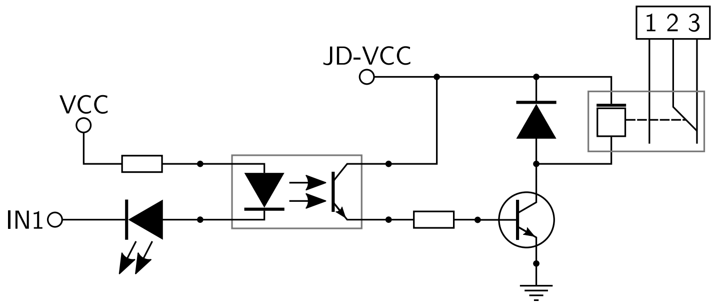
3.4.3 Switching Point and Light Relays

Figure 3.10a shows one of the relay boards which are used to control the switching nodes of the model railway. The used boards provide two independent channels with a galvanic isolation and a relay each. Figure 3.10b shows the general circuit of one of these channels.

The general function of the relays is a 2-way switch which connects the power supply with one of two outputs. The respective output is then connected to the signal wire of the switching point actuators. Section 2.2.2 described the general function of the switching points and mentioned the fact that the switching points generated significant electromagnetic interferences. To isolate the logic boards from these interferences a galvanic isolation is used, shown in Figure 3.10b in the central box. The primary side of this isolation consists of a LED and is powered by the Arduino. The secondary side is powered by the power supply which powers the switching point actuators.

The relays are able to support up to 10A of current at voltages of up to 250V. These relays are substantially larger than the relays used by Stephan Höhrmann, resulting in a higher load on the power supply.


(a) Hardware relay module



(b) Circuit of one relay channel

Figure 3.10. 2-channel relay module

Like with the H bridge driver modules, a small circuit board is used to connect the PSK connectors of the switches to the screw clamps. The relays are also used to control the lights of the railway. In this application one of the two outputs of the relay is not connected.

3.5 Controller Environment

To build a complete controller infrastructure, while keeping the installation maintainable and easy to use, some caution needs to be used while installing the hardware. The combination of controlling components into complete controlling modules as well as the mounting of the hardware is shown in Section 3.5.1. The surrounding Ethernet communication is briefly described in Section 3.5.2. Finally, Section 3.5.3 describes communication parameters which are needed to connect external controllers to the model railway.

3.5.1 Hardware Mounting

The third generation hardware was mounted on plywood bases, which were fixed to the edge of the model railway tables at an angle. This provided a uniform way of mounting the controller boards and provided a cable duct for the power supply of the controller, the Ethernet wires and the CAN bus.

This way of mounting has been kept and the new hardware is placed on similar plywood bases as the third generation. The new hardware consists of 12 nodes instead of 24 in the previous generation. For this reason some cables needed to be rerouted to fit to the new hardware locations. Each of the new controller nodes replaces two of the previous power electronics boards.

Figure 3.11 shows one of the new controller boards. On one controller board, an Arduino is combined with up to 3 H bridge driver modules and up to 3 relay modules. These modules are connected to the Arduino interface board using flat ribbon cable. All peripheral components of the model railway are connected to the controller board using the existing PSK connectors. The power supply is connected to the boards using lustre terminals.

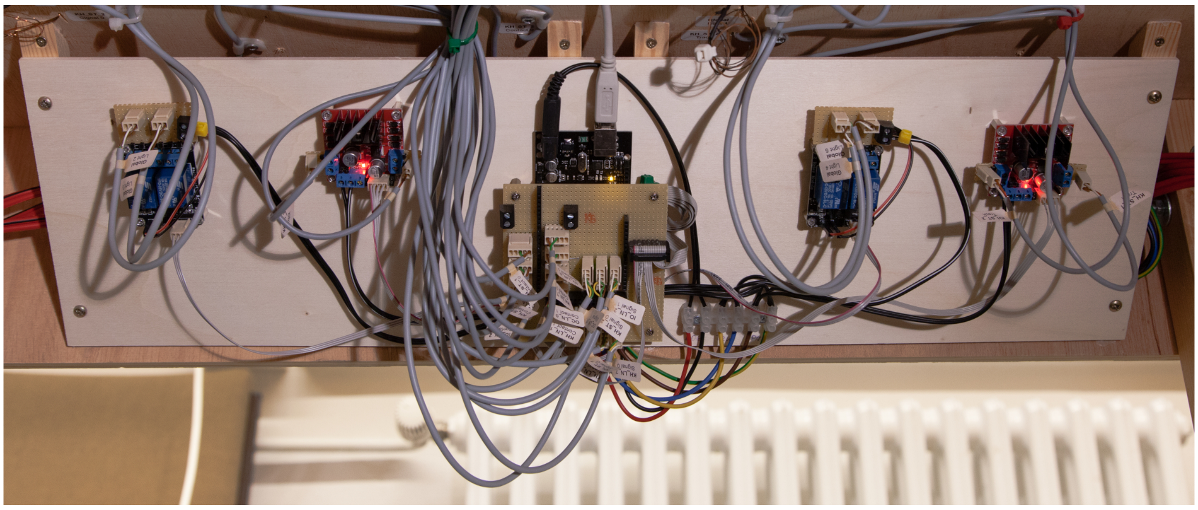


Figure 3.11. A controller node mounted below the model railway installation

To fit to the existing wiring of the model railway, these boards sometimes needed to be split in two parts. In these cases the *primary* board carries the Arduino and some parts of relays and track drivers, while the *secondary* board only carries relays and track drivers.

3.5.2 Ethernet Networking

In the third generation of the model railway, each of the 24 PC/104 nodes was connected via Ethernet network. Due to the reduced number of coordination nodes the amount of hosts in the Ethernet network was substantially reduced. As far as possible, existing components of the previous installation, like CAT5 cables or switches, were reused for the new setup.

Each of the Raspberry Pi nodes is connected to the central Ethernet switch and a new router, which connects the model railway to the university network, as described in Section 3.5.3. The four Raspberry Pi of the main installation are available as `railPi0` to `railPi3` with the IP adresses `10.6.6.10` to `10.6.6.13`. The fifth Raspberry Pi, which controls the testing circle, is configured as `railPi4` with IP address `10.6.6.14`.

3.5.3 External Controllers

The external connection of the model railway is provided by a newly installed Ubiquity Networks EdgeRouter Lite. This router replaces the previous gateway server, which connected the model railway via virtual private network (VPN) to the internal network of the embedded systems group. This setup proved to be problematic, because the VPN caused severe overhead. Due to this overhead, the delay of packets was too high and time constraints were sometimes not met.

The new router is not integrated into the internal network of the Embedded Systems Group anymore, but directly connected to network of the university. The external IP address of the router is set to `134.245.42.135` and can be used to connect a controller to the model railway from inside the university network. Connections from outside the university network are rejected.

To connect to the Railway Daemon on the Raspberry Pi nodes the external controller connects to the external IP of the router on the ports 20010 to 20013. These ports are forwarded via network address translation (NAT) to port 10000 on the Raspberry Pis. The router is configured to support *hairpin NAT*, so that even a controller node from inside the model railway network can connect to the external IP address of the router.

Chapter 4

Serial Communication

This chapter describes the details of the serial communication interface between the Raspberry Pis and the Arduino boards. This serial communication is established through the USB connection between the Arduino and the Raspberry Pi.

Section 4.1 describes the general parameters and the message format for the serial communication. Subsequently, Section 4.2 details the individual messages which are exchanged between the Raspberry Pi and the Arduino.

4.1 General Communication Parameters

The serial communication between the Raspberry Pi and the Arduino uses a symbol rate of 19200 baud and a standard configuration of eight data bits, no parity bit, and one stop bit (8N1).

The Raspberry Pi automatically creates a serial `tty`-interface which can be used as a direct connection to the Arduino. The Arduino board is equipped with a separated dedicated microcontroller, an Atmel ATmega16u2, which takes care of the UART interface and presents the messages to the main microcontroller. If this chip should not be used to communicate to the Arduino, a standard serial console adapter can be connected to the pins 0 and 1 of the Arduino. A secondary serial connection is available by connecting a standard serial console port to the pins 14 and 15 of the Arduino board.

When the serial interface is opened by the Raspberry Pi, the microcontroller on the Arduino board is reset. This reset is performed to give the Raspberry Pi the option to trigger the ICSP of the board and program a new firmware to the microcontroller.

Table 4.1. General serial message format

Byte	Content	
[0]	**Header**	
	Bit 7-0:	0xF0 (Magic Byte)
[1]	**Opcode**	
	Bit 7:	0 = Request
		1 = Response
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	Function code, 0 to 10
[2+]	**Optional parameters**	
[N]	**Checksum**	

Table 4.2. Valid functions and function parameters

			Parameters	
Code	**Name**	**Function**	**Request**	**Response**
0	CMDCONNECT	Establish connection	1	0
1	CMDDISCONNECT	Close connection	1	0
2	CMDKEEPALIVE	Keep connection active	0	0
3	CMDSIGNALS	Set or read signals	1	1
4	CMDPOINTS	Set or read switch points	1	1
5	CMDCONTACTS	Read contacts	0	6
6	CMDTRACKS	Set or read tracks	2	2
7	CMDSENSORS	- Not implemented -	-	-
8	CMDGLOBAL	Exchange global state	12	0
9	CMDRESET	Reset the installation	0	0
10	CMDEEPROM	Access chip EEPROM	-	-

The possible messages between the Raspberry Pi and the Arduino and the general message format are adapted from the previous generation of the model railway [Höh06]. The general message format has been keept and it is sketched in Table 4.1. The names and the encoding of the possible functions has been kept the same as in the third generation, to limit the needed adjustments, if existing systems like the TTP nodes should be connected to the Arduino boards. Some adjustment has been done at the parameters, because the previous message format was not scaling well with additional component ports.

Each message starts with a fixed magic byte 0xF0. In the third generation this magic byte was not only used to start the message, but also functioned as a signal to manage the access to the power electronics between the PC/104 nodes and the TTP powernodes. The second byte contains a bit field, consisting of a single byte to distinguish between requests and responses, a sequence number which is managed by the coordination node and the function code to determine the function of the request or response. Depending on the function, it is optionally followed by one or more parameters. The last byte of the message is the checksum of the message. This checksum is computed in a way that the sum of all bytes of the message is divisible by 256. The sequence number of the message is computed by the coordination node and incremented with each message. The Arduino uses the same sequence number to reply to this request. This way the responses of the Arduino can be matched with the requests.

The valid function codes and the corresponding amount of data parameters for each function are listed in Table 4.2.

4.2 Detailed Messages

Several of the functions listed in Table 4.2 require dedicated bit fields to address certain hardware and communicate the desired settings. These messages are described in the next subsections. Some of the messages are solely kept for backwards compatibility and do not implement a real function at the moment. If the second serial input is used for an alternative controller, these functions need to be reimplemented. The enumeration of the functions is kept the same as in the third generation to keep the needed adjustments of implementations on already existing hardware small.

4.2.1 Establish Connection

At first the coordination node needs to establish a connection to the Arduino by sending a CMDCONNECT message. Table 4.3 shows the detailed message format and the structure of the bit fields. While connecting, the periphery can be reset to start the controller in a known state. This process is kept for backwards compatibility. If the second serial connection were be used in the future, this connection process could manage the rights between the two coordination systems.

Table 4.3. Message format for CMDCONNECT request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	0 (CMDCONNECT)
[2]	**Periphery reset**	
	Bit 7-1:	Reserved / Not used
	Bit 0:	0 = Keep current periphery state
		1 = Reset periphery

The Arduino firmware responds to the request after the connection has been established. The response carries no additional data, as shown in Table 4.4.

Table 4.4. Message format for CMDCONNECT reply

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	0 (CMDCONNECT)

4.2.2 Close Connection

When the connection is not needed anymore, the coordination node can orderly close the connection by sending a CMDDISCONNECT message. Similar to the connection establishment, the periphery can be reset during this function. As detailed in Table 4.5, the message for CMDCONNECT and CMDDISCONNECT differ only in the opcode.

The Arduino acknowledges the closed connection and sends a reply, shown in Table 4.6, to the coordination node. No further messages are processed after this until a new connection has been opened.

Table 4.5. Message format for CMDDISCONNECT request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	1 (CMDDISCONNECT)
[2]	**Periphery reset**	
	Bit 7-1:	Reserved / Not used
	Bit 0:	0 = Keep current periphery state
		1 = Reset periphery

Table 4.6. Message format for CMDDISCONNECT reply

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	1 (CMDDISCONNECT)

4.2.3 Keep Connection Active

The power electronics board of the third generation did close the serial connection if no data was transmitted in a certain amount of time. This was implemented to enable the handover from one controlling system to another. To send data without touching the controller state the message CMDKEEPALIVE was introduced. The message format is shown in Table 4.7.

In the current implementation, the Arduino does not automatically close the connection after a certain amount of time, therefore this message is not strictly necessary. It is primarily kept for backwards compatibility. If desired, a similar behaviour as the previous power electronics could be implemented in the Arduino firmware in the future. This would enable the Arduino to change to a safe state, if the controller seems to be crashed or otherwise faulty.

Table 4.7. Message format for CMDKEEPALIVE request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	2 (CMDKEEPALIVE)

The request is acknowledged and the Arduino replies with a simple reply without any additional data.

4.2.4 Set or Read Signals

Sending the message CMDSIGNALS enables the controller to access the state of a single signal of the model railway. As detailed in Table 4.8, the request takes one byte as parameter. This byte is used as a bit field, encoding the address of the signal, a write enable bit, and the new value of the signal.

Table 4.8. Message format for CMDSIGNALS request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	3 (CMDSIGNALS)
[2]	**Signal data**	
	Bit 7-5:	Signal number
	Bit 4:	Write enable
	Bit 3:	Reserved / Not used
	Bit 2-0:	Signal value

The Arduino board processes the request and, if the write enable bit is set, sets the signal to the new value. If the write enable bit is not set, the state of the signal is not changed. After the message has been processed, the Arduino sends a reply with the state of the siganal, as listed in Table 4.9.

Table 4.9. Message format for CMDSIGNALS reply

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	3 (CMDSIGNALS)
[2]	**Signal data**	
	Bit 7-5:	Signal number
	Bit 4-3:	Reserved / Not used
	Bit 2-0:	Signal mask

4.2.5 Set or Read Switch Points

The command CMDPOINTS modifies the state of a switching point and returns the current state. Similar to CMDSIGNALS, the function needs one byte of additional data. In this bit field the switching point is selected, a write enable bit can be set and the new value of the switching point is supplied, as shown in Table 4.10.

The Arduino processes the request and checks whether the write enabled bit is set. If this bit is set, the point is set to the new value. The response, detailed in Table 4.9, sends the state of the point back to the coordination node. This is either the unchanged old state, if the state was not changed, or the new state.

Table 4.10. Message format for CMDPOINTS request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	4 (CMDPOINTS)
[2]	**Point data**	
	Bit 7-5:	Point number
	Bit 4:	Write enable
	Bit 3-1:	Reserved / Not used
	Bit 0:	Point state

Table 4.11. Message format for CMDPOINTS reply

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	4 (CMDPOINTS)
[2]	**Signal data**	
	Bit 7-5:	Signal number
	Bit 4-1:	Reserved / Not used
	Bit 0:	Point state

4.2.6 Read Contact Events

By sending the message CMDCONTACTS, the coordination node prompts the Arduino to send the current state of the Reed contacts. As shown in Table 4.12, the request carries no additional data apart from the Opcode.

Table 4.12. Message format for CMDCONTACTS request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	5 (CMDCONTACTS)

The Arduino collects all data and prepares the response, as listed in Table 4.13. The first two data bytes of the response consist of the current state of each Reed sensor[1]. In the next two bytes the type of the last event of the Reed contacts is transferred. The last two data bytes of the response consist of the event counters of the Reed contacts. These event counters are integers between 0 and 3 and are incremented with each event.

[1]As a reminder: The Reed sensors are the two individual sensors in one Reed contact.

Table 4.13. Message format for CMDCONTACTS reply

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	5 (CMDCONTACTS)
[2]	**Current state of sensors**	
	Bit 7:	1 = CNT7/1 is currently closed
	Bit 6:	1 = CNT7/0 is currently closed
	Bit 5:	1 = CNT6/1 is currently closed
	Bit 4:	1 = CNT6/0 is currently closed
	Bit 3:	1 = CNT5/1 is currently closed
	Bit 2:	1 = CNT5/0 is currently closed
	Bit 1:	1 = CNT4/1 is currently closed
	Bit 0:	1 = CNT4/0 is currently closed
[3]	Same as byte 2, for CNT3 to CNT0	
[4]	**Last event type**	
	Bit 7-6:	Last event of CNT7
	Bit 5-4:	Last event of CNT6
	Bit 3-2:	Last event of CNT5
	Bit 1-0:	Last event of CNT4
		Event type encoding
		0 = No event
		1 = Forwards
		2 = Backwards
		3 = Unknown direction
[5]	Same as byte 4, for CNT3 to CNT0	
[6]	**Event counter**	
	Bit 7-6:	Event counter for CNT7
	Bit 5-4:	Event counter for CNT6
	Bit 3-2:	Event counter for CNT5
	Bit 1-0:	Event counter for CNT4
[7]	Same as byte 6, for CNT3 to CNT0	

4.2.7 Set or Read Tracks

Similar to CMDPOINTS and CMDSIGNALS, the command CMDTRACKS allows access to the state state of a track driver. As shown in Table 4.14, the track data byte provides a write enable bit, which controls whether the new state should be set or only the active state should be read. The desired track direction is transferred as the last bits of the track data byte. The last byte, apart from the checksum, consists of the PWM duty cycle of the track driver. This duty cycle can be chosen between 0 and 255.

The Arduino processes the request and first checks the write enable bit. If this bit is set, the new data for the track driver is set. The requested PWM duty cycle and the requested direction are stored in a local variable before they are submitted to the track driver. This is done to be able to return a value when the track driver is read. For pure digital outputs the value of the pin can easily be read back and does not need to be stored in the firmware. The PWM signal

Table 4.14. Message format for CMDTRACKS request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	6 (CMDTRACKS)
[2]	**Track data**	
	Bit 7-5:	Track number
	Bit 4:	Write enable
	Bit 3-2:	Reserved / Not used
	Bit 1-0:	Track direction
		Track direction encoding
		0 = Off / Brake
		1 = Forwards
		2 = Backwards
[3]	**PWM duty cycle**	
	Bit 7-0:	PWM duty cylce, 0...255

cannot be read back from the pin, because the Arduino can only access the current logic level of the pin but not the duty cycle. After processing the command, the Arduino replies with a message, as shown in Table 4.15.

Table 4.15. Message format for CMDTRACKS response

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	6 (CMDTRACKS)
[2]	**Track data**	
	Bit 7-5:	Track number
	Bit 4-2:	Reserved / Not used
	Bit 1-0:	Track direction
		Track direction encoding
		0 = Off / Brake
		1 = Forwards
		2 = Backwards
[3]	**PWM duty cycle**	
	Bit 7-0:	PWM duty cylce, 0...255

4.2.8 Read Track Sensors

The third generation of the model railway was able to detect train engines on the track segments by measuring the resistance between the two rails of the tracks. The new hardware uses the driver modules described in Section 3.4.2 which do not support this measurement. For this reason, the function CMDSENSORS is not implemented. Nevertheless, the function is kept in this listing, and the list of available functions, to reduce the needed adjustments of existing hardware like the TTP Powernodes.

4.2.9 Exchange Global State

The CMDGLOBAL function is designed to set all elements connected to one Arduino board. The function takes new values for all elements at once, the order being listed in Table 4.16. This function does not provide a read-only mode like the dedicated functions for single components. This function is primarily aimed at controllers that store a local state in the controller and write their state to the hardware in regular intervals.

Table 4.16. Message format for CMDGLOBAL request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	8 (CMDGLOBAL)
[2]	**Signal data**	
	Bit 7-4:	Signal data for SIG5
	Bit 3-0:	Signal data for SIG4
[3]	Like byte 2 but for SIG3 and SIG2	
[4]	Like byte 2 but for SIG1 and SIG0	
[5]	**Switch point data**	
	Bit 7-6:	Reserved / Not used
	Bit 5:	Setting for point PNT5
	Bit 4:	Setting for point PNT4
	Bit 3:	Setting for point PNT3
	Bit 2:	Setting for point PNT2
	Bit 1:	Setting for point PNT1
	Bit 0:	Setting for point PNT0
[6]	**Track setting**	
	Bit 7-6:	Direction setting for TRK5
	Bit 5-4:	Direction setting for TRK4
	Bit 3-2:	Direction setting for TRK3
	Bit 1-0:	Direction setting for TRK2
[7]	Bit 7-6:	Direction setting for TRK1
	Bit 5-4:	Direction setting for TRK0
	Bit 3-0:	Reserved / Not used
[8]	**PWM duty cycles**	
	Bit 7-0:	PWM duty cycle for TRK5
[9]	Like byte 8 but for TRK4	
[10]	Like byte 8 but for TRK3	
[11]	Like byte 8 but for TRK2	
[12]	Like byte 8 but for TRK1	
[13]	Like byte 8 but for TRK0	

The Arduino receives the request and sets all components to the new state. As shown in Table 4.17, the response carries no data.

Table 4.17. Message format for CMDGLOBAL response

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	8 (CMDGLOBAL)

4.2.10 Reset the Installation

The firmware can be reset to represent the initial state again by sending the message CMDRESET. In this state all tracks are set to stop, the signals are turned off and all switching points are set straight. The resulting state of the controller is similar to clean start of the controller or a hardware reset of the Arduino nodes with the difference that the established connection between the Arduino and the coordination node is not disconnected. Neither the request, shown in Table 4.18, nor the response, shown in Table 4.19, carry additional data.

Table 4.18. Message format for CMDRESET request

Byte	Content	
[1]	**Opcode**	
	Bit 7:	0 (Request)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	9 (CMDRESET)

Table 4.19. Message format for CMDRESET response

Byte	Content	
[1]	**Opcode**	
	Bit 7:	1 (Response)
	Bit 6-4:	Sequence number, 0 to 7
	Bit 3-0:	9 (CMDRESET)

4.2.11 Access the EEPROM

The Arduino can use up to 4kB of Electrically Erasable Programmable Read-Only Memory (EEPROM) memory. This memory is persistent across restarts and can be used to store arbitrary data.

The third generation used its EEPROM to store multiple error counters. If in a Reed contact only one sensor activates, the second sensor is assumed to be faulty. These events were stored in the EEPROM and could be read with this command.

This kind of error counter can also be implemented on the current hardware but currently no EEPROM counters are implemented in the firmware.

Chapter 5

High-Level Communication and Usage

The high-level controller structure consists of multiple different parts, providing methods for controlling the model railway as well as a configuration and testing interface for the railway components. Figure 5.1 shows the primary parts of the high-level controller implementation and the interactions between them.

To provide an easy way of testing the hardware and to allow updates of the configuration without changes to the API, a web-based configuration and testing interface is provided by the Raspberry Pis. The function and the options of this interface is described in Section 5.1. Most commonly, the railway controller will run on a separate computer, e.g., a laptop connected to the railway network or one of the workstations in the railway laboratory. This Client gains access to the model railway through the Railway Daemon, serving as an adapter between the serial communication, detailed in Chapter 4, and the Transmission Control Protocol (TCP)-based communication. The general function of the Railway Daemon is shown in Section 5.2, and the communication with the Daemon is shown in Section 5.3. Finally, Section 5.4 shows the options of the Railway API, the C-API which is used by the controller.

5.1 Configuration and Testing Interface

The Raspberry Pis provide their configuration and testing interface through a webserver. The interface is implemented using HTML, JavaScript and PHP. The server runs locally on all Raspberry Pis and handles the configuration of the Arduinos connected to the individual Raspberry Pi.

5.1.1 General appearance and function

The interface shows the detected Arduinos and the stored configuration for each Arduino in a dedicated tab of the interface. Each Arduino is identified with its serial number because this identifier, unlike device names or USB port numbers, is kept constant across restarts. Each tab shows the current configuration state with a symbol next to the serial number, as shown in Figure 5.2. If the Arduino is detected and configured, a green checkmark is shown in the tab. A yellow questionmark is used for Arduinos which are detected by the Raspberry Pi but are not configured yet. If the tab shows a red cross, a configuration for the Arduino is present but the board is not connected.

Without authentication, the interface allows for read-only access to the configuration and provides the option to test each component individually. The components for each Arduino are listed and the configured name is shown next to the component port name.

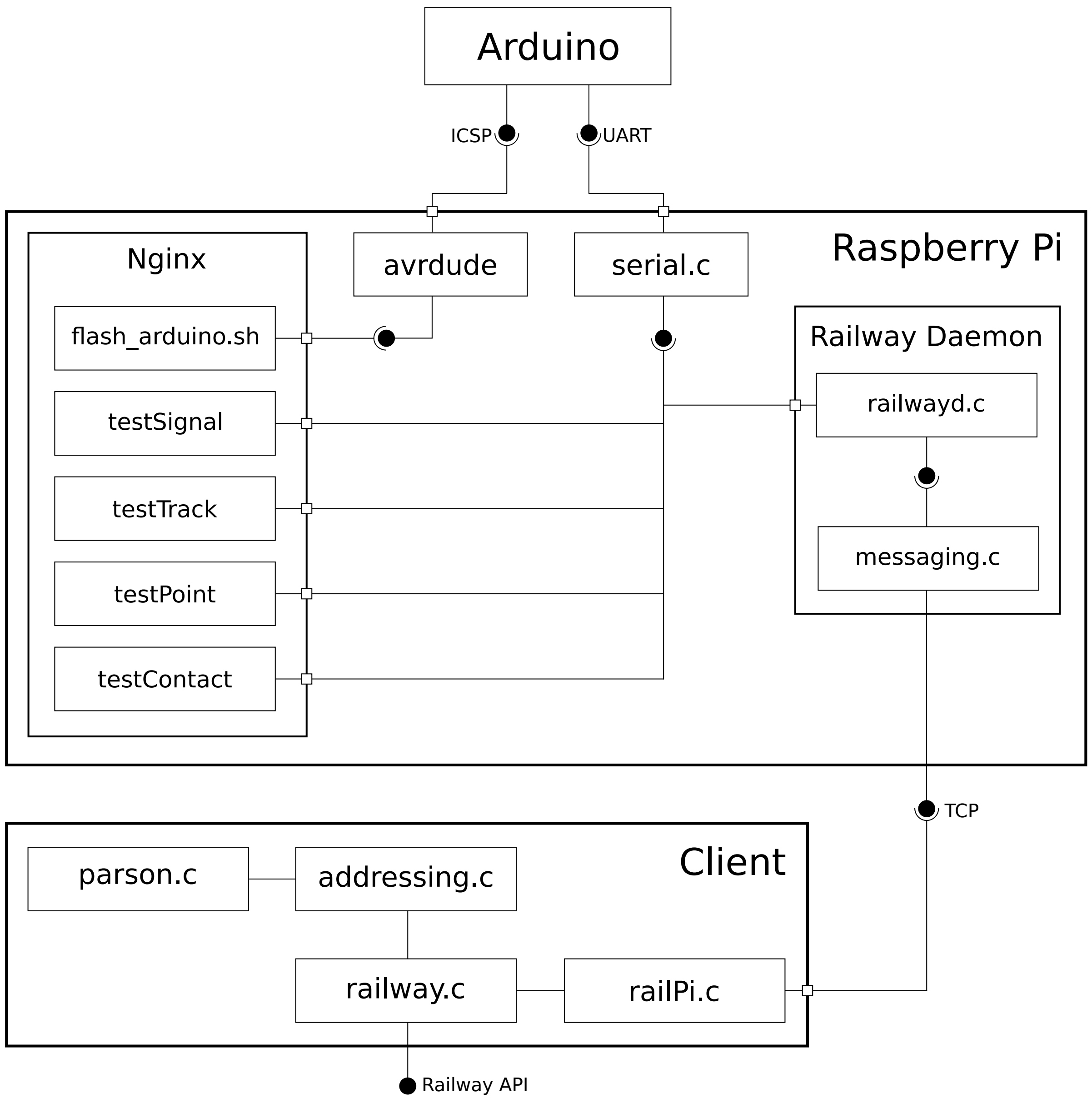
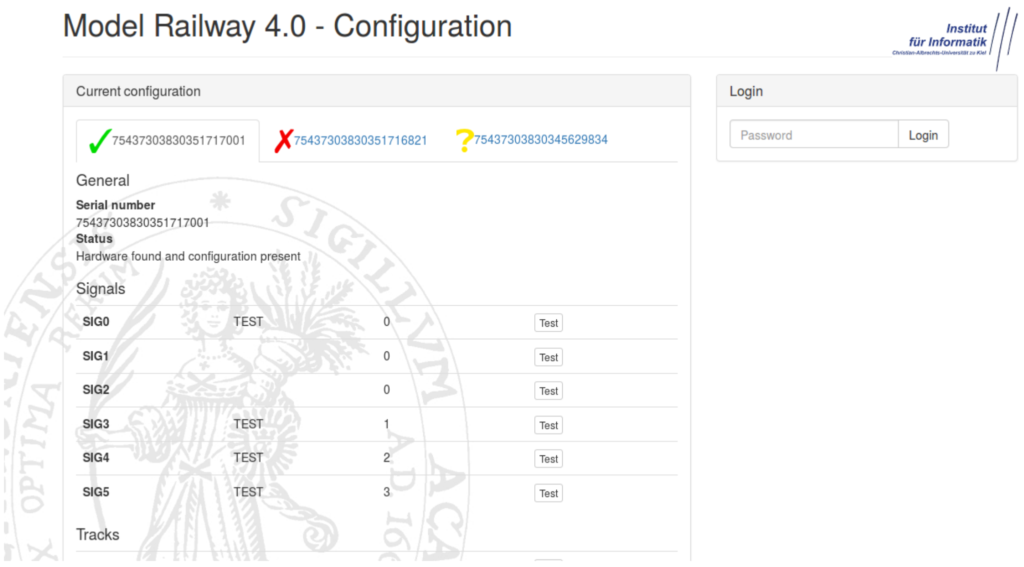


Figure 5.1. High-level controller components



Model Railway 4.0 - Configuration

Current configuration

✓ 75437303830351717001 ✗ 75437303830351716821 ? 75437303830345629834

General

Serial number
75437303830351717001
Status
Hardware found and configuration present

Signals

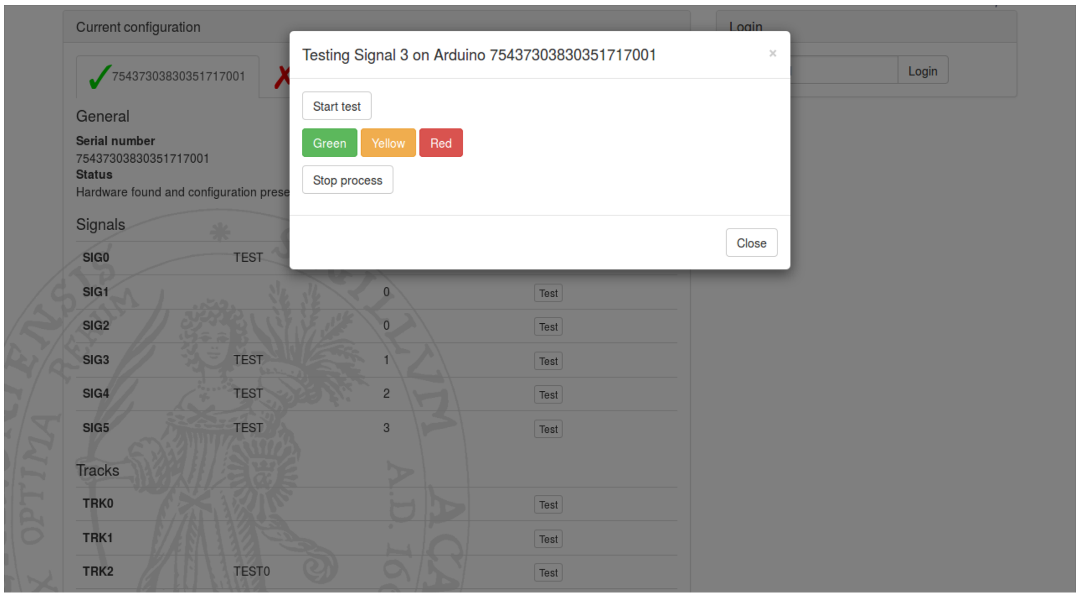
SIG0	TEST	0	Test
SIG1		0	Test
SIG2		0	Test
SIG3	TEST	1	Test
SIG4	TEST	2	Test
SIG5	TEST	3	Test

Tracks

Login

Password Login

Figure 5.2. Default view of the web interface



Testing Signal 3 on Arduino 75437303830351717001

Start test

Green Yellow Red

Stop process

Close

Current configuration

✓ 75437303830351717001

General

Serial number
75437303830351717001
Status
Hardware found and configuration prese

Signals

SIG0	TEST		
SIG1		0	Test
SIG2		0	Test
SIG3	TEST	1	Test
SIG4	TEST	2	Test
SIG5	TEST	3	Test

Tracks

TRK0		Test
TRK1		Test
TRK2	TEST0	Test

Login

Login

Figure 5.3. Signal testing dialog in the web interface

5.1.2 Component Testing

Testing can be done through a simple interface, like the one shown in Figure 5.3. The testing is implemented through dedicated C-programs on the Raspberry Pi, which use a shared implementation of the serial API described in Chapter 4. Upon startup, the testing interface passes the serial number and the component number to the testing program. This program then connects to the Arduino and waits for user input. The testing program and the webserver communicate through a temporary file. When the user changes the desired state, a JavaScript function is executed, which sends the new desired state to the webserver. The webserver writes the desired component state to the temporary file, which is repeatedly read by the testing program. The testing program runs until a special command, depending on the component being tested, is written into the temporary file, upon which the testing program disconnects from the hardware and terminates.

To test the Reed contacts the process is slightly modified because the communication between the testing program and the web interface needs to be bidirectional. The contacts need to communicate their contact events to the user and the user needs the option to stop the test. For this reason, a second temporary file is used. The control commands from the web interface to the testing program are written to the first file, while the testing program writes the events to the second file, which is repeatedly read by the web interface.

This inter-process communication (IPC) loosely resembles the general principle of Unix domain sockets [SR13]. The communication through text files was chosen for the quick prototypes of the initial testing programs but was not refactored or reimplemented for the final submission. A future improvement of these testing programs should implement IPC using Unix domain sockets or a similar method.

5.1.3 Configuration Access

The interface allows the user to authenticate with a password. Upon authentication, the interface allows editing the stored configuration. For each component a name can be entered, as well as a second identification if multiple components can share a name. For example, the tracks are only assigned a name, because each block is driven by exactly one hardware component. For signals or Reed contacts the identification consists of two parts. The first part is the name of the track block, while the second part determines the end of the block. The name and numbering scheme has been kept consistent with the previous generations. Contacts take the same name as their track block, the contacts are numbered in order of passing in main travel direction, starting at 0 for the first contact.

The configuration of the Arduinos is stored on the microSD card of the Raspberry Pi. The data is encoded as JavaScript Object Notation (JSON) data, resulting in a small file while still being human-readable. The web interface currently support no option to transfer the configuration from one Arduino to another, like it would be useful after replacing a faulty Arduino. Alternatively, the serial number can be changed in the configuration file using remote access to the Raspberry Pi or by removing the microSD card from the Raspberry Pi and editing the file on a pc, using a card reader to access the file.

5.1.4 Firmware Upload

In addition to testing and managing the configuration, the web interface provides the option to upload and program a new firmware to the Arduinos. The file is uploaded through the web interface and placed in a temporary directory. The filename and the serial number of the Arduino are then passed to the tool *avrdude*, which is part of the Arduino IDE. This tool checks the file and the connection and programs the Arduino afterwards. The success or errors during the programming are submitted back to the user through the web interface.

5.2 Railway Daemon

Most controllers will not run on the Raspberry Pis directly. Instead the controller is executed on a dedicated system which then connects to all Raspberry Pis. This connection is established using TCP and the messages detailed in Section 5.3. These messages are received by a daemon process on each Raspberry Pi, interpreted by this process and then relayed to the Arduinos.

Upon startup, the Raspberry Pis start the Railway Daemon, which then listens for connections on port 10000. It should be noted that the Raspberry Pis are configured to use IPv6 as well as IPv4. The implementation of the Railway Daemon only has to manage IPv6 sockets as these sockets are fully backwards compatible. The network of the university is currently only using IPv4, so this implementation is merely a precaution for improvements of the network.

5.3 Network Communication Protocol

The network-based communication is used between the Raspberry Pis and the system running the controller. At first, Section 5.3.1 documents the general message format and general communication parameters. In Section 5.3.2 the valid messages are listed and described.

5.3.1 General Network Communication

The protocol uses TCP as the unterlying protocol, which guarantees reliable message exchange. The network protocol defines valid message types, similar to the serial protocol in Chapter 4. Unlike in the serial communication no bitfields are used to transfer the data but complete bytes instead. By using bitfields the transferred data could possibly be reduced by one or two bytes which is almost insignificant, regarding the protocol overhead of 20 bytes for TCP and 40 bytes for IPv6.

Table 5.1. General network message format

Byte	Content	
[0]	**Header**	0x42 (Magic Byte)
[1]	**Function**	0...8
[2+]	**Optional Parameters**	
[N]	**Closing Byte**	0x00

The general message format is described in Table 5.1. Each message starts with a single message header byte, similar to the magic byte in the serial communication. The second byte is used to distinguish the function. Depending on the function, up to 23 additional bytes are needed. These bytes carry the serial number of the controlled Arduino, the component port number and the desired state. Each message is terminated with a final null byte. It is not needed to check the messages with a dedicated checksum, like at the serial interface, because the underlying TCP already takes care of correct transmission.

One problem arose with the transmission of null bytes in the body of the messages, i.e., as the component port. These null bytes cause the socket API to end the message before it is sent completely. For this reason all data bytes are transferred with all bits inversed. A possible future improvement of the network communication should address this issue and implement a more stable messaging layer.

5.3.2 Detailed Network Messages

The valid functions are listed in Table 5.2. The messages loosely resemble the functions from Chapter 4.

Table 5.2. Valid functions and function parameters

Code	Name	Function	Parameters
1	MESSAGECONNECT	Connect RailPi to Arduino	0
2	MESSAGECONFIG	Request config from RailPi	0
3	MESSAGESIGNAL	Set signal	22
4	MESSAGEPOINT	Set switching point	22
5	MESSAGETRACK	Set track segment	23
6	MESSAGECONTACT	Request contact events	22
7	MESSAGEDISCONNECT	Disconnect RailPi from Arduino	0
8	MESSAGESHUTDOWN	Shut Railway Daemon down	0

As shown in Figure 5.1, these messages are sent by the modules railPi on the client side and are received by the module messaging in the Railway Daemon.

Connect RailPi to Arduino The message MESSAGECONNECT prompts the Railway Daemon to establish the serial connection to all Arduinos. As already mentioned in Section 4.1, the Arduinos are reset when this connection is opened.

Request Config from Railpi By sending the message MESSAGECONFIG, the client promts the Raspberry Pis to transmit their component configuration to the client. The Raspberry Pis respond to this message with a message containing the configuration as a JSON object.

Set Signal The message MESSAGESIGNAL requires additional parameters for addressing the signal and the desired signal state. The message format is described in Table 5.3.

Table 5.3. Format of MESSAGESIGNAL request

Byte	Content	
[0]	**Header**	0x42 (Magic Byte)
[1]	**MESSAGESIGNAL**	0x02
[2-21]	**Arduino Serial Number**	Hex String
[22]	**Port Number**	0...5
[23]	**Signal State**	0...4
[24]	**Closing Byte**	0x00

The Raspberry Pi processes the message and sends the request to the Arduino, using the serial interface. This message currently supports no way of receiving the current state of the signals. This is a drawback in comparison to the previous generation of the hardware and should be resolved in a future iteration.

Set Switching Point For setting switching points the same message format as for signals is used. The data byte 23 carries the desired point state, which can either be 0 or 1.

Set Track Segment The message MESSAGETRACK sets a track segment to a new state. This message is similar to MESSAGESIGNAL but needs an additional data byte for the PWM duty cycle. The detailed message format is shown in Table 5.4

Table 5.4. Format of MESSAGETRACK request

Byte	Content	
[0]	**Header**	0x42 (Magic Byte)
[1]	**MESSAGETRACK**	0x04
[2-21]	**Arduino Serial Number**	Hex String
[22]	**Port Number**	0...5
[23]	**Track Direction**	0...2
[24]	**PWM Duty Cycle**	0...254
[25]	**Closing Byte**	0x00

The PWM duty cyle is limited to 254 due to the transmission problems mentioned in Section 5.3.1. If the duty cycle were set to 255, the inverted data byte would form a null byte, which could again cause transmission errors.

Request Contact Events By sending the message MESSAGECONTACT, the user program can query the contact events from a single Reed contact. The message format is similar to the format for setting signals or points. If the data byte is set to 1 the contact events are cleared on the Raspberry Pi.

Table 5.5. Format of MESSAGECONTACT request

Byte	Content	
[0]	**Header**	0x42 (Magic Byte)
[1]	**MESSAGECONTACT**	0x06
[2-21]	**Arduino Serial Number**	Hex String
[22]	**Port Number**	0...7
[23]	**Remove Event**	0/1
[24]	**Closing Byte**	0x00

The Raspberry Pi responds to this message by sending the type of the last event back to the client.

Disconnect RailPi from Arduino This message causes the Raspberry Pi to close the serial connection to the Arduino boards and close the connection between the client and the Raspberry Pi. When closing the connection to the Arduinos, the model railway is reset, setting all components to their safe state. The Arduinos then await another connection, to start again.

Shut Railway Daemon Down This message is solely used during development and should probably never be used during normal usage of the model railway. Sending this message causes the Railway Daemon to terminate, requiring shell access or a reboot of the Raspberry Pi to start again. Without the Railway Daemon process running, the model railway is not usable for most controllers.

5.4 Railway API

The Railway API provides a high-level interface for user programs. The current implementation provides a subset of the API of the third generation of the model railway. The function names and signatures are kept stable from the third generation. This allows legacy controllers that were developed for the third generation of the model railway to be executed on the new installation, if they do not use functions that were dropped in the new generation. In addition to the functions listed here, the third generation Railway API provided an event-based interface for the model railway components. These functions were rarely used and have not been implemented in the current generation. If needed, these functions could be reimplemented to use a form of event-based system with the new hardware. Because the track sensors are not available anymore in the new hardware, the functions for checking the positions and velocities of the trains have also been removed.

The API provides two kinds of functions. The first group of functions manages the connection, the configuration and takes care of correct addressing. These functions are listed Section 5.4.1. The second type of function provides access to individual railway components via dedicated functions.

5.4.1 Initialization and hardware

To begin the work with the railway system, the Railway API needs to be initialized. This function prepares memory which is used for handling communication and storing the local state of the model railway.

```
struct railway_system *railway_initsystem(struct railway_hardware *hardware)
```

The parameter `hardware` is only kept for legacy code and is not interpreted by the API. This function does not establish the connection to the Raspberry Pis.

The connection is created by the following function, which takes the `railway_system` that was returned from the previous function. The two other parameters are, again, kept for compatibility with legacy code.

```
int railway_openlinks_udp(struct railway_system *railway, char *hostformat, char *device)
```

It should be noted that although the function name suggests otherwise, the connection is established using TCP. The name has been ported from the previous generation. The address of the model railway is currently not taken from the parameter of the function, but defined statically in the header of the API.

After the connection has been established, the configuration is automatically read from the Raspberry Pis. The configuration is sent by the Raspberry Pis as a JSON object and parsed by the Railway API. The API then creates a lookup table for each type of component and stores the corresponding configuration entries in these lookup tables. The lookup tables are then used by the dedicated hardware control functions to determine the correct Raspberry Pi, the Arduino serial number as well as the component port.

After the client has established a connection to the Raspberry Pis, it can then claim the Arduino board and start the controller logic.

```
int railway_startcontrol(struct railway_system *railway, unsigned mincycle, unsigned maxcycle)
```

This function uses the connections which have been established and are managed in the `railway_-system`. In the third generation, the parameters `mincycle` and `maxcycle` could be used to set limits for the execution time of the controller. These parameters were usually set to 0, resulting in no upper or lower limits. The current implementation does not use these parameters.

For each of the presented function an inverse function is available, which is used after the controller has been executed. These functions disconnect the Raspberry Pis from the Arduinos, disconnect from the Arduinos and clean the memory.

```
int railway_stopcontrol(struct railway_system *railway, int reset)
int railway_closelinks(struct railway_system *railway)
int railway_donesystem(struct railway_system *railway)
```

5.4.2 Controlling Hardware

After the connection has been established, the Railway API provides functions to control individual components of the model railway. These functions are a subset of the API developed in the third generation.

The addressing of railway components is encapsulated in the Railway API. Depending on the type of component, the components are either enumerated and identified by their number or the components are identified by the track block they are related to. The track blocks are identified by the circle they are on, the type of block (station, junction or lane), and a block number. These informations are combined to names like `IC_LN_2` for the third block of the inner circle lane. The names of all track blocks and the numbers of all other components are listed in the track schematic of the model railway.

The names of the track blocks have been defined in an `enum` and should be used directly in the related function parameters.

```
enum track {
  IC_JCT_0, IC_LN_0, IC_LN_1, IC_LN_2, IC_LN_3, IC_LN_4, IC_LN_5, IC_ST_0,
  IC_ST_1, IC_ST_2, IC_ST_3, IC_ST_4, IO_LN_0, IO_LN_1, IO_LN_2, KH_LN_0,
  KH_LN_1, KH_LN_2, KH_LN_3, KH_LN_4, KH_LN_5, KH_LN_6, KH_LN_7, KH_LN_8,
  KH_ST_0, KH_ST_1, KH_ST_2, KH_ST_3, KH_ST_4, KH_ST_5, KH_ST_6, KIO_LN_0,
  KIO_LN_1, OC_JCT_0, OC_LN_0, OC_LN_1, OC_LN_2, OC_LN_3, OC_LN_4, OC_LN_5,
  OC_ST_0, OC_ST_1, OC_ST_2, OC_ST_3, OC_ST_4, OI_LN_0, OI_LN_1, OI_LN_2
};
```

Because C implements this `enum` as integers internally, it can also be used to iterate over all blocks of the installation.

```
int signalexists(struct railway_system *railway, int block, int signaln)
int contactexists(struct railway_system *railway, int block, int contact)
```

These functions check if a particular signal or contact exists. This is useful if the controller iterates over all track blocks and sets signals or queries contacts. This would result in errors because some of the track blocks are not equipped with Reed sensors. To be exact, these functions return 1 if an address is present in the configuration for the selected component and 0 otherwise.

```
void setsignal(struct railway_system *railway, int block, int signal, int state);
void setgatesignal(struct railway_system *railway, int signal, int state);
void settrack(struct railway_system *railway, int track, unsigned mode, unsigned target);
void setpoint(struct railway_system *railway, int point, int state);
void setlight(struct railway_system *railway, int light, int state);
void setgate(struct railway_system *railway, int gate, int state);
void setbell(struct railway_system *railway, int bell, int state);
```

These functions are used to set the components to new values. The possible desired states are defined as precompiler variables, and should be used in the functions. Possible values for signals or gatesignals are `RED`, `YELLOW`, `GREEN` or `OFF`. Points can be set to either `STRAIGHT` or `BRANCH`. The lights and the bell can be set to `ON` or `OFF` and the gates can be set to `UP` or `DOWN`.

```
unsigned getcontact(struct railway_system *railway, int block, int contact, int clear);
```

This function returns the last contact event of the selected Reed contact. This event can be one of `NONE`, `FWD`, `REV` or `UNI`, meaning no event, an event in forward direction, an event in reverse direction or an event with unknown direction. If the parameter `clear` is set to 0 this event is kept in the event queue otherwise it is removed from the buffer. If multiple events are present in the buffer, the direction is only stored for the first event. Subsequent events are signaled as `UNI`.

```
unsigned getsignal(struct railway_system *railway, int block, int signal);
unsigned getgatesignal(struct railway_system *railway, int signal);
unsigned getgatesensor(struct railway_system *railway, int gatesensor, int clear);
void gettrack(struct railway_system *railway, int track, unsigned *mode, unsigned *target);
int getpoint(struct railway_system *railway, int point);
int getlight(struct railway_system *railway, int light);
int getgate(struct railway_system *railway, int gate);
int getbell(struct railway_system *railway, int bell);
```

These functions are meant to get the current state of the components. Currently these functions are not implemented by the Railway API.

Chapter 6

Results

The achieved goals of this thesis are summarized in Section 6.1. Subsequently, Section 6.2 discusses some test that were run on the new model railway hardware and Section 6.3 shows options regarding the extendability of the new hardware. Section 6.4 presents some of the knowledge obtained from working on this thesis. Finally, Open Tasks are discussed in Section 6.5.

6.1 Goals Achieved

As listed in Section 1.1, the primary goal of this thesis was the replacement of the outdated controlling hardware of the model railway. This former hardware, consisting of 24 custom circuit boards and 24 PC/104 computers, has been replaced with a combination of 12 Arduino Mega 2560 microcontroller boards, some COTS circuit boards and 4 Raspberry Pis. The hardware replacement was achieved with only minor modifications to the wiring of the model railway and no changes to the model railway components itself. Prior to the implementation of the envisioned approach, the hardware was tested on a test circle that was built from scratch for this thesis. The choice of the hardware components has been documented and the hardware should provide a reliable basis for future laboratory courses.

The new hardware implements a layered approach to the controlling of the model railway similar to the third generation. While the Arduino boards manage the low-level access to the periphery, the Raspberry Pis provide communication with the end-user. The Arduinos and the Raspberry Pis communicate via a serial interface, while the Raspberry Pis use Ethernet communication with the end-user.

In comparison to the third generation hardware, the used hardware can easily be replaced in the event of a hardware defect.

The new hardware does not rely on external networking infrastructure, as in NFS or VPN connections. Only the existing network switch and the newly purchased router are required to operate the model railway. Even a fault of the router can be worked around with small adjustments to the connection handling of the Railway API.

The new hardware implements a broad subset of the previous generations API, providing good backwards compatibility. Most of the legacy controllers developed in the last courses should work out-of-the-box or with only minor modifications. Some features of the third generation were lost by the replacement of the hardware. The third generation hardware used sophisticated circuits in conjunction with the track drivers. This circuitry were able to detect the presence and the velocity of trains by measuring the resistance between the two rails of the tracks. Additionally, it provided extended control over the state of the tracks, like detecting short circuits caused by faulty train engines or broken switch points. These features are not possible to implement on the current hardware, because the H bridge modules cannot be configured to allow these kind of measurements.

The mapping from railway periphery parts to ports on the Arduino boards is not statically included in the API but configured through a web-based configuration tool. Additionally, the web interface on the Raspberry Pis provides a way to test each port of the Arduino boards individually and the option to program the Arduinos with new firmware.

6.2 Evaluation

The first tests for the hardware were performed on the testing circle. In these tests, the interaction of the model railway periphery, the Arduino boards and the hardware periphery was observed. The tests showed that the general approach is feasible and that0 the implementation on the real model railway should be possible. These tests did not cover the network communication but implemented the controller on the Raspberry Pi directly.

A second test was able to control two trains on the test circle simultaneously, using the controller design of the most recent laboratory course.

The hardware of the complete model railway was initially tested with the new testing interface, to ensure the correct function of each component. These test revealed two faults in the soldering of two of the interface boards. Additionally, the test revealed unsoldered pins on two of the Sainsmart Arduino boards.

Later, small controllers, testing single circles of the installation were executed successfully on the new hardware. These tests showed that the general approach is scaling to the bigger installation. During these tests some problems with unreliable Reed contact events were observed, especially with weaker magnets at the end of the trains. These problems probably arise from the analog input of the Arduino boards but should be able to be fixed by adjustments to the Arduino firmware.

Testing bigger controllers showed some problems regarding network latency. The quick, repeated tests for contact events created a notable delay on the processing of events.

6.3 Extendability

The track layout has not been altered since the second generation of the installation. The new hardware can easily be extended with additional Arduino boards. Each of the Raspberry Pis can support one additional Arduino board without the need for any additional hardware. If more Arduinos would be needed, a simple USB switch could provide the required USB ports for the boards.

If another circle of approximately the same size as the existing circles would be added, about two or three new Arduino boards would be needed to provide the needed component ports. Most of the Arduino boards still have port unused, making smaller adjustments to the track layout possible, without the need for additional hardware. The firmware for the Arduinos is kept very general and could, with minor adjustments, be used on other Arduino boards than the Arduino Mega 2560. This also provides the option to replace broken Arduino boards with newer versions without major adjustments to the firmware.

The hardware of the third generation was built with the focus on easy addition of new communication infrastructures. Apart from the addition of the TTP powernodes, this option has never been used again. For this reason, this option has been reduced in the new generation of the hardware. Instead of the 4 possible controlling systems, that the previous generation was capable of connecting to, the new hardware only provides two serial connections to connect coordination nodes to.

6.4 Lessons Learned

During the work on this thesis, some experiences were gathered:

Arduino Clones The Arduino boards used in this thesis are not manufactured by Arduino SLC, but by Sainsmart. Altough the board design is identical to the original boards, the overall quality of the boards seems to be lower. Several boards were not soldered completely, resulting in non-functioning signals or switch points. These faults could easily be fixed by soldering the faulty pin by hand. If genuine Arduino boards were used, these faulty boards would be replaced by the manufacturer.

Network Latency The network communication creates a substantial delay. The initial implementation of the Railway API used individual blocking network calls to gather contact data from the Raspberry Pis. These blocking calls were not critical while running small tests but became an issue when testing bigger controllers. The third generation of the model railway circumvented this problem by keeping a local copy of the complete state which was synchronized with the real state in regular intervals.

Serial Connection on Raspberry Pi The first tests of the serial communication with the Arduino boards were not run on the Raspberry Pi, but on a standard laptop running Ubuntu Linux. Running the same tests on the Raspberry Pi, after a certain time, a significant amount of transmission errors could be detected on the serial connection. These messages have to be retransmitted to ensure a correct behavior of the controller.

General Approach The general approach of the controller design seems to be feasible and easy to adapt to the concrete implementation. The network communication requires significant work to provide a reliable infrastructure, even when using TCP for the communication.

6.5 Future Work

The network protocol of the approach should be improved in several regards. The messaging layer could implement a safer way of determining the end of a message, preventing problems between recognizing data bytes and stop bytes.

Another desirable addition is the implementation of reading access to railway components. These functions are already prepared in the Railway API but were not implemented, due to the tight schedule being disturbed by significant delays in the supply of certain hardware parts. The implementation of this reading access requires additional messages to be added to the network communication layer.

In the previous generation, the power electronics stored error counters for certain events in its EEPROM. These counters were used for counting faulty contact events or short circuits on the tracks. At least the error counters for the faulty contact events could be implemented on the new hardware.

The current implementation does not delegate control logic to the Raspberry Pis. An improvement could be the delegation of address resolution to the Raspberry Pis. This would significantly reduce the size of the messages sent by the end-user. Additionally, it would provide a simple networking API which could be used by to implement a distributed controller on the Raspberry Pis.

The new network infrastructure exposes the model railway to the whole university. This could potentially result in unwanted behavior if unauthorized persons connect to the model railway. Most of the time, this problem cannot turn up due to the fact that the model railway is usually powered down. Nevertheless, a form of authentication would be a useful extension of the current network communication. Additionally, the communication could be encrypted during transport, possibly by integrating a common library like OpenSSL to the Railway API.

Appendix A

Components Mapping

The following tables document at which port each component is connected to the controlling hardware. These informations are needed by the controller to determine which coordination node needs to be used to set a specific component to a desired state.

RailPi0 - Arduino 0 (Serial Number 75437303830351714161)

Port	Component	Port	Component
SIG0	(free)	PNT0	PNT_12
SIG1	(free)	PNT1	L_8
SIG2	(free)	PNT2	PNT_8
SIG3	(free)	PNT3	PNT_9
SIG4	KH_LN_0, Signal 0	PNT4	L_11
SIG5	KIO_LN_1, Signal 0	PNT5	(free)
TRK0	(free)	CNT0	(free)
TRK1	(free)	CNT1	OI_LN_0, Contact 1
TRK2	IC_LN_4	CNT2	KH_LN_0, Contact 0
TRK3	OI_LN_2	CNT3	(free)
TRK4	OC_LN_1	CNT4	KIO_LN_1, Contact 0
TRK5	KH_ST_6	CNT5	(free)
		CNT5	IC_LN_4, Contact 1
		CNT5	(free)

RailPi0 - Arduino 1 (Serial Number 75437303830351712071)

Port	Component	Port	Component
SIG0	KH_ST_1, Signal 1	PNT0	Point 7
SIG1	KH_ST_2, Signal 1	PNT1	Point 6
SIG2	KH_ST_3, Signal 1	PNT2	Point 5
SIG3	OI_LN_2, Signal 1	PNT3	Light 10
SIG4	IC_LN_0, Signal 1	PNT4	Light 9
SIG5	OC_LN_4, Signal 1	PNT5	(free)
TRK0	(free)	CNT0	OC_LN_5, Contact 0
TRK1	(free)	CNT1	OC_LN_4, Contact 1
TRK2	KH_LN_6	CNT2	OI_LN_0, Contact 0
TRK3	IC_LN_1	CNT3	KH_ST_2, Contact 1
TRK4	OI_LN_0	CNT4	IC_LN_0, Contact 1
TRK5	OC_LN_4	CNT5	IC_LN_1, Contact 0
		CNT6	OC_LN_1, Contact 0
		CNT7	KH_ST_1, Contact 1

RailPi0 - Arduino 2 (Serial Number 7543730383035181F052)

Port	Component	Port	Component
SIG0	KH_ST_5, Signal 1	PNT0	Light 7
SIG1	KH_ST_4, Signal 1	PNT1	Point 13
SIG2	(free)	PNT2	Light 6
SIG3	IC_LN_4, Signal 1	PNT3	Point 10
SIG4	KH_LN_6, Signal 1	PNT4	Point 14
SIG5	KH_LN_5, Signal 1	PNT5	Point 11
TRK0	(free)	CNT0	KH_ST_3, Contact 1
TRK1	(free)	CNT1	KH_ST_4, Contact 1
TRK2	KH_ST_5	CNT2	(free)
TRK3	IC_JCT_0	CNT3	KH_ST_5, Contact 1
TRK4	KH_LN_5	CNT4	(free)
TRK5	OC_JCT_0	CNT5	KH_LN_5, Contact 1
		CNT6	OI_LN_2, Contact 1
		CNT7	KH_LN_6, Contact 1

RailPi1 - Arduino 3 (Serial Number 754373038303517170F0)

Port	Component	Port	Component
SIG0	IO_LN_2, Signal 1	PNT0	Light 4
SIG1	KH_ST_2, Signal 0	PNT1	Light 5
SIG2	OC_LN_0, Signal 1	PNT2	Light 2
SIG3	KH_LN_7, Signal 0	PNT3	Light 3
SIG4	KH_ST_1, Signal 0	PNT4	(free)
SIG5	KH_LN_1, Signal 1	PNT5	(free)
TRK0	(free)	CNT0	IO_LN_0, Contact 0
TRK1	(free)	CNT1	KH_ST_1, Contact 0
TRK2	KH_ST_1	CNT2	KH_LN_7, Contact 0
TRK3	KH_ST_2	CNT3	IC_LN_5, Contact 0
TRK4	IC_LN_5	CNT4	OC_LN_0, Contact 1
TRK5	KH_ST_3	CNT5	KH_ST_2, Contact 0
		CNT6	KH_LN_1, Contact 1
		CNT7	IO_LN_2, Contact 1

RailPi1 - Arduino 4 (Serial Number 75437303830351816142)

Port	Component	Port	Component
SIG0	(free)	PNT0	Point 4
SIG1	IC_LN_1, Signal 1	PNT1	Point 3
SIG2	KH_ST_5, Signal 0	PNT2	Light 1
SIG3	KH_ST_4, Signal 0	PNT3	(free)
SIG4	KH_ST_3, Signal 0	PNT4	Point 1
SIG5	KH_LN_2, Signal 0	PNT5	Point 2
TRK0	(free)	CNT0	KH_ST_3, Contact 0
TRK1	(free)	CNT1	(free)
TRK2	IO_LN_2	CNT2	KH_LN_2, Contact 0
TRK3	KH_ST_4	CNT3	(free)
TRK4	KH_LN_7	CNT4	KH_ST_5, Contact 0
TRK5	KH_LN_1	CNT5	OC_LN_4, Contact 0
		CNT6	KH_ST_4, Contact 0
		CNT7	IC_LN_1, Contact 1

RailPi1 - Arduino 5 (Serial Number 75437303830351719191)

Port	Component	Port	Component
SIG0	IO_LN_1, Signal 1	PNT0	Point 0
SIG1	OC_LN_3, Signal 1	PNT1	Light 0
SIG2	KIO_LN_0, Signal 1	PNT2	(free)
SIG3	KH_LN_8, Signal 1	PNT3	(free)
SIG4	(free)	PNT4	(free)
SIG5	(free)	PNT5	(free)
TRK0	(free)	CNT0	(free)
TRK1	(free)	CNT1	(free)
TRK2	KIO_LN_0	CNT2	KIO_LN_0, Contact 1
TRK3	KH_LN_8	CNT3	(free)
TRK4	KH_ST_0	CNT4	OC_LN_3, Contact 1
TRK5	IO_LN_0	CNT5	IO_LN_2, Contact 0
		CNT6	KH_LN_8, Contact 1
		CNT7	IC_LN_2, Contact 0

RailPi2 - Arduino 6 (Serial Number 754373038303517142A1)

Port	Component	Port	Component
SIG0	(free)	PNT0	Gate_0, Boom
SIG1	KIO_LN_0, Signal 0	PNT1	Gate_1, Boom
SIG2	IC_LN_5, Signal 1	PNT2	Light 22
SIG3	Gate_1, St Andrews Cross	PNT3	Light 23
SIG4	IO_LN_0, Signal 1	PNT4	Bell 0
SIG5	Gate_0, St Andrews Cross	PNT5	Point 16
TRK0	(free)	CNT0	IO_LN_1, Contact 1
TRK1	(free)	CNT1	Gate_1, Boom Detector
TRK2	IC_ST_0	CNT2	KIO_LN_0, Contact 0
TRK3	OC_ST_4	CNT3	Gate_0, Boom Detector
TRK4	IO_LN_1	CNT4	IC_LN_5, Contact 1
TRK5	OC_LN_0	CNT5	IO_LN_1, Contact 0
		CNT6	OC_LN_0, Contact 0
		CNT7	IO_LN_0, Contact 1

RailPi2 - Arduino 7 (Serial Number 75437303830351712111)

Port	Component	Port	Component
SIG0	(free)	PNT0	Point 22
SIG1	(free)	PNT1	Point 20
SIG2	(free)	PNT2	Point 19
SIG3	KH_LN_8, Signal 0	PNT3	Point 21
SIG4	OC_ST_3, Signal 1	PNT4	Point 17
SIG5	KH_LN_7, Signal 1	PNT5	Point 18
TRK0	(free)	CNT0	(free)
TRK1	(free)	CNT1	IC_ST_2, Contact 0
TRK2	IC_ST_3	CNT2	IC_ST_1, Contact 0
TRK3	OC_LN_2	CNT3	IC_LN_3, Contact 0
TRK4	IC_LN_2	CNT4	OC_LN_3, Contact 0
TRK5	OC_LN_3	CNT5	IC_LN_2, Contact 1
		CNT6	KH_LN_8, Contact 0
		CNT7	KH_LN_7, Contact 1

RailPi2 - Arduino 8 (Serial Number 7543730383035171A021)

Port	Component	Port	Component
SIG0	OC_ST_2, Signal 1	PNT0	Light 21
SIG1	(free)	PNT1	Light 20
SIG2	OC_ST_1, Signal 1	PNT2	Light 18
SIG3	OC_LN_2, Signal 1	PNT3	Light 17
SIG4	KH_LN_2, Signal 1	PNT4	Light 19
SIG5	IC_LN_2, Signal 1	PNT5	Point 15
TRK0	(free)	CNT0	(free)
TRK1	(free)	CNT1	(free)
TRK2	IC_ST_2	CNT2	OC_ST_1, Contact 1
TRK3	IC_ST_1	CNT3	OC_ST_2, Contact 1
TRK4	IC_LN_3	CNT4	IC_ST_3, Contact 0
TRK5	KH_LN_2	CNT5	KH_LN_2, Contact 1
		CNT6	OC_ST_3, Contact 1
		CNT7	OC_LN_2, Contact 1

RailPi3 - Arduino 9 (Serial Number 75437303830351811292)

Port	Component	Port	Component
SIG0	OC_LN_1, Signal 1	PNT0	Light 15
SIG1	KH_LN_3, Signal 0	PNT1	Light 16
SIG2	KH_LN_1, Signal 0	PNT2	Light 14
SIG3	IC_LN_3, Signal 1	PNT3	(free)
SIG4	(free)	PNT4	(free)
SIG5	KH_LN_4, Signal 0	PNT5	(free)
TRK0	(free)	CNT0	IC_ST_3, Contact 1
TRK1	(free)	CNT1	OI_LN_1, Contact 1
TRK2	KH_LN_4	CNT2	IC_ST_2, Contact 1
TRK3	KH_LN_3	CNT3	IC_ST_1, Contact 1
TRK4	OC_ST_3	CNT4	KH_LN_6, Contact 0
TRK5	KH_LN_0	CNT5	KH_LN_3, Contact 0
		CNT6	KH_LN_1, Contact 0
		CNT7	KH_LN_4, Contact 0

RailPi3 - Arduino 10 (Serial Number 754373038303517171A1)

Port	Component	Port	Component
SIG0	IC_ST_2, Signal 1	PNT0	Point 26
SIG1	KH_LN_6, Signal 0	PNT1	Light 13
SIG2	OI_LN_1, Signal 1	PNT2	Light 12
SIG3	IC_ST_3, Signal 1	PNT3	Point 25
SIG4	KH_LN_0, Signal 1	PNT4	Point 23
SIG5	IC_ST_1, Signal 1	PNT5	Point 24
TRK0	(free)	CNT0	OC_ST_2, Contact 0
TRK1	(free)	CNT1	OI_LN_2, Contact 0
TRK2	OC_ST_1	CNT2	KH_LN_4, Contact 1
TRK3	OC_ST_2	CNT3	OC_ST_3, Contact 0
TRK4	IC_LN_0	CNT4	OC_ST_1, Contact 0
TRK5	OI_LN_1	CNT5	IC_LN_3, Contact 1
		CNT6	KH_LN_0, Contact 1
		CNT7	OC_LN_2, Contact 0

RailPi3 - Arduino 11 (Serial Number 7543730383035171B1D0)

Port	Component	Port	Component
SIG0	KH_LN_4, Signal 1	PNT0	Point 29
SIG1	OC_LN_5, Signal 1	PNT1	Point 27
SIG2	KIO_LN_1, Signal 1	PNT2	Point 28
SIG3	KH_LN_3, Signal 1	PNT3	(free)
SIG4	OI_LN_1, Signal 1	PNT4	(free)
SIG5	KH_LN_5, Signal 0	PNT5	(free)
TRK0	(free)	CNT0	IC_LN_4, Contact 0
TRK1	(free)	CNT1	OC_LN_1, Contact 1
TRK2	IC_ST_4	CNT2	KH_LN_3, Contact 1
TRK3	OC_ST_0	CNT3	KIO_LN_1, Contact 1
TRK4	OC_LN_5	CNT4	IC_LN_0, Contact 0
TRK5	KIO_LN_1	CNT5	OC_LN_5, Contact 1
		CNT6	OI_LN_1, Contact 0
		CNT7	KH_LN_5, Contact 0

Free ports

Table A.1 lists the free ports on the Arduino controller hardware boards.

Table A.1. Free ports of the Arduino controller hardware

	0	1	2	3	4	5	6	7	8	9	10	11
SIG	4	0	1	0	1	2	1	3	1	1	0	0
TRK	2	2	2	2	2	2	2	2	2	2	2	2
PNT	1	1	0	2	1	4	0	0	0	3	0	3
CNT	4	0	2	0	2	3	0	1	2	0	0	0

Appendix B

Raspberry Pi Installation

The Raspberry Pis use a standard Raspbian image as a starting point. From this point only minor adjustments need to be implemented to transform a RasPi into a RailPi.

Basic installation

The RailPi use the default Raspbian Image, which is copied to the microSD card of the RailPi. Currently the RailPis use the image version 2014-09-09 but no special dependencies to a specific version exist, so the most recent image should be suitable.

The RailPi is started and the basic configuration is performed using the tool `raspi-config`. In this tool, the partition on the SD card is enlarged and the hostname is set to railPiX, where X is the number of the controlling node.

After `raspi-config` finished its work and the RasPi has restarted, the network configuration is adjusted. The RailPis at the model railway are statically set to an address between `10.6.6.10` and `10.6.6.13`.

Enable IPv6

The Railway Daemon already uses IPv6 sockets for the connection to the controlling computer. IPv6 is available on Raspbian, but not enabled by default. To enable IPv6 the corresponding kernel module needs to be loaded. This can be done at system start by adding a line with the entry `ipv6` to the file `/etc/modules`. After a reboot the module should be loaded and IPv6 should be working.

Install nginx

The testing and configuration interface for the railway is provided by the webserver *nginx* and the PHP interpreter. These packages are installed from the packet repositories.

```
> sudo apt-get install nginx php5-fpm
```

After the packages have been installed, the configuration for the webserver is created. Paste the following content to the file `/etc/nginx/sites-available/railway`.

```
server {
       root /srv/www;
       index index.php index.htm index.html;
       server_name localhost;
```

```

        location / {
                try_files $uri $uri/ /index.html;
        }

        location ~ \.php$ {
                fastcgi_split_path_info ^(.+\.php)(/.+)$;

                fastcgi_pass unix:/var/run/php5-fpm.sock;
                fastcgi_index index.php;
                include fastcgi_params;
        }
}
```

The configuration is activated by setting a symlink in `/etc/nginx/sites-enabled/`.

```
> sudo rm /etc/nginx/sites-enabled/default
> sudo ln -s /etc/nginx/sites-available/railway /etc/nginx/sites-enabled/railway
```

Install Railway Components

The Railway Software is installed from the Stash. Clone the repository to a folder like `/opt/railway` and give write access to the user *www-data*. Then, create a symlink from the folder `/srv/www` to the subfolder `Code/www` in your local working copy of the repository.

The testing programs of the testing interface need to be compiled on the RailPi. This is done by running `make` in the subdirectory `Code/www/bin`.

The Railway Daemon is compiled by running `make` in the subdirectory `Code/railwayd`.

All done

This should provide you with a working RailPi that is able to word as a coordination node on the model railway.

Track Diagrams

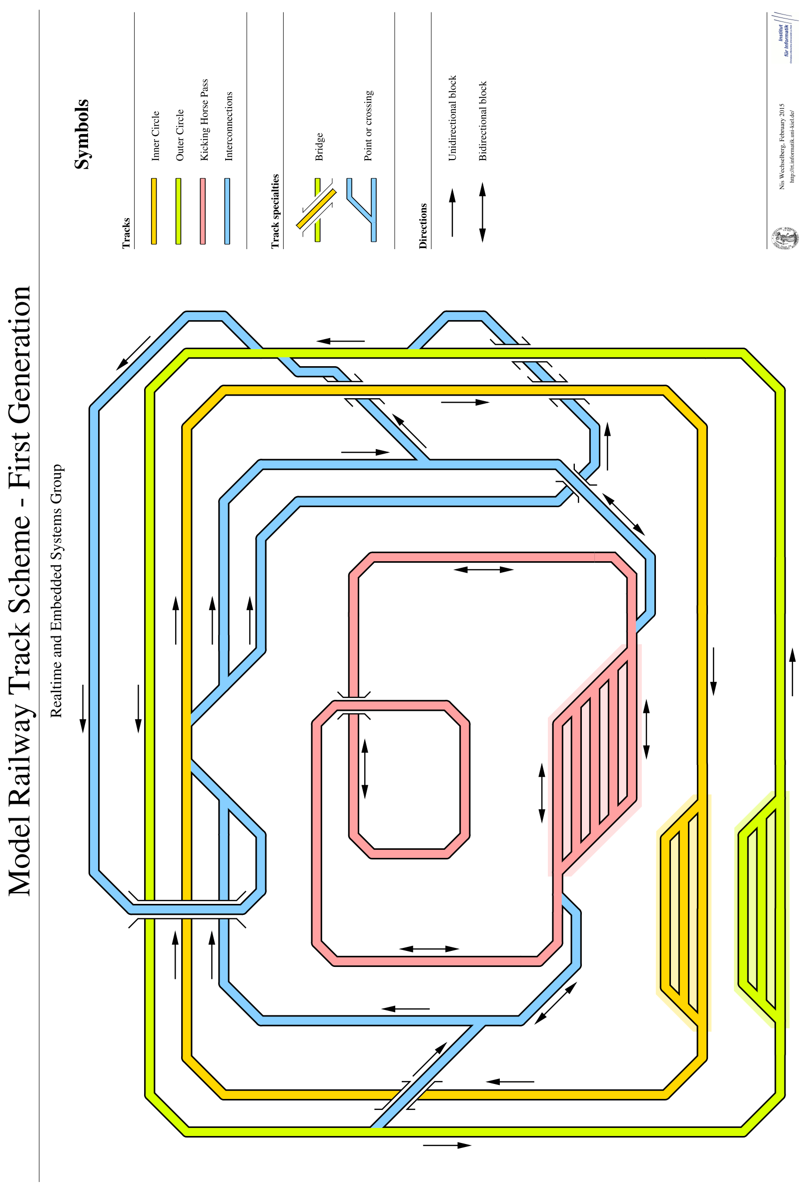


Figure C.1. Schematic of the first generation track layout

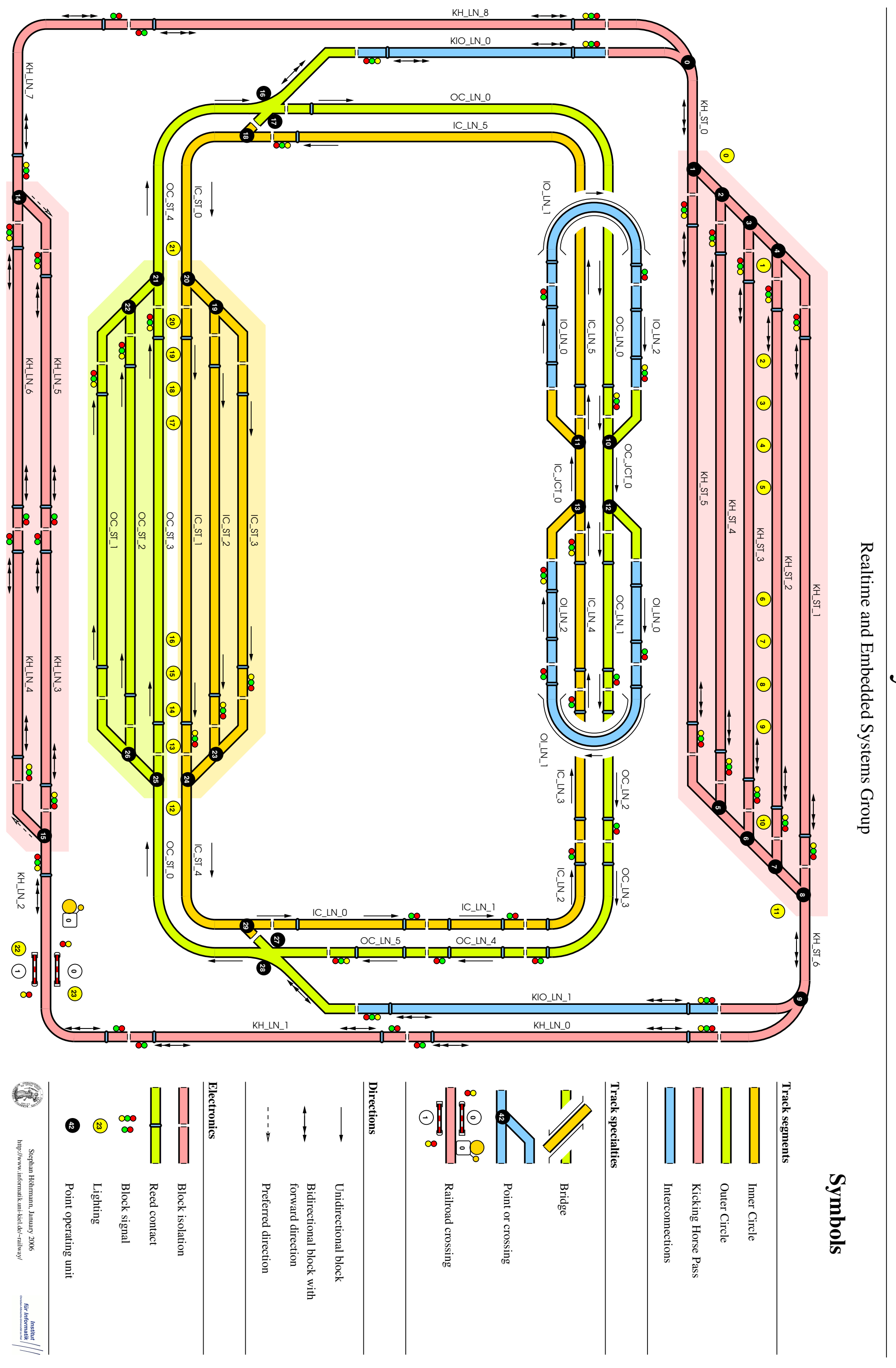


Figure C.2. Schematic of the current track layout

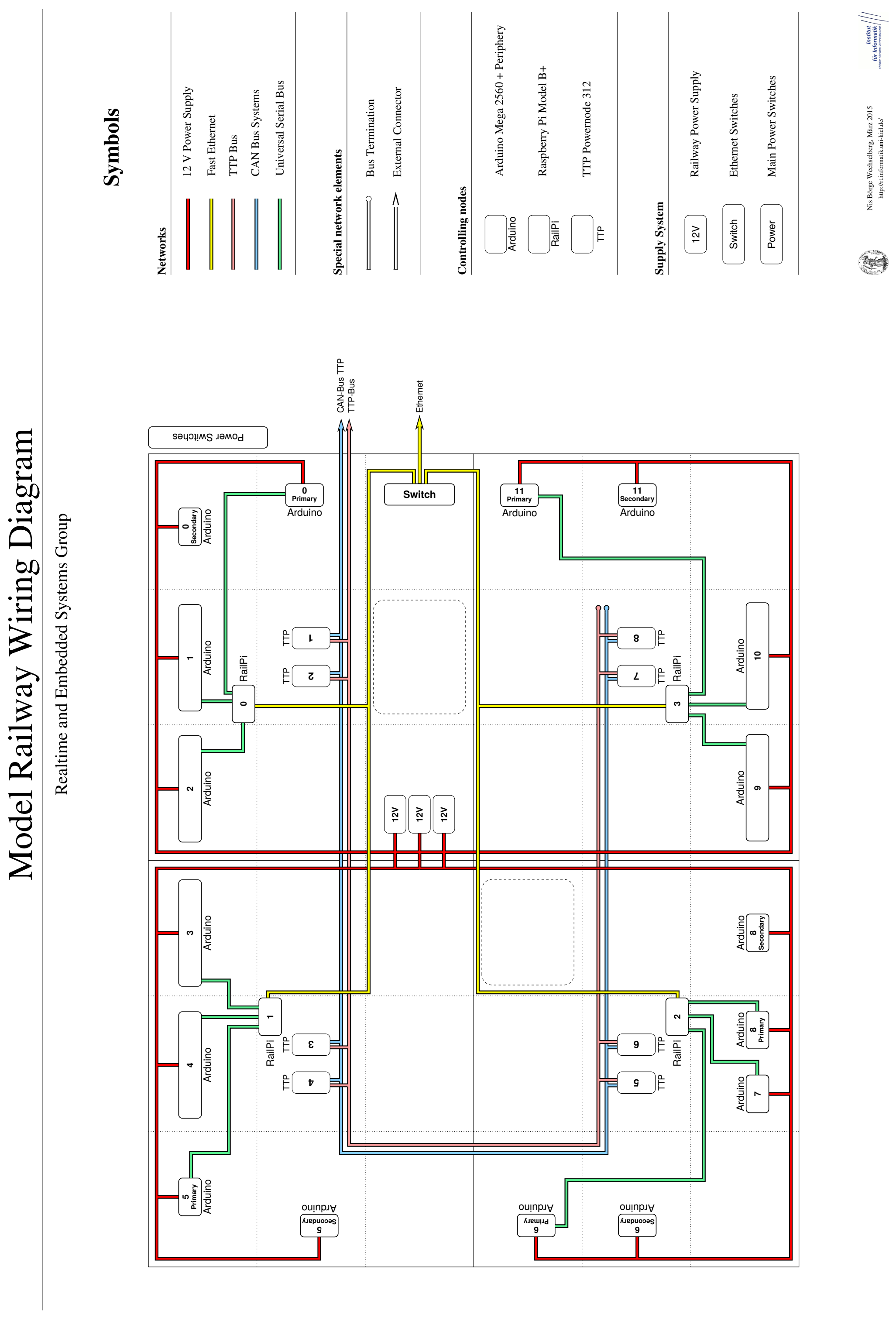


Figure C.3. Wiring diagramm of the model railway

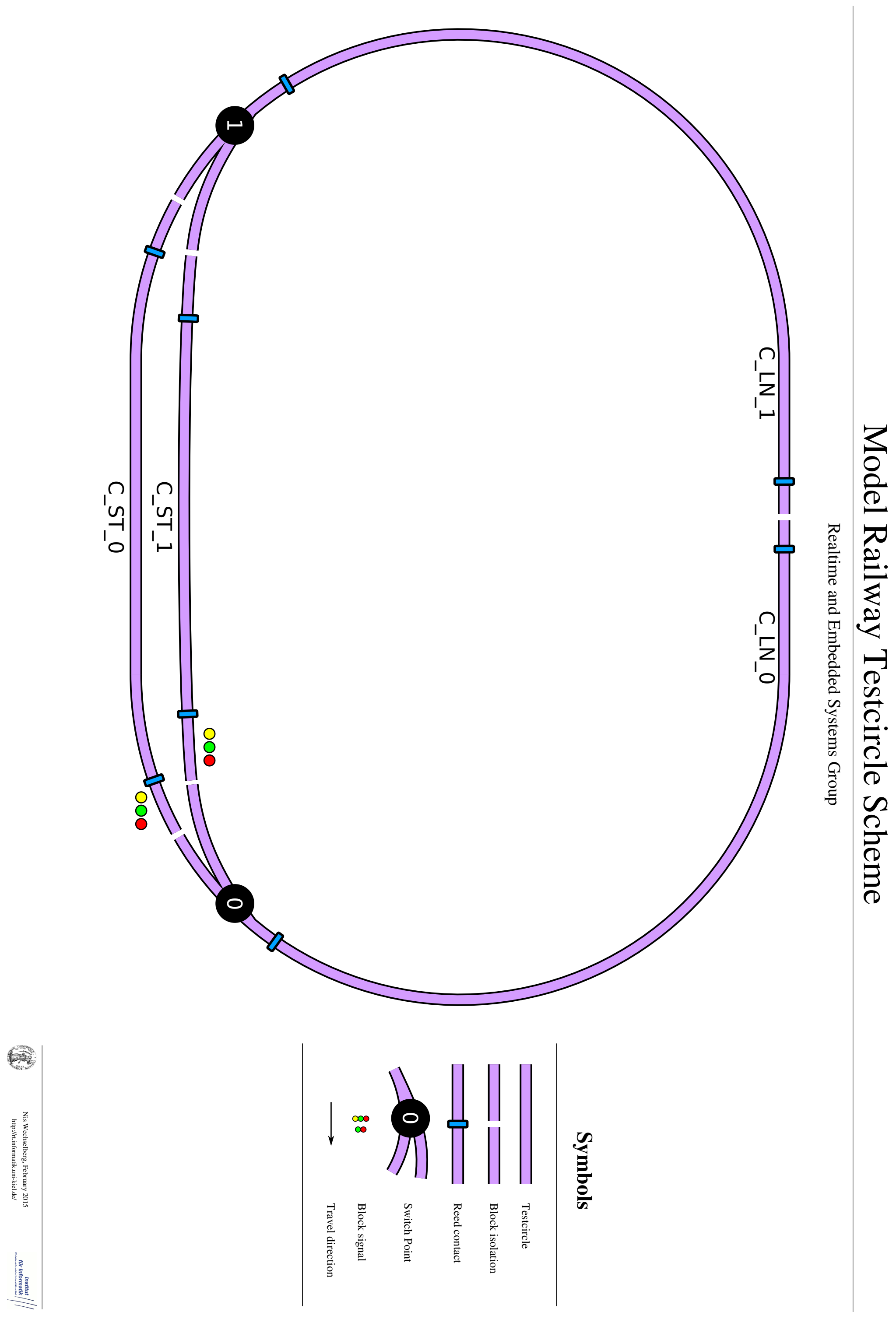

Model Railway Testcircle Scheme
Realtime and Embedded Systems Group
C_LN_1
C_LN_0
C_ST_1
C_ST_0
1
0
Symbols
Testcircle
Block isolation
Reed contact
Switch Point
Block signal
Travel direction
Nis Wechselberg, February 2015
http://rt.informatik.uni-kiel.de/
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Figure C.4. Schematic of the testcircle

Figure Attributions

Figure 1.1 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.1 *The Spiral Tunnels - Yoho National Park* by *Conrad Olson*, `https://flic.kr/p/cCFMcm`, Licensed under *CC-BY-NC-SA 2.0*, `https://creativecommons.org/licenses/by-nc-sa/2.0/` / Cropped from original.

Figure 2.2 Schematic by Nis Börge Wechselberg, Kiel, 2015 / Adapted from original schematic by Werner E. Kluge in [Klu98, p. 2].

Figure 2.3 Photo by Clemens Grelck, Kiel, 1998.

Figure 2.4a Photo by Jochen Koberstein, Kiel, 2001. Published in [Kob01, p. 57].

Figure 2.4b Photo by Stephan Höhrmann, Kiel, 2004.

Figure 2.5a Photo by Stephan Höhrmann, Kiel, 2005. Published in [Höh06, p. 19].

Figure 2.5b Photo by Stephan Höhrmann, Kiel, 2005.

Figure 2.6a Schematic by Christian Motika, Kiel, 2014. Published in [Mot14a].

Figure 2.6b Photo by Christian Motika, Kiel, 2014. Published in [Mot14a].

Figure 2.7 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.8 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.9 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.10 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.11 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.12a Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 2.12b Schematic by Jochen Koberstein, Kiel, 2001. Published in [Kob01, p. 24].

Figure 2.13a Photo by Stephan Höhrmann, Kiel, 2005. Published in [Höh06, p. 32].

Figure 2.13b Photo by Stephan Höhrmann, Kiel, 2005. Published in [Höh06, p. 33].

Figure 2.14a Schematic by Stephan Höhrmann, Kiel, 2006. Published in [Höh06].

Figure 2.14b Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 2.15 Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.1a Schematic by Stephan Höhrmann, Kiel, 2006. Published in [Höh06, p. 62].

Figure 3.1b Schematic by Nis Börge Wechselberg, Kiel, 2015 / Adapted from original schematic in Figure 3.1a.

Figure 3.2a Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 3.2b *Beaglebone Black - Top* by *Gareth Halfacree*, `https://flic.kr/p/o5xarv`, Licensed under *CC-BY-SA 2.0*, `https://creativecommons.org/licenses/by-sa/2.0/`.

Figure 3.2c Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 3.2d *Cubieboard 2 - Top* by *Gareth Halfacree*, `https://flic.kr/p/omCuYN`, Licensed under *CC-BY-SA 2.0*, `https://creativecommons.org/licenses/by-sa/2.0/`.

Figure 3.3a Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.3b Image by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.4a *ATtiny2313* by *Windell Oskay*, `https://flic.kr/p/tTGEo`, Licensed under *CC-BY 2.0*, `https://creativecommons.org/licenses/by/2.0/` / Rotated and smoothed edges

Figure 3.4b *Unmarked SOIC chip* by *Nis Börge Wechselberg*, Licensed under *CC-BY-SA 3.0*, `https://creativecommons.org/licenses/by-sa/3.0/` / Based on *NE555 DIP & SOIC* by *Swift.Hg*, `http://commons.wikimedia.org/w/index.php?title=File:NE555_DIP_&_SOIC.jpg&oldid=127310901`, Licensed under *CC-BY-SA 3.0*, `https://creativecommons.org/licenses/by-sa/3.0/`

Figure 3.4c Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.4d Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.5a *Arduino Micro* by *Arduino SA*, `http://arduino.cc/en/Main/ArduinoBoardMicro`, Licensed under *CC-BY-SA 3.0*, `https://creativecommons.org/licenses/by-sa/3.0/`

Figure 3.5b *Arduino Uno* by *Arduino SA*, `http://arduino.cc/en/Main/ArduinoBoardUno`, Licensed under *CC-BY-SA 3.0*, `https://creativecommons.org/licenses/by-sa/3.0/`.

Figure 3.5c *Arduino Mega 2560* by *Arduino SA*, `http://arduino.cc/en/Main/ArduinoBoardMega2560`, Licensed under *CC-BY-SA 3.0*, `https://creativecommons.org/licenses/by-sa/3.0/`.

Figure 3.6 Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.7a Photo by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.7b Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.8a *Vierquadrantensteller* by *Biezl*, `http://commons.wikimedia.org/w/index.php?title=File:Vierquadrantensteller.svg&oldid=151000083`, Licensed under *Public Domain*.

Figure 3.8b Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 3.9a Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.9b Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.10a Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 3.10b Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 3.11 Photo by Christoph Daniel Schulze, Kiel, 2015.

Figure 5.1 Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure 5.2 Image by Nis Börge Wechselberg, Kiel, 2015.

Figure 5.3 Image by Nis Börge Wechselberg, Kiel, 2015.

Figure C.1 Schematic by Nis Börge Wechselberg, Kiel, 2015 / Adapted from original schematic by Wolfgang Hielscher in [HUR+98].

Figure C.2 Schematic by Stephan Höhrmann, Kiel, 2006. Published in [Höh06].

Figure C.3 Schematic by Nis Börge Wechselberg, Kiel, 2015.

Figure C.4 Schematic by Nis Börge Wechselberg, Kiel, 2015.

Bibliography

[All13] Alasdair Allan. “Which board is right for me?” In: *MAKE Magazine* 36 (Nov. 2013). URL: `http://makezine.com/magazine/make-36-boards/which-board-is-right-for-me/`.

[Ard15] Arduino SA. Arduino Policy. 2015. URL: `http://arduino.cc/en/Main/Policy`.

[Ber99] Gérard Berry. The Esterel v5 Language Primer. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps`. 1999.

[CPH+87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. “Lustre: a declarative language for programming synchronous systems”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’87)*. Munich, Germany: ACM, 1987, pp. 178–188. ISBN: 0-89791-215-2. DOI: `10.1145/41625.41641`. URL: `http://doi.acm.org/10.1145/41625.41641`.

[ELL+97] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. “Design of embedded systems: Formal models, validation, and synthesis”. In: *Proceedings of the IEEE* 85.3 (Mar. 1997), pp. 366–390.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for safety-critical applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’14)*. Edinburgh, UK: ACM, June 2014.

[HFP+06] Stephan Höhrmann, Hauke Fuhrmann, Steffen Prochnow, and Reinhard von Hanxleden. “A versatile demonstrator for distributed real-time systems: using a model-railway in education”. In: *Proceedings of the ERCIM/DECOS Dependable Embedded Systems Workshop at Euromicro 2006*. Ed. by Amund Skavhaug and Erwin Schoitsch. Cavtat/Dubrovnik, Croatia, Aug. 2006.

[Höh06] Stephan Höhrmann. “‘Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage’”. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sho-dt.pdf`. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mar. 2006.

[HUR+98] Wolfgang Hielscher, Lars Urbszat, Claus Reinke, and Werner Kluge. “On modelling train traffic in a model train system”. In: *Daimi PB-532: Workshop on Practical Use of Coloured Petri Nets and Design/CPN*. Ed. by K. Jensen. Department of Computer Science, University of Aarhus, Denmark. 1998, pp. 83–102.

[Klu98] Werner E. Kluge. “The Kicking Horse Pass problem”. In: *Petri Net News Letters No. 54* (1998), pp. 3–15.

[Kob01] Jochen Koberstein. “‘Realisierung eines geordneten Mehrzugbetriebs auf einer Modellbahnanlage’”. Diploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2001.

[LS11] Edward A. Lee and Sanjit A. Seshia. Introduction to Embedded Systems, A Cyber-Physical Systems Approach. Lulu, 2011. ISBN: 978-0-557-70857-4. URL: `http://LeeSeshia.org`.

[Men13] Jirka Niklas Menke. “Auf dem richtigen Gleis”. In: *Unizeit* 78 (Oct. 2013), p. 12.

[Mot14a] Christian Motika. Model Railway 2.0—An Exploration of Alternatives for Interface Hardware. `http://www.informatik.uni-kiel.de/fileadmin/arbeitsgruppen/realtime_embedded/railway/railway2.0-cmot-handout.pdf`. 2014.

[Mot14b] Christian Motika. SCCharts in Motion—Interactive Model-Based Compilation for a Railway System. Presentation at the 21th International Open Workshop on Synchronous Programming (SYNCHRON’14), Aussois, France. Dec. 2014.

[Pol95] G. Pole. Spiral Tunnels and the Big Hill: A Canadian Railway Adventure. Altitude Pub., 1995. ISBN: 9781551530871.

[SR13] W. Richard Stevens and Stephen A. Rago. Advanced programming in the UNIX environment. Third edition. Pearson Education, 2013.

[Tex14] Texas Instruments Incorporated. Lift-off with the LaunchPad Ecosystem. 2014. URL: `http://www.ti.com/lit/sg/slat152a/slat152a.pdf`.

[The03] The model railway. 2. Generation—Project homepage. Jürgen Noss, Department of Computer Science, Kiel, Germany. 2003. URL: `http://www.informatik.uni-kiel.de/inf/Kluge/trains/`.

[The06] The model railway. 3. Generation—Project homepage. Group of Real-Time and Embedded Systems, Department of Computer Science, Kiel, Germany. 2006. URL: `http://www.informatik.uni-kiel.de/~railway`.

[The99] The model railway. 1. Generation—Project homepage. Clemens Grelk, Department of Computer Science, Kiel, Germany. 1999. URL: `https://staff.fnwi.uva.nl/c.u.grelck/railway/html/index.html`.