FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Sicherheitskritische Modelleisenbahnsteuerung in Ada

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Mark Volcic
Matrikelnummer 0425294

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Johann Blieberger

Wien, 05.09.2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Sicherheitskritische Modelleisenbahnsteuerung in Ada

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Technical Informatics

by

Mark Volcic
Registration Number 0425294

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dr. Johann Blieberger

Vienna, 05.09.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Mark Volcic
Dreyhausenstraße 10/8, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Ich möchte mich hiermit bei dem Betreuer dieser Arbeit, Prof. Johann Blieberger, recht herzlich bedanken, der es mir ermöglicht hat, an einer praktischen Diplomarbeit zum Abschluss meines Studiums zu arbeiten. Weiters bedanke ich mich für die Geduld und Unterstützung während der gesamten Dauer.
Ein weiterer großer Dank geht an Fritz Praus, der mir vor allem im Bereich der Hardware einige Male mit Rat und Tat zur Seite gestanden ist.

Der größte Dank geht an meine Familie, die mich sowohl in meiner Ausbildung als auch im Privatleben immer tatkräftig unterstützt und mir diesen Ausbildungsweg ermöglicht hat.

Abstract

Goal

The goal of this master thesis was to develop a safe and reliable system to control some railroads in a closed system. For the development, simulation and testing, a model railroad system was used, which has the same requirements regarding safety and reliability. The hardware and software were partially available but they have to be adapted and further developed. The software for the microcontroller was written in C but the target software should be in Ada. The available parts of a controlling software for the PC were discarded and completely new developed. The design of the hardware had to be changed too, because there were some new requirements including a new processor.

Short Description

It should be possible to use more than one train simultaneously in a big and complex system. A PC is used for the controlling and for the visualisation. Nodes with microcontrollers are used to operate with the trains. The whole system is devided into parts, where each part is controlled by one node. The node has to ensure that there could only be one train within its segment of the system. To exchange data within the railroad system and with the PC a ring topology with a RS232 interface is used. This topology has the advantage that any defect node could be found quickly, which is needed to ensure that there could not be any dangerous situations (e.g a crash). Each node uses an Atmel microcontroller, an interface for the rail and the switch, two interfaces for communication (RX and TX), some LEDs to show the current situation and the supply voltage.
The PC is used for visualisation of the parameters like the speed or the direction and to control the trains. It also communicates with the nodes and sends the parameter information to them.

Kurzfassung

Ziel

Das Ziel der Diplomarbeit ist die Entwicklung einer sicheren und zuverlässigen Steuerung von Zügen in einem abgeschlossenen Gleissystem. Für die Implementierung und die Testphase wurde eine Modelleisenbahn gewählt, die den sicherheitskritischen Ansprüchen von realen Zügen und Gleisen entsprechen muss. Hard- und Software waren teilweise schon vorhanden, allerdings wurden diese weiterentwickelt bzw. verändert. Die vorhandene Software der Mikrocontroller wurde in C geschrieben, die Lösung soll ausschließlich in Ada entwickelt werden. Bei der Hardware musste das Design geändert und ein neuer Prozessor gewählt werden, der den Anforderungen entspricht. Für die Steuerungssoftware am PC gab es erste Testversionen, die den Anforderungen nicht mehr entsprachen und komplett neu entwickelt werden mussten.

Kurzbeschreibung

Die Steuerung der Modelleisenbahn soll es ermöglichen, mehrere Züge gleichzeitig auf einem komplexen Gleissystem zu betreiben. Dafür werden auf der einen Seite ein Steuer-PC, auf der anderen Seite Mikrocontroller verwendet. Das Gleissystem wird in einzelne Abschnitte unterteilt, denen jeweils ein Controller zugeordnet ist. Dieser übernimmt die Steuerung des Zuges, der sich aktuell am Gleisabschnitt befindet. Ist dies der Fall, muss sichergestellt werden, dass keine weiteren Züge in diesen Abschnitt einfahren dürfen. Dies wird dadurch ermöglicht, dass die Controller miteinander und mit dem Master-Rechner kommunizieren. Hierfür ist ein Datenring über RS232-Schnittstellen (Punkt-zu-Punkt-Kommunikation) vorgesehen. Durch die Kommunikationstopologie kann auch sehr einfach sichergestellt werden, ob ein Controller ausgefallen ist und eventuell der Betrieb eingestellt oder umgeleitet werden muss. Ausgefallene Züge (z.B. durch einen Defekt) müssen erkannt werden, damit es zu keinen Kollisionen kommen kann.
Jeder Mikrocontroller besteht aus einem Atmel-Prozessor, einem Anschluss für die Versorgungsspannung, jeweils einem Anschluss für einen Schienenabschnitt und einer Weiche. Weiters sind – wie bereits erwähnt – Anschlüsse für die Kommunikation der Controller untereinander und vier LEDs für die Anzeige des aktuellen Zustands (Betriebsmodus, Kommunikation, etc.) vorhanden.
Der Master-PC dient zur Visualisierung der aktuellen Parameter und Steuerung der Züge und übergibt die Steuerinformationen an die Mikrocontroller.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

KAPITEL 1

Begriffserklärung

1.1 Master

Als *Master* wird jener PC bezeichnet, der zur Steuerung und Visualisierung des Modelleisenbahnnetzes dient. Auf diesem Rechner läuft eine in Ada geschriebene Software, die den Anwender bei der Steuerung seiner Züge auf einem beliebigen Gleissystem unterstützen soll. Es besteht die Möglichkeit, die Züge sowohl in einem analogen als auch in einem digitalen Betriebsmodus zu steuern. Das Ziel der Software-Entwicklung war, eine möglichst einfach und intuitiv zu bedienende, aber auch sichere und fehlertolerante Software zu schreiben, die auch von einem unerfahrenen Benutzer in Betrieb genommen werden kann.

1.2 Controller

Jene Platine (inklusive ihrer Software), die im Zuge dieser Diplomarbeit entwickelt wurde und die mit den Schienen und Weichen direkt verbunden ist, wird in diesem Dokument als *Controller*, aber auch *Knoten* oder *Node* bezeichnet. Die Software für den Controller wurde in Ada geschrieben. Ausschlaggebend für die Wahl der Programmiersprache war, dass sie sich sehr gut für sicherheitskritische und fehlertolerante Echtzeitsysteme eignet, aber auch die Tatsache, dass noch kaum Erfahrung mit der Programmiersprache Ada auf selbst entwickelten Hardwareumgebungen vorhanden ist. Der Aufbau eines Controllers, die Übersicht über die verwendeten Komponenten inklusive einer genauen Auflistung der Anforderungen werden im Kapitel 3 beschrieben.

1.3 Abschnitt

Das Schienennetz enthält *Unterbrecher-Elemente* (Isolatoren), die zwischen den Streckenteilen gesteckt werden und die leitende Verbindung unterbrechen. Somit entstehen im Gleissystem

Abschnitte, wobei für jeden Abschnitt ein Controller benötigt wird, der die geforderte Modellierung der Versorgungsspannung im Digital- und Analogbetrieb über die Schiene der Lokomotive bereitstellt. Die Zugmaschine muss im Digitalbetrieb über einen Digitaldecoder verfügen.

1.4 Sicherer (Betriebs-) Zustand

Als sicherer Betriebszustand gilt jene Situation am Master und den Knoten, in der alle Daten vollständig und korrekt übertragen werden können, es keine defekten Komponenten oder Softwarefehler gibt, die dazu führen könnten, dass es zu Kollisionen, Entgleisungen oder anderen Katastrophen kommen kann, bei der Menschen, Züge etc. gefährdet oder beschädigt werden.
Ein sicherer Zustand ist somit auch jene Situation, bei der alle Züge gestoppt werden, um oben genannte Probleme zu vermeiden. Das Ziel dieses Systems ist ein ständig sicherer Zustand. Kann diese Voraussetzung vorübergehend nicht erfüllt werden, so muss schnellst möglich reagiert und ein sicherer Betriebszustand wieder hergestellt werden.

KAPITEL 2

Aufbau

Schon aus der Begriffserklärung kann der prinzipielle Aufbau des Gesamtsystems erkannt werden, der aus den folgenden drei Teilen besteht:

1. Die Schienen inklusive Weichen und den Zügen.

2. Die Hardware.

3. Die Software, wobei hier noch zwischen der Software am Master und jener auf den Controllern unterschieden werden muss.

2.1 Schienen, Weichen und Züge

Bei der Entwicklung dieses Systems wurde ausschließlich auf Piko-A-Gleisen mit einer Kleinbahn-Zugmaschine, die mit einem digitalen Decoder ausgestattet ist, getestet. Somit war es möglich, sowohl im digitalen als auch im analogen Betrieb zu fahren. Prinzipiell können hier auch andere Systeme verwendet werden, sofern die elektronischen Komponenten der Controller der Belastung standhalten. Sowohl die Schienen und Weichen also auch die Zugmaschine wurden nicht modifiziert, das Modellbahnsystem kann also nahezu beliebig erweitert bzw. verändert werden, ohne in die Hard- und Software einzugreifen.

2.2 Hardware

Als Hardware werden die Knoten, also die Platinen mit ihrer Bestückung verstanden. Im Kapitel 3 wird auf die verwendeten Komponenten im Detail eingegangen.

2.3 Software

Bei der Software muss zwischen jener Software, die am Controller läuft und jener zur Steuerung des Systems am Master unterschieden werden. Das Kapitel 4 beschäftigt sich genau mit diesen beiden Teilen der Software, deren Aufbau und der detaillierten Funktionsweise.

KAPITEL 3

Hardware

Für die Steuerung der Züge musste die entsprechende Hardware geplant, entworfen und letztendlich gefertigt werden, wobei für das Design der Layout-Editor *Eagle* verwendet wurde und die Fertigung der Platine inklusive Ätzen der Leiterbahnen, aber ohne Bestückung an eine externe Firma ausgelagert wurde. Die Bestückung wurde selbst vorgenommen. Für die Auswahl der Hardware waren die Kriterien

- möglichst geringe Kosten und
- möglichst kleine Platinenausmaße

ausschlaggebend. Die Grafiken 3.1 und 3.2 zeigen das Layout der Platine und den zugehörigen Schaltplan. Um die benötigte Hardware und alle Leiterbahnen auf der Platine unterzubringen, wurde eine Größe von 5 cm mal 8 cm mit 2 Ebenen gewählt. Die folgenden Abschnitte erläutern die verwendeten Hardwarekomponenten im Detail.

3.1 Prozessor: Atmage8

In der ersten Version der Platine wurde ein *Attiny2313* Prozessor der Firma *Atmel* verwendet, der allerdings aufgrund einer zu geringen Flash-Größe (2 KB) dem *Atmega8* mit 8 KB Flash weichen musste. Da die ersten Entwürfe der Software in *C* geschrieben wurden, war mit 2 KB ausreichend Flash-Speicher vorhanden, nach der Umstellung auf *Ada* und einer Erweiterung der Anforderungen an die Software, musste ein neuer Prozessor ausgewählt werden. Aufgrund der unterschiedlichen Anordnung der Pins der beiden Prozessoren musste das Layout vollständig geändert werden. Für die Wahl eines passenden Prozessors waren die folgenden Kriterien ausschlaggebend:

- Der Prozessor muss in *SMD*-Bauweise erhältlich sein, um den Platzbedarf möglichst gering zu halten.

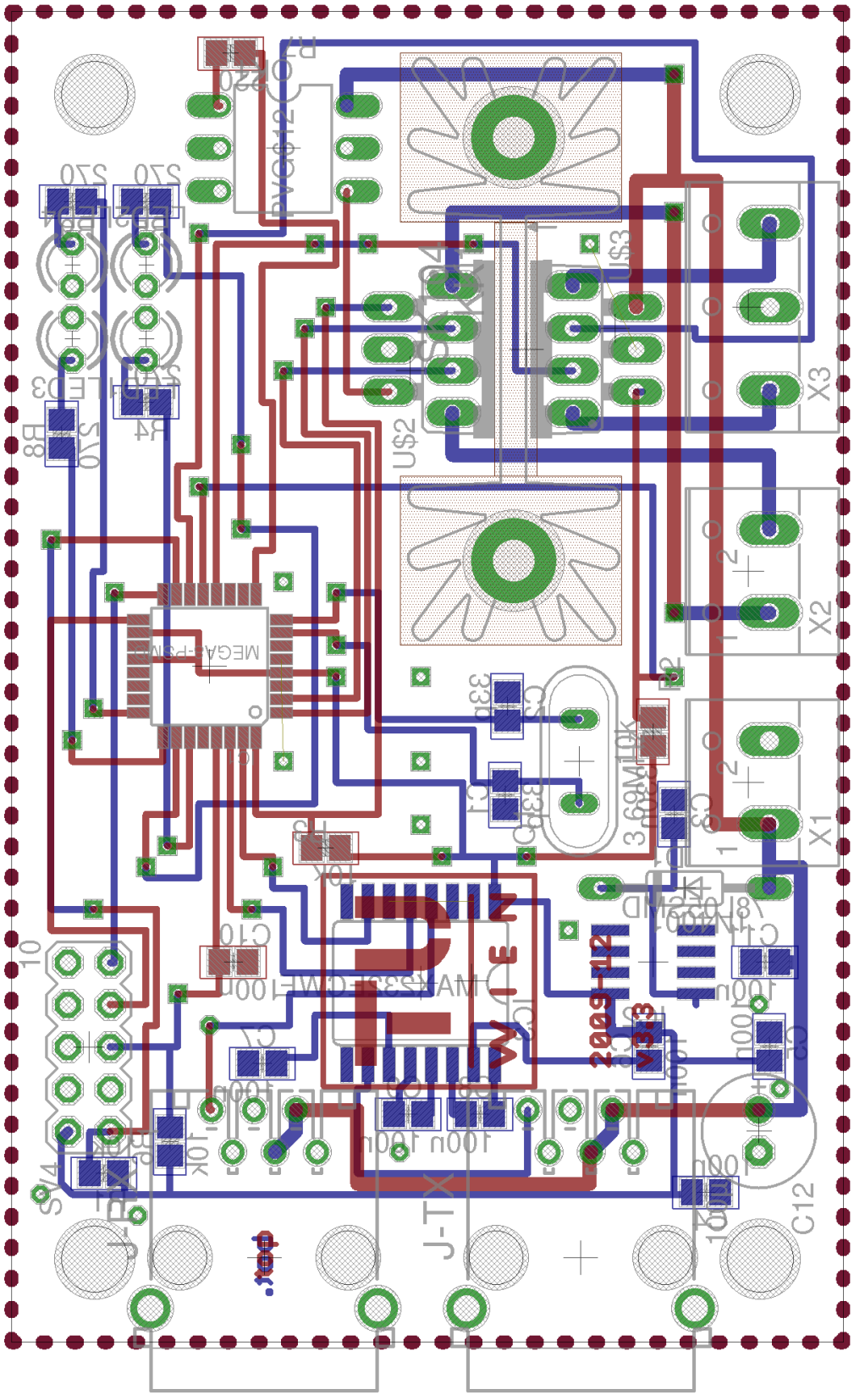

Abbildung 3.1: Platinen Layout - Version 4.0

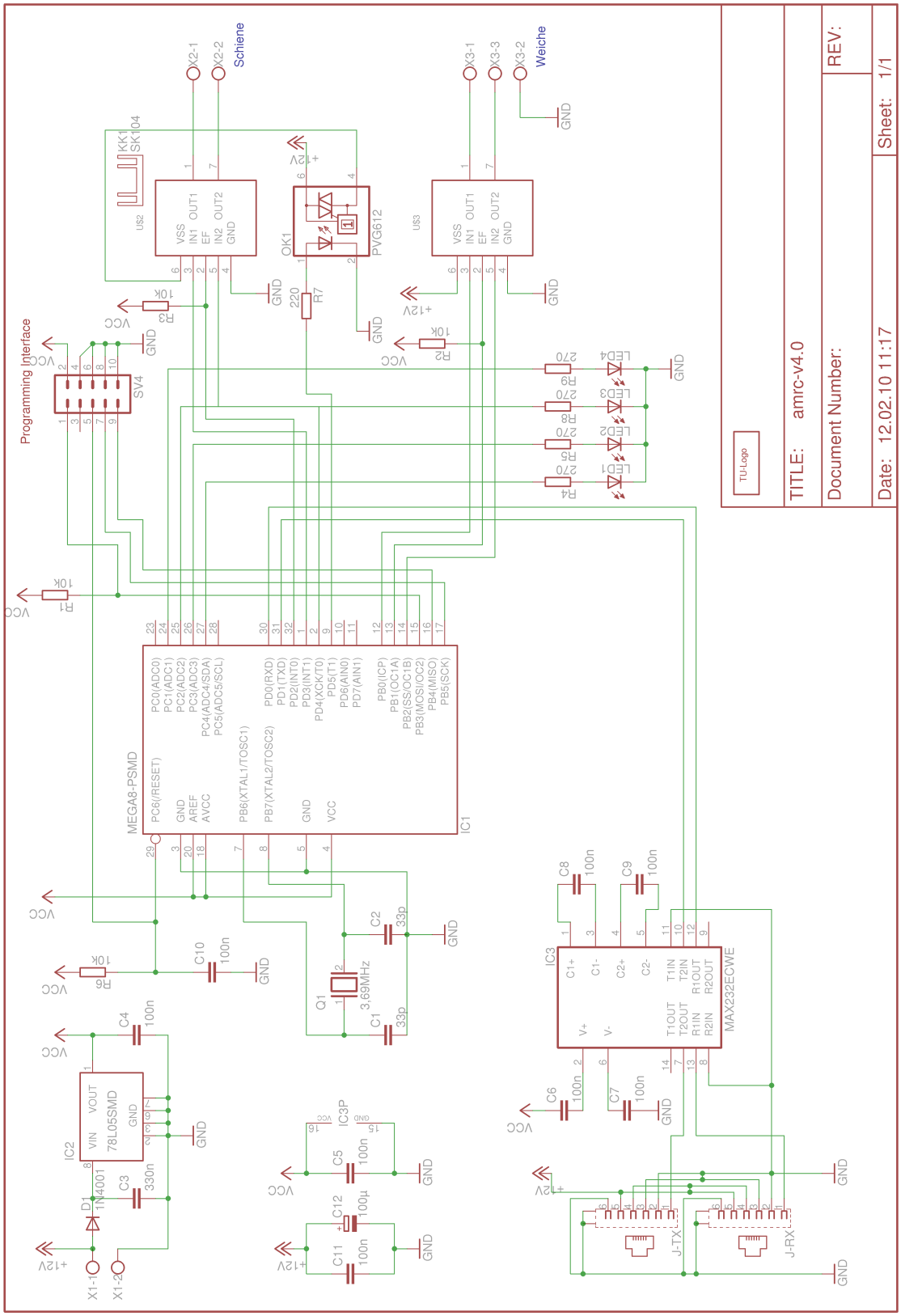

Abbildung 3.2: Schaltplan - Version 4.0

- Es müssen zumindest 21 Pins zur Verfügung stehen. Diese Zahl ergibt sich aus:
  - 2 Pins für die Spannungsversorgung VCC und GND
  - 4 Pins für das Programming-Interface ($RESET$, $MISO$, $MOSI$, SCK)
  - 2 Pins für den Anschluss des *MAX232* Spannungswandler (RXD, TXD)
  - 2 Pins für externe Quarz ($XTAL1$ und $XTAL2$)
  - 4 Pins für die Steuerung der LEDs
  - 3 Anschlüsse für den TLE zur Steuerung der Weichenstellung
  - 3 Anschlüsse für den TLE zur Steuerung der Zugmaschinenmotoren
  - 1 Pin für einen Optokoppler zum Ein- und Ausschalten der Versorgung des Motor-TLEs
- Es muss für die Kommunikation ein U(S)ART vorhanden sein.
- Für diverse Funktionen in der Software müssen 2 Timer/Counter zur Verfügung stehen.
- Es muss möglich sein, dass der gewählte Mikroprozessor mit dem STK500 Development-Kit [2] programmiert werden kann, da ansonsten die Anschaffung eines neuen Entwicklungsboards nötig wäre.
- Um an den Knoten eine in Ada geschriebene Software für den Prozessor zu compilieren, zu linken und letztendlich laufen zu lassen, muss der Prozessor von ADA-AVR unterstützt werden.
- Trotz dieser Anforderungen sollten die Kosten für den Mikrocontroller möglichst gering gehalten werden.

3.2 H-Brücke zur Steuerung der Weichenstellung

Bei jeder Weiche können zwei Stellungen eingenommen werden. Hier erzeugt eine Spule ein Magnetfeld, mit dem die Position der Schienen verändert werden kann. Dafür ist es nötig – je nach gewünschter Stellung – eine Spannung von +12 V bzw. -12 V bereit zu stellen. Um dies zu erreichen wird eine H-Brücke (TLE-5205-2) verwendet, die einen Eingang für die Versorgungsspannung von +12 V, 2 Eingänge zur Steuerung der Polung und 2 Ausgänge besitzt. Des weiteren steht noch ein Errorflag zur Verfügung, das aber in dieser Anwendung nicht benötigt wird. Die Ausgänge dieses Bausteins sind mit zwei Anschlussklemmen der Weiche verbunden. Die dritte Klemme ist mit GND verbunden. So kann mit der entsprechenden Schaltung der Eingänge eine Potentialdifferenz von 12 V zwischen X3-1 und X3-3 bzw. X3-2 und X3-3 aus der Abbildung 3.2 zur Verfügung gestellt werden, was ein entsprechend gepoltes Magnetfeld in der Spule zur Folge hat, das nun die Schiene in die geforderte Stellung zieht. Anschließend kann die Spannung wieder an den Ausgängen der Brücke deaktiviert werden, weil sich die Schiene in der gewünschten Position befindet und nur bei einem erneuten Stellvorgang Spannung angelegt

werden muss. Somit gibt es an den Weichen keinen unnötigen Stromverbrauch und die Bauteile werden keiner zusätzlichen Beanspruchung ausgesetzt. Es besteht ansonsten das Risiko einer Überhitzung an der Spule, was zur Beschädigung dieser und durch die Überlast zum Durchbrennen der Motorbrücken führen könnte.
Der genaue Aufbau, die Funktionsweise dieses Bauteils und die Schaltung der Eingänge kann dem Datenblatt [3] entnommen werden.

3.3 H-Brücke zur Steuerung der Gleichstrommotoren

Für die Steuerung der Gleichstrommotoren der Zugmaschine wurde ebenfalls die Motorbrücke *TLE-5205-2* verwendet, da es sowohl im Analog- als auch im Digitalbetrieb nötig ist, an den Gleisen +12 V und -12 V zur Verfügung stellen zu können. Für die Steuerung des Motors im Analogbetrieb wird je nach Fahrtrichtung +12 V oder -12 V für eine bestimmte Zeit zur Verfügung gestellt. Über die Eingänge der H-Brücke kann somit das folgende Verhalten der Motoren erreicht werden [3]:

- Der Motor dreht sich in Vorwärtsrichtung der Lokomotive.
- Der Motor dreht sich in Rückwärtsrichtung der Lokomotive.
- Der Motor wird gebremst.
- Der Motor befindet sich im Freilaufbetrieb (zur Erkennung ob ein Verbraucher vorhanden ist).

Wird der Knoten im Digitalbetrieb betrieben, so müssen innerhalb weniger Millisekunden Umpolungen zwischen +12 V und -12 V vorgenommen werden. Dies wird ebenfalls über die Ein- und Ausgänge der Brücke gesteuert (siehe Kapitel 4.4.1).

Diese Motorbrücke verträgt Lasten von maximal 5 A und Spannungen bis 40 V, was für einen Modellbahnbetrieb ausreichend ist. Ein wichtiger Punkt für die Wahl dieses Bauteils ist die Tatsache, dass die Umpolung intern absolut kurzschlussfrei durchgeführt wird. Des Weiteren existiert eine Fehlerprüfung (Kurzschluss, Überlast, Überhitzung) dessen Zustand (Fehler, kein Fehler) an einem Ausgang der Motorbrücke geprüft werden kann. Außerdem reagiert der TLE sehr schnell auf Änderungen an den Eingängen, was für die Modellierung der digitalen Pakete eine wichtige Voraussetzung ist. Obwohl der TLE sehr hitzebeständig (-40 C bis +150 C) ist, wird eine Kühlrippe auf der Platine angebracht, um einen besseren Temperaturaustausch mit der Umgebung zu gewährleisten, sodass eine unerwünschte Abschaltung des TLEs aufgrund von zu großer Hitze vermieden werden kann.

3.4 Optokoppler zum Ein- und Ausschalten der Versorgung

Da es über die Ein- und Ausgänge der H-Brücke nicht möglich ist, die Ausgänge von der Spannung zu trennen, wurde vor den 12-V-Eingang der Brücke ein Optokoppler vorgeschaltet, über

den die Versorgung ein- und ausgeschaltet werden kann. Dies ist für den Übertritt eines Zuges von einen in einen anderen Abschnitt wichtig (siehe Kapitel 4.2.2).

3.5 Spannungswandler für den UART: MAX232ECWE

Um die RS232-Schnittstelle am Knoten verwenden zu können, wurde noch ein *MAX232ECWE* Spannungswandler [4] der Firma *Maxim* auf der Platine angebracht, der mit den Pins TXD und RXD des Mikroprozessors und den Anschlüssen für die Kommunikationskabel verbunden ist.

3.6 Spannungswandler für die Eingangsspannung: 78L05SMD

Um die Eingangsspannung von 12 V, die für den Betrieb der Eisenbahnen nötig ist, auf die 5 V Versorgungsspannung des Mikroprozessors und des Spannungswandlers für die serielle Schnittstelle zu transformieren, wird ein Spannungwandler verwendet, der einen Ausgang für 5 V Gleichspannung bei einem maximalen Strom von 100 mA besitzt. Der Bauteil *78L05SMD* des Herstellers *STMicroelectronics* kommt hierfür zum Einsatz – die charakteristischen Parameter können dem Datenblatt [1] entnommen werden. Die VCC-Eingänge der TLE-Bauteile sind ohne Spannungswandler direkt mit dem 12-V-Eingang der Platine verbunden, da an den Gleisen 12 V benötigt werden.

3.7 Leds

Auf der Platine befinden sich vier LEDs, die zur Visualisierung des aktuellen Zustands dienen – eine Beschreibung kann im Kapitel Software unter 4.3.2 gefunden werden.

3.8 Programmierschnittstelle

Um den Mikroprozessor zu programmieren, wird eine 10-polige-Schnittstelle verwendet, die mit dem Prozessor verbunden ist. Über ein Development-Board (*Atmel STK500* [2]) wird die Software ins Flash des Mikrocontrollers programmiert. Auf der PC-Seite wird die Programmiersoftware `avrdude` mit dem Aufruf

```
1 avrdude -p m8 -P /dev/ttyUSB0 -c stk500v2 -U flash:w:
    controller.hex
```

verwendet, wobei die Parameter folgende Bedeutung haben:

- `-p m8` Es wird der Prozessor *Atmega8* verwendet.
- `-P /dev/ttyUSB0` Als Programmierschnittstelle kommt der USB0-Anschluss zur Anwendung.

- `-c stk500v2` Als Programmierwerkzeug wird das Developmentboard *Atmel STK500* verwendet.

- `-U flash:w:controller.hex` Die Programmierung erfolgt ins Flash des Prozessors, wobei die Datei `controller.hex` gewählt wurde.

Der Atmega8 besitzt zwei Fuse-Bytes (siehe Kapitel *Fuse Bits* in [5]), die unter anderem dafür verwendet werden, die Taktquelle zu wählen. Die initiale Programmierung dieser Bits erfolgt bei der Herstellung, wobei hier als Quelle der prozessorinterne 1 MHz Quarz gewählt wird. Für die Programmierung ist zu beachten, dass die Bits *low-active* sind und bei falscher Programmierung nicht mehr auf den Prozessor mit den herkömmlichen Methoden zugegriffen werden kann. Für den reibungslosen Betrieb werden die Werte $0xEF$ und $0xD9$ für das Low- bzw. High-Byte gewählt. Die Standard-Einstellung ab Werk wäre $0xE1$ und $0xD9$, wobei hier der einzige Unterschied darin besteht, dass als Taktquelle ein externer Quarz mit einer Frequenz zwischen 3.0 Mhz und 8.0 Mhz gewählt wurde, nachdem der Quarz auf der Platine eine Frequenz von 3.6869 Mhz liefert. Für die Programmierung der Fuse-Bit wird ebenfalls `avrdude` verwendet:

```
1 avrdude -p m8 -P /dev/ttyUSB0 -c stk500v2 -U lfuse:w:0xEF:m
2 avrdude -p m8 -P /dev/ttyUSB0 -c stk500v2 -U hfuse:w:0xD9:m
```

Die Parameter sind im wesentlichen dieselben wie jene bei der Programmierung der Software im Flash, der einzige Unterschied ist jener, dass die Programmierung in den Fuses erfolgt – in diesem Fall der Wert $0xEF$ in das Low-Byte (`lfuse`) und $0xD9$ ins High-Byte (`hfuse`).
Der Vorteil bei der Verwendung eines externen Quarz besteht darin, dass dieser wesentlich präziser und schneller als die CPU interne Taktung arbeitet und sich somit als Taktquelle für eine zuverlässige Übertragung von Daten über den UART eignet, weil es bei der Datenübertragung wichtig ist, dass die Taktung zwischen den Kommunikationsteilnehmern möglichst synchron erfolgt, da mit wachsenden Timingunterschieden auch die Fehlerrate ansteigt.

3.9 Kabelverbindungen

Für die Programmierung und den Betrieb sind zwei unterschiedliche Verbindungen zu einem PC nötig, die in den folgenden zwei Kapiteln beschrieben werden.

3.9.1 Programmierschnittstelle

Die Programmierung erfolgt – wie bereits beschrieben – über die 10-polige Schnittstelle am Controller, die mit dem Developmentboard *STK500* über ein 10-poliges Kabel verbunden wird. Das Entwicklungsboard wird über ein serielles Kabel mit dem PC verbunden und kann mittels `avrdude` programmiert werden (siehe Kapitel 3.8).

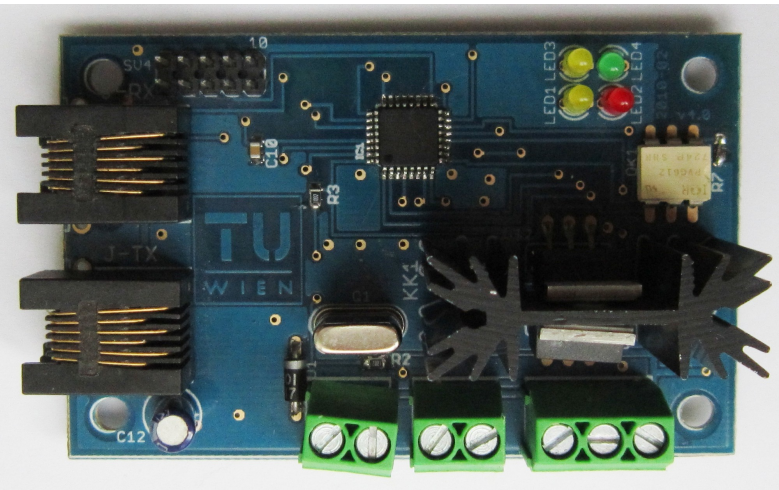

Abbildung 3.3: Fotografie eines fertigen Knotens

3.9.2 Serielle Schnittstelle zur Kommunikation im Betrieb

Die Knoten sind über 6-polige RJ11-Kabel miteinander verbunden, wobei die Daten ausschließlich an Pin 1 übertragen werden. Pin 5 ist mit der $12V$ Spannungsversorgung verbunden, was den Vorteil mit sich bringt, dass nur ein Knoten an die $12V$ Versorgung direkt angeschlossen werden muss und die anderen über die RJ11-Kabel versorgt werden. Die übrigen Pins sind ungenutzt und mit GND verbunden. Die Wahl ist deshalb auf diese Art von Kabel gefallen, da sie im Handel besorgt und ohne weitere Modifikation verwendet werden können. Über die Anschlüsse ist eine einfache Verwendung mit sicherer Befestigung möglich und die Kosten der Kabel können gering gehalten werden.

3.9.3 Platine

Die Abbildung 3.3 zeigt das Foto einer fertigen Platine und das folgende Listing die komplette Bestückungsliste als Export aus *Eagle*.

Part	Value	Device	**Package**	Library
C1	33p	C-EUC0805	C0805	rcl
C2	33p	C-EUC0805	C0805	rcl
C3	330n	C-EUC0805	C0805	rcl
C4	100n	C-EUC0805	C0805	rcl
C5	100n	C-EUC0805	C0805	rcl
C6	100n	C-EUC0805	C0805	rcl
C7	100n	C-EUC0805	C0805	rcl
C8	100n	C-EUC0805	C0805	rcl
C9	100n	C-EUC0805	C0805	rcl

C10	100n	C-EUC0805	C0805	rcl
C11	100n	C-EUC0805	C0805	rcl
C12	100u	CPOL-EUE2.5-7	E2,5-7	rcl
D1	1N4001	1N4148DO35-7	DO35-7	diode
IC1	MEGA8-PSMD	MEGA8-PSMD	TQFP32-08	amrc
IC2	78L05SMD	78L05SMD	SO08	linear
IC3	MAX232ECWE	MAX232ECWE	SO16L	maxim
J-RX		555154-1	555154-1	con-amp
J-TX		555154-1	555154-1	con-amp
LED1		LED3MM	LED3MM	led
LED2		LED3MM	LED3MM	led
LED3		LED3MM	LED3MM	led
LED4		LED3MM	LED3MM	led
OK1	PVG612	IL410	DIL06	amrc
Q1	3.69MHz	XTAL/S	QS	special
R1	10k	R-EU_R0805	R0805	rcl
R2	10k	R-EU_R0805	R0805	rcl
R3	10k	R-EU_R0805	R0805	rcl
R4	270	R-EU_R0805	R0805	rcl
R5	270	R-EU_R0805	R0805	rcl
R6	10k	R-EU_R0805	R0805	rcl
R7	220	R-EU_R0805	R0805	rcl
R8	270	R-EU_R0805	R0805	rcl
R9	270	R-EU_R0805	R0805	rcl
SV4		MA05-2	MA05-2	CON-LSTB
U2		TLE5205-2	TO-220-7-11	flohbunt
U3		TLE5205-2	TO-220-7-11	flohbunt
X1		W237-102	W237-102	con-wago-500
X2		W237-102	W237-102	con-wago-500
X3		W237-103	W237-103	con-wago-500

KAPITEL 4

Software

In diesem Kapitel wird die Software am Master und an den Knoten beschrieben. Dabei wird in den einzelnen Kapiteln auf die folgenden Teile der Software im Detail eingegangen:

- Die wichtigsten verwendeten Datenstrukturen
- Verwendete Algorithmen
- Erklärung der Betriebsmodi
- Statemachines am Master und auf den Knoten
- User Interface

4.1 Datenstrukturen

Dieses Kapitel beschäftigt sich mit den wichtigsten Datenstrukturen, die in der Software verwendet werden. In den Abschnitten werden anschließend folgende Strukturen beschrieben:

1. `TYPE_MESSAGE` zur Speicherung von Nachrichten bei der Kommunikation über den Datenring
2. `TYPE_NODE` und `TYPE_SWITCH` zur Speicherung von abschnittsspezifischen Daten
3. `TYPE_TRAIN` zur Speicherung von zugspezifischen Daten
4. `TYPE_SEARCH_ROUTE` und `TYPE_SEARCH_TRAIN_ROUTE` zur temporären Speicherung von Suchergebnissen bei der Suche nach Routen
5. `TYPE_ROUTE` zur Speicherung einer ausgewählten Route
6. `TYPE_CONTROLLER` zur Speicherung der Daten am Knoten

Die Datenstruktur in (1) wird sowohl vom Master als auch von den Knoten verwendet. (2) bis (5) sind lediglich am Master und die letzte Struktur ausschließlich an den Knoten im Einsatz.

4.1.1 UART-Nachricht zur Datenübertragung

Übersicht

Für die Kommunikation am Datenring (mittels UART) wurde eine Datenstruktur definiert, die sowohl vom Master als auch von allen Knoten verwendet wird (`TYPE_MESSAGE`). Die Elemente der Struktur werden in der Tabelle 4.1 aufgelistet und anschließend beschrieben. Bei jeder Datenübertragung wird ein Element dieser Struktur angelegt, von der Software befüllt und anschließend wird jedes Byte sequentiell über die serielle Schnittstelle übertragen bzw. auf der anderen Seite der Kommunikation Byte für Byte empfangen und die Datenstruktur befüllt.

Element-Name	**Datentyp**	**Initialisierung**	**Speicherbedarf [Byte]**
`MSG_Sender`	`Byte`	`B_MIN`	1
`MSG_Receiver`	`Byte`	`B_MIN`	1
`MSG_Type`	`TYPE_ MESSAGETYPE`	`MSG_TYPE_DATA`	1
`MSG_Train`	`Byte`	`B_MIN`	1
`MSG_Data`	`Byte`	`B_MIN`	1
`MSG_Config`	`Byte`	`B_MIN`	1
`MSG_Checksum`	`Byte`	`B_MIN`	1

Tabelle 4.1: Datenstruktur: TYPE_MESSAGE

Beschreibung der Elemente

MSG_Sender Dieses Element enthält die Adresse des Absenders der Nachricht. Da hier ein Byte zur Verfügung steht, kann ein Wertebereich von $0x00$ bis $0xFF$ mit folgender Einschränkung verwendet werden:

- Die Adresse $0x00$ ist immer die Master-Adresse.
- Die Adresse $0xFF$ gilt als Broadcast-Adresse, das bedeutet, dass sie nur als Empfängeradresse (siehe `MSG_Receiver`), nicht aber für Absenderadressen verwendet werden kann.
- Für den Adressbereich der Controller bleibt somit $0x01$ bis $0xFE$, wodurch sich insgesamt 254 Adressen im System verwenden lassen. Dies stellt somit auch das Maximum an Abschnitten in einem System dar, begründet darin, dass jeder Abschnitt über einen Controller gesteuert wird und jeder Controller über eine, im System eindeutige, Adresse verfügen muss.

MSG_Receiver Das Element `MSG_Receiver` enthält die Adresse des Empfängers einer Nachricht. Hier gelten dieselben Konventionen wie zuvor bei `MSG_Sender`. Nachrichten, die den Wert $0xFF$ als Empfängeradresse verwenden, werden von allen Knoten am Ring empfangen, die Daten werden verarbeitet und anschließend wird die Nachricht am Ring weitergeleitet.

MSG_Type Dieses Element enthält einen der folgenden Nachrichtentypen:

- `MSG_TYPE_INIT` (Initialisierungs-Nachricht): Dieser Typ wird beim Initialisierungsvorgang verwendet. Eine Beschreibung der Initialisierung (inklusive Kommunikation) kann dem Kapitel 4.3.1 entnommen werden.
- `MSG_TYPE_STOP` (Stop-Nachricht): Diese Nachricht signalisiert dem Controller, dass der Betrieb beendet wird und der Controller heruntergefahren werden muss.
- `MSG_TYPE_DATA` (Nutzdaten-Nachricht): Diese Nachricht wird zum Austausch der Nutzdaten für die Zugsteuerung (Fahrtrichtung und -geschwindigkeit, Beleuchtung), und für die Übermittlung von Konfigurationsdaten (Weichenstellung, Betriebsmodus und Controller-Status) an die Knoten verwendet.
- `MSG_TYPE_BUSY` (Busy-Nachricht): Eine Nachricht dieses Typs wird von einem Knoten an den Master gesendet, wenn ein Zug den Abschnitt des Knotens verlässt. Der Ablauf und die Verwendung wird in Kapitel 4.2.4 erklärt.

Weitere Informationen über die Verwendung dieser Typen können der Beschreibung der Algorithmen in Kapitel 4.2 und der Master- bzw. Controller-Statemachine in 4.3.1 und 4.3.2 entnommen werden.

MSG_Train Dieses Strukturelement enthält die Adresse des digitalen Decoders unter den folgenden Bedingungen:

- Die Initialisierung ist abgeschlossen und die Software befindet sich im normalen Betriebsmodus.
- Der Controller befindet sich im Digitalbetrieb.
- Es werden Nutzdaten (der Nachrichtentyp entspricht `MSG_TYPE_DATA`) oder eine Belegt-Nachricht (`MSG_TYPE_BUSY`) übertragen.

Für alle anderen Fälle wird dieses Byte nicht verwendet und enthält den Wert $0x00$.

MSG_Data Das Element `MSG_Data` enthält die Nutzdaten, abhängig vom Nachrichtentyp:

- `MSG_TYPE_INIT`: Aktueller Zählerstand für den Node-Counter während der Initialisierungsphase (siehe Kapitel 4.3.1)
- `MSG_TYPE_STOP`: Bei Stop-Nachrichten wird dieses Element nicht verwendet und enthält immer den Wert $0x00$.

- `MSG_TYPE_DATA`: Nutzdaten zur Zugsteuerung (siehe auch Kapitel 4.3.2). Dieses Byte setzt sich aus den folgenden Werten an den gegebenen Positionen zusammen:
  - Bit 1-4: Fahrgeschwindigkeit ($0x0 \ldots 0xF$)[1]
  - Bit 5: Beleuchtung (`LIGHT_ON`, `LIGHT_OFF`)
  - Bit 6: Fahrtrichtung (`DIR_FORWARD`, `DIR_BACKWARD`)
  - Bit 7-8: Nicht in Verwendung
- `MSG_TYPE_BUSY`: Bei diesem Nachrichtentyp wird das Nutzdaten-Element nicht verwendet und enthält – wie auch bei Stop-Nachrichten – den Wert $0x00$.

MSG_Config Dieses Element enthält die Konfigurationsdaten für den Controller, wobei nur bei einer Nutzdaten-Nachricht derartige Informationen übermittelt werden. Bei den anderen Nachrichtentypen enthält dieses Feld den Wert $0x00$.

- Bit 1: Weichenstellung (`POS_AHEAD`, `POS_TURN`)
- Bit 2: Betriebsmodus (`MODE_DIGITAL`, `MODE_ANALOG`)
- Bit 3: Controller-Status (`STATUS_GO`, `STATUS_STOP`)
- Bit 4-8: Nicht in Verwendung

MSG_Checksum Um eine fehlerhafte Übertragung (z.B. durch ein umgefallenes Bit) zu erkennen, enthält das letzte Byte der Nachricht eine Checksumme, die über eine **XOR**-Verkettung der anderen sechs Bytes unmittelbar vor der Übertragung berechnet wird. Beim Empfang einer Nachricht wird die Checksumme der ersten sechs Bytes am Empfänger berechnet und mit der empfangenen Checksumme (Byte sieben der Nachricht) verglichen. Bei unterschiedlichen Werten ist bei der Übertragung ein Fehler aufgetreten und die empfangenen Daten müssen aus Gründen der Betriebssicherheit verworfen werden. Andernfalls könnte es zu einem unerwünschten Verhalten führen und zu einer Kollision oder einer Entgleisung kommen.

Speicherbedarf

Eine Nachricht besteht aus den oben beschriebenen Elementen, für die jeweils ein Byte zur Verfügung steht. Somit ergeben sich in Summe 7 Byte, die für eine Nachricht benötigt werden.

Beispiel

Die Tabelle 4.2 zeigt ein Beispiel einer Nachricht im Betrieb.

Hierbei handelt es sich um eine Nutzdatennachricht (`MSG_TYPE_DATA`), gesendet von der Adresse $0x00$ (Master) an den Controller mit der Adresse $0x03$ mit dem Auftrag, dass für die

[1] Für den Fall, dass im User Interface fünf Bits für die Geschwindigkeit gewählt wurden, wird das Bit 5 (Beleuchtung) als fünftes Geschwindigkeitsbit verwendet. In dieser Arbeit wird in den nachfolgenden Kapiteln bei der Beschreibung der Funktionalität von nur vier Bits ausgegangen.

Element-Name	Wert
MSG_Sender	$0x00$
MSG_Receiver	$0x03$
MSG_Type	MSG_TYPE_DATA
MSG_Train	$0x05$
MSG_Data	$0x2A$
MSG_Config	$0x02$
MSG_Checksum	$0x28$

Tabelle 4.2: Beispiel: TYPE_MESSAGE

Lokomotive mit der Adresse $0x05$ die Nutzdaten $0x2A$ bereitgestellt werden müssen. Außerdem werden die Konfigurationsdaten $0x02$ übertragen. Die Checksumme $0x2D$ ergibt sich aus der **XOR**-Verknüpfung der Werte $0x00$, $0x03$, $0x03$ (MSG_TYPE_DATA), $0x05$, $0x2A$ und $0x02$.

4.1.2 Speicherung der Nachbarn

Übersicht

Für die Software ist es erforderlich (z.B. für die Berechnung der Routen), dass zu jedem Knoten seine Nachbarn bekannt sind, das heißt welche Controlleradresse in beiden Fahrtrichtung und für jede Richtung über jeweils beide Weichenstellungen erreicht werden können. Um diese Daten zu speichern wurde die Datenstruktur TYPE_SWITCH eingeführt, deren Elemente in Tabelle 4.3 dargestellt werden:

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
Switch_Ahead	Byte	B_MIN	1
Switch_Turn	Byte	B_MIN	1

Tabelle 4.3: Datenstruktur: TYPE_SWITCH

Beschreibung der Elemente

Switch_Ahead Dieses Element enthält die Adresse des Controllers, der mit gerade Weichenstellung POS_AHEAD erreicht wird. Der Wert $0x00$ zeigt an, dass es keinen Nachbarn über diese Weichenstellung gibt.

Switch_Turn Hier wird die Adresse jenes Controllers gespeichert, der mit der Weichenstellung POS_TURN erreicht werden kann. Hier wird ebenfalls die Adresse $0x00$ dazu verwendet, um zu zeigen, dass kein Nachbar über diese Weichenstellung existiert.

Speicherbedarf

Diese Datenstruktur besteht lediglich aus zwei Elementen, die jeweils den Typ `Byte` haben und resultierend daraus benötigt die Struktur insgesamt 2 Byte an Speicherplatz.

Beispiel

Die Tabelle 4.4 zeigt ein Beispiel zur Speicherung der zuvor beschriebenen Daten.

Element-Name	Wert
Switch_Ahead	$0x04$
Switch_Turn	$0x06$

Tabelle 4.4: Beispiel: TYPE_SWITCH

Das Beispiel zeigt die Datenstruktur eines Controllers für eine bestimmte Fahrtrichtung, wobei der Nachbar, der mit der Weichenstellung `POS_AHEAD` erreicht wird, die Adresse $0x04$ und jener, der mit der Weichenstellung `POS_TURN` erreicht wird, den Wert $0x06$ besitzt.

4.1.3 Speicherung der Knotendaten

Dieser Abschnitt befasst sich mit der Speicherung von abschnittsspezifischen Daten, welche in der Datenstruktur `TYPE_NODE` zusammengefasst werden. Da diese Informationen für jeden Abschnitt gespeichert werden müssen, werden die Daten in einem Array (`TYPE_NODE_ARRAY`) abgelegt, dessen Größe der maximalen Anzahl der Knoten (`MAX_NODES`) entspricht. Die Tabelle 4.5 zeigt eine Übersicht dieses Datentyps.

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
Node_Forward	TYPE_SWITCH	-	2
Node_Backward	TYPE_SWITCH	-	2
Node_Busy	Boolean	false	1
Node_Mode	TYPE_ CONTROLLER_ MODE	MODE_DIGITAL	1
Node_Status	TYPE_ CONTROLLER_ STATUS	STATUS_GO	1

Tabelle 4.5: Datenstruktur: TYPE_NODE

Beschreibung der Elemente

Node_Forward Dieses Element enthält die beiden Nachbarn für die Fahrtrichtung vorwärts (`DIR_FORWARD`, siehe Kapitel 4.1.2).

Node_Backward Dieses Element enthält die beiden Nachbarn für die Fahrtrichtung rückwärts (`DIR_BACKWARD`, siehe Kapitel 4.1.2).

Node_Busy Dieses Element stellt das so genannte *Busy-Flag* dar und zeigt an, ob der aktuelle Abschnitt gerade von einem Zug befahren wird.

Node_Mode Dieses Element enthält den derzeitigen Betriebsmodus des Knotens (Analogbetrieb, Digitalbetrieb)

Node_Status Dieses Element enthält den aktuellen Controller-Status, wobei beim Status-Wert `STATUS_STOP` die Versorgung der Schienen deaktiviert und bei `STATUS_GO` diese aktiviert ist.

Speicherbedarf

Insgesamt ergeben sich 7 Bytes an Speicherplatzbedarf am Master pro Abschnitt. Mit einer maximal möglichen Anzahl von Abschnitten (254) ergibt sich in Summe ein Speicherbedarf von

$$7 \text{ Byte} \cdot 254 = 1778 \text{ Byte} \tag{4.1}$$

Durch die fixe Größe des Arrays wird unter Umständen mehr Speicherplatz reserviert, als während des Betriebs benötigt wird. Alternativ dazu wäre die Speicherung der Controllerdaten in einer Liste möglich. Mit dieser Methode könnte Speicherplatz gespart werden, vor allem bei Systemen mit wenigen Knoten, da pro Knoten nur jeweils ein Element zur Liste hinzugefügt werden müsste und nicht das komplette Array reserviert wird.

Die Entscheidungsgrundlage, ein Array stattdessen zu verwenden, liegt darin, dass Zugriffe auf Elemente des Arrays einfach (über einen Index) und schnell erfolgen, während durch die Liste iteriert werden müsste und dies bei langen Listen zu langen Abarbeitungszeiten führen würde und die Zugriffszeit somit auch abhängig von der Position wäre. Im Array hingegen wird für jedes Element dieselbe Zugriffszeit benötigt, die bei einer Liste nur beim Zugriff auf das erste Element erreicht werden könnte.

Da es sich bei dieser Eisenbahnsteuerung um ein sicherheitskritisches System handelt, bei dem jeder Speicherzugriff möglichst schnell erfolgen und für jedes Element dieselbe Zeit benötigen sollte, wird ein Array verwendet und der erhöhte Speicherplatz zugunsten der Zugriffsgeschwindigkeiten in Kauf genommen. Außerdem stellen moderne Rechner ein Vielfaches an Arbeitsspeicher zur Verfügung und es kann die Software beim Faktor Geschwindigkeit optimiert und der Speicherplatzbedarf als eher unwichtigeres Kriterium betrachtet werden.

Beispiel

Die Tabelle 4.6 zeigt ein Beispiel zur Speicherung der Knotendaten im Betrieb.

Element-Name	Wert
Node_Forward	$0x02 \vert 0x03$
Node_Backward	$0x04 \vert 0x08$
Node_Busy	false
Node_Mode	MODE_DIGITAL
Node_Status	STATUS_GO

Tabelle 4.6: Beispiel: TYPE_NODE

Dieses Beispiel zeigt einen Knoten, der in Fahrtrichtung vorwärts (DIR_FORWARD) mit den Knoten $0x02$ bei gerader Weichenstellung und $0x03$ bei Weichenstellung POS_TURN und rückwärts (DIR_BACKWARD) mit den Knoten $0x04$ (gerade Weiche) und $0x08$ (gestellte Weiche) verbunden ist. Das Busy-Flag steht auf false, das bedeutet, dass sich auf dem Abschnitt derzeit kein Zug befindet. Der Controller befindet sich im digitalen Betriebsmodus (MODE_DIGITAL), und die Spannungsversorgung der Schienen ist aktiviert (STATUS_GO).

4.1.4 Speicherung der zugspezifischen Daten

Übersicht

Das Schienensystem wird in Abschnitte unterteilt, wobei mindestens ein Abschnitt vorhanden sein muss und maximal 254 Abschnitte vorhanden sein dürfen – die Erklärung hierfür ist in Kapitel 4.1.1 zu finden. In der Software kann dieses Maximum allerdings herabgesetzt werden, was in geringerem Speicherbedarf und einer schnelleren Verarbeitung bestimmter Prozesse resultieren würde. Hierfür steht die Konstante MAX_NODES zur Verfügung.
Ebenso muss die maximale Anzahl der im System verwendeten Züge beschränkt werden. Hierfür existiert die Konstante MAX_TRAINS, dessen Maximalwert MAX_NODES-1 (also maximal 253) beträgt, um sicherzustellen, dass mindestens ein freier Abschnitt existiert, der von einem Zug befahren werden kann, da ansonsten von Beginn an eine Deadlock-Situation vorherrscht. Prinzipiell kann dieser Wert auch beliebig verringert werden.
Jeder Zug kann sich auf jedem Abschnitt unterschiedlich verhalten, das bedeutet, es kann die Geschwindigkeit und der Zustand der Beleuchtung auf den einzelnen Abschnitten variieren. Zusätzlich ist die Stellung der Weiche vom Zug abhängig. Daraus ergibt sich eine Datenstruktur aus Tabelle 4.7, die dazu verwendet wird, die zugspezifischen Daten zu speichern. Dieser Typ wird zu einem Array (TYPE_TRAIN_NODE_ARRAY) der Größe MAX_NODES zusammengefasst, womit für jeden Knoten dessen Zug-Parameter gespeichert werden können.

Beschreibung der Elemente

Train_Position Das Element Train_Position steht für die Weichenstellung, wobei hier POS_AHEAD für die gerade Weichenstellung und POS_TURN zum Abbiegen verwendet wird.

Train_Speed In diesem Element wird die Fahrtgeschwindigkeit gespeichert und kann die Werte $0x00$ bis $0x0F$ annehmen. $0x00$ steht für Stop und $0x0F$ für die maximale Geschwindigkeit.

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
Train_Position	TYPE_ SWITCHPOSITION	POS_AHEAD	1
Train_Speed	Byte	SPEED_MIN	1
Train_Light	Byte	LIGHT_OFF	1

Tabelle 4.7: Datenstruktur: TYPE_TRAIN

Train_Light Über dieses Element kann die Beleuchtung bzw. der Scheinwerfer der Lokomotive (falls vorhanden) im Digitalbetrieb ein- bzw. ausgeschaltet werden. Andernfalls hat dieser Wert keine Auswirkung.

Speicherbedarf

Nachdem jedes Element der Struktur genau ein Byte an Speicher benötigt, ergibt sich ein Bedarf von insgesamt drei Bytes an Zugdaten pro Knoten. Wie bereits in der Einleitung dieses Kapitels beschrieben, können maximal 254 Knoten über diese Software im System vorhanden sein. Resultierend daraus ergibt sich ein minimaler und maximaler Speicherbedarf bei den Zugdaten von

$$S_{Min} = S(1) = 3 \text{ Byte} \cdot 1 = 3 \text{ Byte} \tag{4.2}$$

$$S_{Max} = S(254) = 3 \text{ Byte} \cdot 254 = 762 \text{ Byte} \tag{4.3}$$

In den Gleichungen 4.3 und 4.3 steht $S(n)$ für den benötigten Speicherplatz und n für die Anzahl der Knoten ($S_{Min} = S(1)$ entspricht dem minimalen bzw. $S_{Max} = S(254)$ dem maximalen Bedarf).

Beispiel

Die Tabelle 4.8 zeigt ein Beispiel zur Speicherung der Zugdaten.

Element-Name	Wert
Train_Position	POS_AHEAD
Train_Speed	$0x0A$
Train_Light	LIGHT_OFF

Tabelle 4.8: Beispiel: Train_Value

In diesem Beispiel handelt es sich um einen Zug, dessen Geschwindigkeit den Wert $0x0A$ besitzt, das Licht deaktiviert ist ($0x00$) und die Weichen auf POS_AHEAD gesetzt wurde.

4.1.5 Temporäre Speicherung einer Route

Übersicht

Eine Anforderung an diese Arbeit war es, Routen, für die ein Start- und ein Zielknoten gewählt werden, zu berechnen und Züge über diese Routen automatisch fahren zu lassen. Dies wird dem Anwender über einen Dialog (siehe Kapitel 4.5) ermöglicht, bei dem er nach bestimmten Kriterien nach Routen suchen kann. Die Suchergebnisse müssen anschließend zur Verfügung gestellt werden, damit der Anwender seine Auswahl treffen kann. Um die entsprechenden Daten zur Verfügung stellen zu können, müssen diese in bestimmten Datenstrukturen abgelegt werden. Eine einfache Route mit ihren Informationen wird in einer Struktur aus Tabelle 4.9 gespeichert.

Element-Name	**Datentyp**	**Initialisierung**	**Speicherbedarf [Byte]**
`Route_ID`	`Word`	$0x0000$	2
`Route_Direction`	`TYPE_DIRECTION`	-	1
`Route_Position`	`TYPE_SWITCH_POSITION_ARRAY`	-	254
`Route_Length`	`Byte`	$0x00$	1

Tabelle 4.9: Datenstruktur: TYPE_SEARCH_ROUTE

Beschreibung der Elemente

Route_ID Dieses Element der Datenstruktur enthält die ID der Route. Diese beginnt beim Wert $0x0001$ und wird für jedes Ergebnis der aktuellen Suche inkrementiert und der Route zugewiesen. Der Nutzen liegt lediglich darin, eine Route, die über den Dialog ausgewählt wurde, wieder zu finden.

Route_Direction Für jede Route wird die Fahrtrichtung des Zuges (`DIR_FORWARD` oder `DIR_BACKWARD`) in diesem Element gespeichert.

Route_Position Dieses Element enthält ein Array, wobei jedes Element für die Stellung der Weiche des jeweiligen Knotens (Adresse = Index des Arrays) steht. Über die Weichenstellung und die Fahrtrichtung kann ermittelt werden, welcher Knoten als nächstes befahren wird.

Route_Length Dieses Element enthält die Länge (Anzahl der vorhandenen Abschnitte inklusive Start- und Zielknoten) der Route.

Speicherbedarf

Auch für diese Datenstruktur spielt die Größe des Systems eine Rolle, weil das Array für die Position der Weichenstellung von der Anzahl der Knoten abhängt. Das Maximum wird – wie

bereits erwähnt – über die Konstante MAX_NODES festgelegt werden. Insgesamt ergibt sich somit ein Speicherbedarf von maximal 258 Byte.

Beispiel

Die Tabelle 4.10 zeigt ein Beispiel, in dem eine einzelne Route gespeichert ist.

Element-Name	Wert
Route_ID	$0x0005$
Route_Direction	DIR_BACKWARD
Route_Position	POS_AHEAD ...
Route_Length	$0x05$

Tabelle 4.10: Beispiel: TYPE_SEARCH_ROUTE

In diesem Beispiel handelt es sich um eine Route mit der ID $0x0005$ (5. Ergebnis der Suche), die aus fünf Abschnitten besteht und mit den entsprechenden Weichenstellungen rückwärts durchfahren wird.

4.1.6 Speicherung von Routen

Übersicht

Über die zuvor beschriebene Struktur können die Daten einer einzelnen Route gespeichert werden. Um alle Routen inklusive weiterer Daten zu speichern, benötigt man eine Datenstruktur, deren Aufbau aus der Tabelle 4.11 entnommen werden kann.

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
Route_Source	Byte	B_MIN	1
Route_Sink	Byte	B_MIN	1
Route_List	TYPE_NODE_PTR	-	-

Tabelle 4.11: Datenstruktur: TYPE_SEARCH_TRAIN_ROUTE

Beschreibung der Elemente

Route_Source In diesem Element wird die Adresse des Startknotens gespeichert.

Route_Sink In diesem Element wird die Adresse des Zielknotens gespeichert.

Route_List Hinter diesem Element verbirgt sich ein Zeiger auf eine Liste, in der jedes Suchergebnis (siehe vorheriges Kapitel) eingetragen wird. Jedes Listenelement besteht aus einem Element des Typs TYPE_SEARCH_ROUTE für die Routeninformationen und einem Zeiger auf

das nächste Element der Liste. Um dem Anwender die Ergebnisse im Dialog zur Verfügung zu stellen, wird durch diese Liste iteriert und die Routeninformation auf eine anwenderfreundliche Form gebracht und im Dialog ausgegeben.
Die Abbildung 4.1 zeigt den Zusammenhang zwischen den beiden vorgestellten Datenstrukturen anhand eines Beispiels, bei dem drei Routen – beginnend beim Knoten 02 zum Knoten 07 führend – gefunden wurden.

Speicherbedarf

Für die Ermittlung des Speicherbedarfs aller Routen kann nicht wie bisher vorgegangen werden, da noch keine Aussage über die maximale Länge der Liste aller Routen getroffen werden kann. Die Tabellen 4.12 und 4.13 zeigen eine Übersicht über alle möglichen Routen eines Systems, das aus zwei bzw. drei Knoten besteht. Der Startknoten ist in allen drei Fällen jener mit der Controlleradresse 01, das Ziel jener mit Adresse 02 bzw. 03. In den Spalten zwischen den Controlleradressen wird angezeigt, ob es sich um die Weichenstellung `POS_AHEAD` (-) oder `POS_TURN` (+) handelt.
Über die folgende Überlegung kann nun die maximale Anzahl aller theoretisch möglichen Routen berechnet werden:

- Für zwei Knoten bestehen maximal zwei Möglichkeiten in Fahrtrichtung `DIR_FORWARD` vom ersten den zweiten zu erreichen:
  - Mit der Weichenstellung `POS_AHEAD` von 01 nach 02 (Zeile 1 der Tabelle 4.12)
  - Mit der Weichenstellung `POS_TURN` von 01 nach 02 (Zeile 2 der Tabelle 4.12)
- Für zwei Knoten bestehen maximal zwei Möglichkeiten in Fahrtrichtung `DIR_BACKWARD` vom ersten den zweiten zu erreichen:
  - Mit der Weichenstellung `POS_AHEAD` von 01 nach 02 (Zeile 3 der Tabelle 4.12)
  - Mit der Weichenstellung `POS_TURN` von 01 nach 02 (Zeile 4 der Tabelle 4.12)

1	01	-	02	DIR_FORWARD
2	01	+	02	DIR_FORWARD
3	01	-	02	DIR_BACKWARD
4	01	+	02	DIR_BACKWARD

Tabelle 4.12: Route mit zwei Abschnitten (von 01 nach 02)

Für eine Berechnung kann nun festgehalten werden, dass zwischen zwei Knoten, bei denen es maximal zwei Weichenstellungen für jeweils eine der beiden Richtungen geben kann, insgesamt vier Möglichkeiten der direkten Verbindung existieren können (siehe Tabelle 4.12). Für drei Knoten existieren dann insgesamt acht mögliche Routen vom Knoten mit der Adresse 01 zu jenem mit der Adresse 03 zu gelangen, für vier Knoten wären es schon 16 Routen usw. Für diese Berechnung wurden allerdings nur Routen mit der maximalen Länge eingerechnet. Zusätzlich

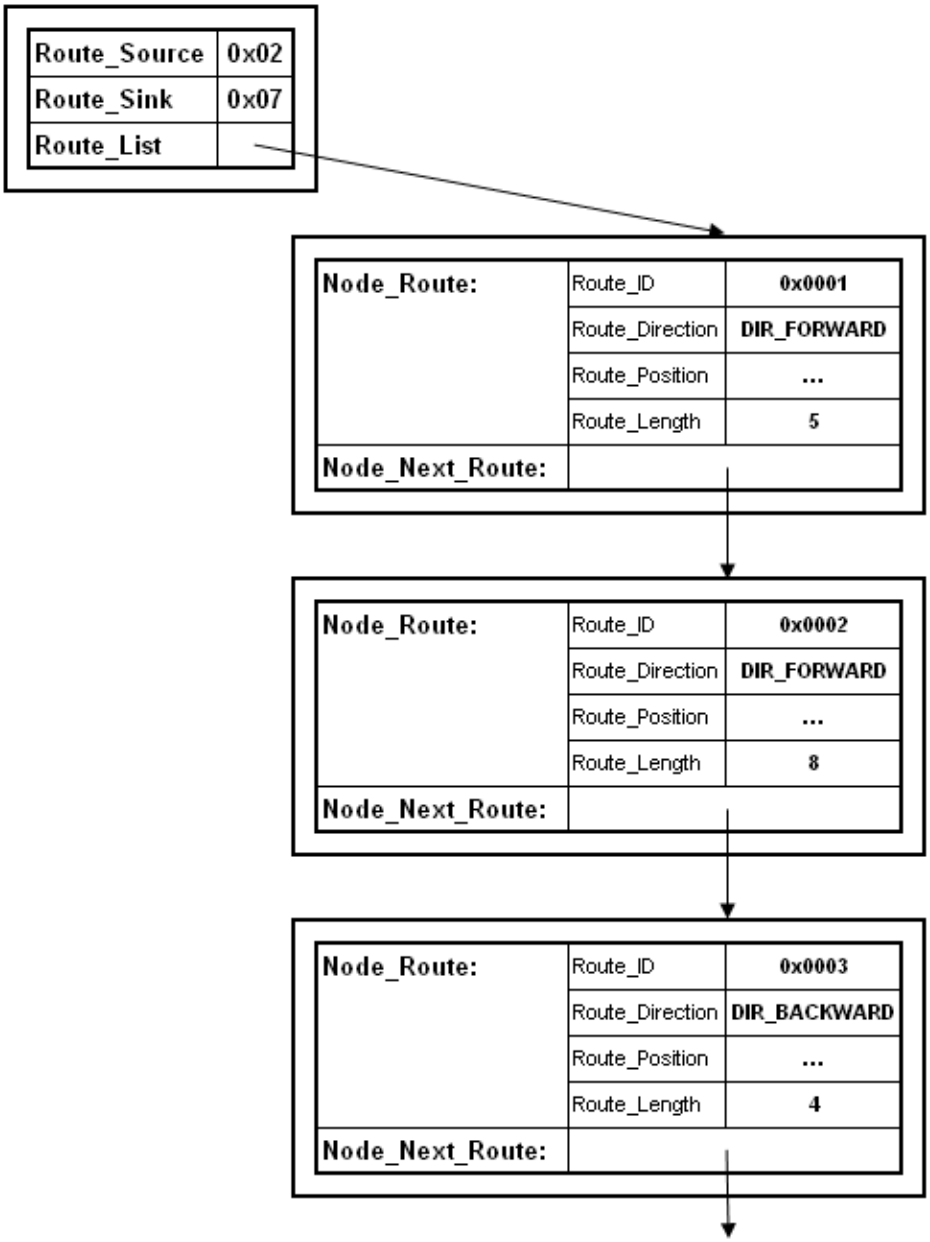

Abbildung 4.1: Übersicht über die temporäre Speicherung von Routen

1	01	-	02	-	03	DIR_FORWARD
2	01	-	02	+	03	DIR_FORWARD
3	01	+	02	-	03	DIR_FORWARD
4	01	+	02	+	03	DIR_FORWARD
5	01		-		03	DIR_FORWARD
6	01		+		03	DIR_FORWARD
7	01	-	02	-	03	DIR_BACKWARD
8	01	-	02	+	03	DIR_BACKWARD
9	01	+	02	-	03	DIR_BACKWARD
10	01	+	02	+	03	DIR_BACKWARD
11	01		-		03	DIR_BACKWARD
12	01		+		03	DIR_BACKWARD

Tabelle 4.13: Route mit drei Abschnitten (von 01 nach 03)

müssen noch alle Wege vom Start zum Ziel mit einer geringeren Länge berücksichtigt werden. Für die Berechnung bzw. zur Herleitung einer Formel für die Berechnung aller möglichen Routen zwischen zwei Knoten mit maximal n Abschnitten müssen die folgenden Punkte einbezogen werden:

- Es gibt 2^{n-1} Möglichkeiten um von einem Start- zu einem Zielknoten in eine bestimmte Fahrtrichtung mit insgesamt n Abschnitten zu gelangen.
- Um auch die zweite Fahrtrichtung zu berücksichtigen, muss dieser Wert noch mit 2 multipliziert werden. Somit erhält man bereits $2 \cdot 2^{n-1} = 2^n$ Routen.
- Wenn nun der Start- und Zielknoten als fix angesehen werden, gibt es dazwischen noch $n - 2$ Knoten, für die noch alle Variationen berücksichtigt werden müssen, das wären $(n - 2)!$. Für die Gesamtberechnung ergibt das einen Wert von $(n - 2)! \cdot 2^n$ Routen, wobei bisher nur Routen der Länge n berücksichtigt wurden.
- Um nun auch kürzere Strecken zu berücksichtigen, muss diese Berechnung für alle möglichen Längen $(2 \ldots n)$ aufsummiert werden, wobei bei den kürzeren Routen noch der Binomialkoeffizient (alle ungeordneten Möglichkeiten der Länge $k-2$) eingerechnet werden muss. Somit ergibt sich für das Maximum an möglichen Routen

$$R(n) = \sum_{k=2}^{n} \binom{n-2}{k-2} \cdot (k-2)! \cdot 2^k \tag{4.4}$$

  wobei $R(n)$ hier für die Anzahl der Routen mit maximal n Abschnitten steht.

Es ist leicht ersichtlich, dass diese Berechnung für große n sehr hohe Ergebnisse liefert. Dies wäre allerdings nur dann theoretisch möglich, wenn alle Knoten sternförmig verbunden wären, wobei die physikalischen Gegebenheiten maximal vier verschiedene Nachbarn (2 Richtungen, 2 Weichenstellungen) pro Abschnitt erlauben. Prinzipiell kann man sich alle möglichen Routen

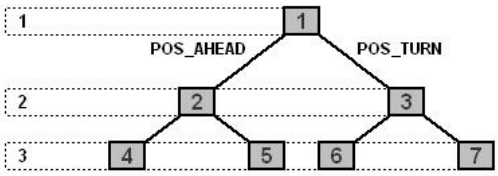

Abbildung 4.2: Baumdarstellung von Routen

als binären Baum vorstellen, wobei in diesem Fall eine Fahrtrichtung fix ausgewählt wird. Die Wurzel des Baumes stellt der Startknoten dar, die einzelnen Blätter sind die anderen Knoten der Route. Die Abbildung 4.2 zeigt ein Beispiel mit dem Startknoten 1 und einem maximal aufspannenden, ausbalancierten binären Baum mit drei Ebenen. Binäre Bäume dieses Aufbaus enthalten die maximale Anzahl an Kanten und somit auch die maximale Anzahl an möglichen Routen. In der ersten Ebene befindet sich der Startknoten (Wurzel) und es gibt 2 Möglichkeiten der Weichenstellung (`POS_AHEAD` und `POS_TURN`) um im Baum in die nächste Ebene zu gelangen, somit gibt es in dieser zweiten Ebene zwei Knoten, die das Ziel sein könnten. Sind sie es nicht, so muss in die nächst tiefere Ebene weiter iteriert werden. Die Anzahl der Ebenen wird mit m und die Anzahl der Knoten mit n bezeichnet. Aus den zuvor angestellten Überlegungen ergibt sich nun für die maximale Anzahl an Routen $R'(n)$ mit n Knoten in unserem System

$$R'(n) = \sum_{k=1}^{m} 2^k \tag{4.5}$$

wobei die Anzahl der Ebenen über $m = \lceil log_2(n) \rceil$ berechnet wird. Für unser System mit maximal $n = 254$ Knoten ergibt sich somit im Extremfall ein binärer Baum mit $m = 8$ Ebenen und somit höchstens

$$R'(254) = \sum_{k=2}^{8} 2^k = 510 \tag{4.6}$$

Routen zwischen einem bestimmten Start und Zielknoten. Wie bereits in Kapitel 4.1.5 beschrieben, benötigt jede Route 258 Bytes, woraus sich eine Gesamtsumme von

$$258 \text{ Byte} \cdot 510 + \underbrace{2 \text{ Byte}}_{\text{Start und Ziel}} = 131.582 \text{ Byte} = 128 \text{ KByte} \tag{4.7}$$

für alle möglichen Routen inklusive aller nötigen Informationen ergibt. Da für jeden Zug eine Route ausgewählt wird, müssen die Routendaten pro Zug gespeichert werden. Mit einer Maximalzahl von 253 Zügen im System ergibt das

$$128 \text{ KByte} \cdot 253 = 32.256 \text{ KByte} = 31,5 \text{ MByte} \tag{4.8}$$

an Speicherbedarf, wenn es 254 Knoten gibt, wobei alle Knoten maximal verbunden sind. Das bedeutet, dass jeder Abschnitt vier Nachbarn besitzt (soweit als möglich) und für jeden Zug alle Routen berechnet und gespeichert werden.

Beispiel

Die Tabelle 4.14 zeigt abschließend zu den Routen ein Beispiel für alle Routen für einen bestimmten Start- und Zielknoten.

Element-Name	Wert
Route_Source	$0x05$
Route_Sink	$0x03$
Route_List	...

Tabelle 4.14: Beispiel: T_Train_Route

Das Beispiel enthält alle Routen, die beim Knoten $0x05$ beginnen und im Knoten $0x03$ enden. Die Liste der Datenstruktur enthält alle verfügbaren Routen zwischen den beiden Knoten.

4.1.7 Speicherung einer Route

Übersicht

In den beiden vorherigen Kapitel wurden jene Datenstrukturen erläutert, die zum Speichern der Suchergebnisse beim Suchen von bestimmten Routen verwendet werden. Nachdem der Anwender sich für eine Route entschieden hat, muss diese im System gespeichert und die Suchergebnisse können verworfen werden. Dies ermöglicht ein Anpassen bestimmter Parameter (z.B. Fahrtgeschwindigkeit auf einem bestimmten Abschnitt) und den Austausch der Daten mit den Knoten. Um nun die Informationen einer Route zu speichern wird die Datenstruktur TYPE_ROUTE verwendet, die in der Tabelle 4.15 dargestellt ist und im Folgenden beschrieben wird.

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
Route_Source	Byte	B_MIN	1
Route_Sink	Byte	B_MIN	1
Route_Direction	TYPE_DIRECTION	DIR_FORWARD	1
Route_Nodes	TYPE_TRAIN_ NODE_ARRAY	-	762
Route_Started	Boolean	false	1

Tabelle 4.15: Datenstruktur: T_Route

Beschreibung der Elemente

Route_Source Hier wird die Adresse des Startknotens gespeichert.

Route_Sink In diesem Element steht die Adresse des Zielknotens.

Route_Direction In diesem Element wird die Fahrtrichtung des Zugs der Route gespeichert. Eine Route kann vorwärts (`DIR_FORWARD`) oder rückwärts (`DIR_BACKWARD`) befahren werden.

Route_Nodes Dieses Element enthält ein Array (siehe Kapitel 4.1.4), wobei jedes Element des Arrays für einen Abschnitt steht und die Adresse der Abschnitte dem Index des Arrays entspricht. Jedes Element des Arrays enthält die zugspezifischen Daten eines Abschnitts der Route.

Route_Started Dieses Element signalisiert, ob die Fahrt des Zuges vom Start zum Ziel gestartet wurde.

Speicherbedarf

Auch für diese Datenstruktur spielt die Größe des Systems eine Rolle, weil das Array von der Anzahl der Knoten abhängt. Das Maximum wird – wie bereits erwähnt – über die Konstante `MAX_NODES` festgelegt werden. Insgesamt ergibt sich somit ein Speicherbedarf von maximal 766 Byte.
Da für jeden Zug eine eigene Route existieren kann, wird die beschriebene Datenstruktur zu einem Array (`TYPE_ROUTE_ARRAY`) der Größe `MAX_TRAINS` zusammengefasst, was bei der maximal möglichen Anzahl von Zügen zu einem Bedarf von

$$766Byte \cdot 253 = 193.798 \text{ Byte} \approx 189 \text{ KByte} \tag{4.9}$$

führen würde.

Beispiel

Die Tabelle 4.16 zeigt ein Beispiel, in dem eine einzelne Route gespeichert ist.

Element-Name	**Wert**
`Route_Source`	$0x05$
`Route_Sink`	$0x0A$
`Route_Direction`	`DIR_BACKWARD`
`Route_Nodes`	`...`
`Route_Started`	`false`

Tabelle 4.16: Beispiel: TYPE_ROUTE

In diesem Beispiel handelt es sich um eine Route mit dem Startknoten $0x05$, die mit Fahrtrichtung `DIR_BACKWARD` zum Knoten mit der Adresse $0x0A$ mit den entsprechenden Einstellungen in `Route_Nodes` führt. Die Fahrt des Zuges wurde in diesem Fall noch nicht gestartet.

4.1.8 Speicherung der Knotendaten

Übersicht

Die Tabelle 4.17 zeigt eine Übersicht über die (an jedem Controller) gespeicherten Daten, die in der Datenstruktur `T_Node` zusammengefasst sind.

Element-Name	Datentyp	Initialisierung	Speicherbedarf [Byte]
`Node_Address`	`Byte`	`B_MIN`	1
`Node_Running`	`Boolean`	`true`	1
`Node_Used`	`Boolean`	`false`	1
`Node_Train_Data`	`Byte`	`B_MIN`	1
`Node_Train_Conf`	`Byte`	`B_MIN`	1
`Node_Train_ Address`	`Byte`	`B_MIN`	1

Tabelle 4.17: Datenstruktur: T_Node

Beschreibung der Elemente

Node_Address Dieses Element enthält die Adresse des Controllers.

Node_Running Dieses Element enthält ein Flag, das signalisiert, ob die Software läuft und wird für die Statemachine (siehe Kapitel 4.3.2) benötigt.

Node_Used Hier wird gespeichert, ob sich gerade ein Zug am Abschnitt befindet.

Node_Train_Data Dieses Element enthält die Nutzdaten eines Zuges in folgender Form gespeichert werden:

- Bit 1-4: Geschwindigkeit (`SPEED_MIN` $\cdots$ `SPEED_MAX`)
- Bit 5: Beleuchtung (`LIGHT_ON`, `LIGHT_OFF`)
- Bit 6: Fahrtrichtung (`DIR_FORWARD`, `DIR_BACKWARD`)
- Bit 7-8: Nicht verwendet (00)

Die Speicherung der Daten in dieser Form hat zwar den Nachteil der erschwerten Lesbarkeit der Nutzdaten, dem gegenüber stehen aber folgende Vorteile:

- Geringer Speicherbedarf: Pro Zugmaschine wird nur ein Byte für die Speicherung der Nutzdaten im Digitalbetrieb benötigt. Dies stellt gerade für die Speicherung in Mikrocontroller einen großen Vorteil dar, da die Kosten unter Anderem relativ stark von der Speichergröße abhängen und die Controller möglichst günstig gehalten werden sollten.

- Kommunikation und Speicherung: Die Nutzdaten können in einem einzelnen Byte übertragen werden. Somit wird die Länge einer Message relativ kurz gehalten und die Datenübertragung schnell durchgeführt werden. Am Master-PC werden die Daten in diese Form gebracht, über den Kommunikationsring gesendet und können am Controller ohne weitere Behandlung gespeichert werden.
- Datenpakete: Die Daten müssen für den Digitalbetrieb (siehe Kapitel 4.4.1) lediglich an die richtige Position im Paket kopiert werden.

Node_Train_Conf Hier werden die Konfigurationsdaten gespeichert in folgender Form:

- Bit 1: Weichenstellung (`POS_AHEAD`, `POS_TURN`)
- Bit 2: Betriebsmodus (`MODE_DIGITAL`, `MODE_ANALOG`)
- Bit 3: Controller-Status (`STATUS_GO`, `STATUS_STOP`)
- Bit 4-8: Nicht in Verwendung

Node_Train_Address Das letzte Byte der Datenstruktur enthält die Adresse des Zuges, mit dem aktuell eine Kommunikation stattfindet bzw. als letztes stattgefunden hat.

Speicherbedarf

Die Daten innerhalb dieser Struktur bestehen aus jeweils einem Byte und benötigen somit in Summe 6 Byte für Speicherung der Datenstruktur.

Beispiel

Die Tabelle 4.18 enthält ein Beispiel für die Speicherung der Daten am Controller.

Element-Name	**Wert**
`Node_Address`	$0x05$
`Node_Running`	`true`
`Node_Used`	`true`
`Node_Train_Data`	$0x0F$
`Node_Train_Conf`	$0x02$
`Node_Train_ Address`	$0x03$

Tabelle 4.18: Beispiel: T_Node

In diesem Beispiel handelt es sich um den Controller mit der Adresse $0x05$, dessen Software gerade läuft und dessen Abschnitt von einem Zug mit der Adresse $0x03$ befahren wird. Die Nutz- bzw. Konfigurationsdaten enthalten die Werte $0x0F$ bzw. $0x02$.

4.2 Algorithmen

Dieses Kapitel beschäftigt sich mit den wichtigsten Algorithmen zur Berechnung, Verteilung und Erkennung der benötigten Daten, Informationen und Situationen während der Programmlaufzeit.

4.2.1 Ermittlung der möglichen Routen am Master

Übersicht

Gibt ein Anwender einen Start- und einen Endknoten in der Software an, so müssen alle möglichen Routen vom Start zum Ziel ermittelt und dem Anwender angezeigt werden. Dieser kann sich anschließend für eine Route entscheiden, die dann für die Fahrt eines ausgewählten Zugs verwendet wird.
Die Ermittlung aller möglichen Routen wird von einem Algorithmus durchgeführt, der zwei Mal aufgerufen wird – einmal in Fahrtrichtung `DIR_FORWARD` und einmal in Fahrtrichtung `DIR_BACKWARD`. Die Funktionsweise wird anschließend beschrieben, die Auflistung des Codes ist danach zu finden.

Funktionsweise

Bei dem Algorithmus, der alle Routen zu einem gegebenen Start- und Zielknoten findet, handelt es sich um einen rekursiven Algorithmus. Beim Aufruf wird immer der aktuelle Knoten, der Start und das Ziel angegeben. Im ersten Schritt wird in einem Array an der aktuellen Position (Adresse des aktuellen Knotens) der Wert $true$ gesetzt, um zu signalisieren, dass dieser Knoten bereits bei der Suche berücksichtigt wurde. Anschließend wird überprüft, ob der aktuelle Knoten schon das Ziel ist. Ist diese Bedingung erfüllt, so wird die Route gespeichert und die Rekursionsstufe verlassen. Danach wird überprüft, ob in die aktuelle Richtung ein Nachfolgeknoten in Weichenstellung `POS_AHEAD` existiert. Trifft dies zu, so wird die nächste Rekursionsebene des Algorithmus aufgerufen, wobei als aktueller Knoten der soeben ermittelte Nachfolgeknoten gesetzt wird. Dieses Vorgehen darf allerdings nur dann erfolgen, wenn der Nachfolgeknoten nicht der Startknoten ist. Anschließend wird auf die zweite Weichenstellung überprüft und das selbe Vorgehen wie zuvor beschrieben angewendet.

Code-Listing

```
1 function FindRoute (Source : Byte; Sink : Byte)
2    return Boolean is
3       found_forward : Boolean;
4       found_backward : Boolean;
5    begin
6
```

```
7        -- store the ID of the source
8        routes.Route_Source := Source;
9        -- store the ID of the sink
10       routes.Route_Sink   := Sink;
11       -- set the recursive-level to 1
12       counter := 1;
13       -- clear all visited-flags
14       visited := (others => false);
15
16       -- set the direction of the new route to forward
17       new_route.Route_Direction := DIR_FORWARD;
18       found_forward := FindRouteRecursive(Source,
19                                           Source,
20                                           Sink);
21       -- set the recursive-level to 1
22       counter := 1;
23       -- clear all visited-flags
24       visited := (others => false);
25       -- set the direction of the new route to backward
26       new_route.Route_Direction := DIR_BACKWARD;
27       found_backward := FindRouteRecursive(Source,
28                                            Source,
29                                            Sink);
30       return found_forward or found_backward;
31
32    end FindRoute;
```

```
1  function FindRouteRecursive (Position : Byte;
2                               Source : Byte;
3                               Sink : Byte)
4     return Boolean is
5        retValue : Boolean;
6        nodeAhead : Byte;
7        nodeTurn : Byte;
8     begin
9
10       if new_route.Route_Direction = DIR_FORWARD then
11          -- Node Forward Ahead
12          nodeAhead := sms_nodes_array(Position).
13                       Node_Forward.Switch_Ahead;
14          -- Node Forward Turn
```

```
15          nodeTurn :=  sms_nodes_array(Position).
16                       Node_Forward.Switch_Turn;
17       else
18          -- Node Backward Ahead
19          nodeAhead := sms_nodes_array(Position).
20                       Node_Backward.Switch_Ahead;
21          -- Node Backward Turn
22          nodeTurn :=  sms_nodes_array(Position).
23                       Node_Backward.Switch_Turn;
24       end if;
25
26
27       -- Mark the node as visited
28       visited(Integer(Position)) := true;
29
30       -- current position = sink => route complete
31       if Position = Sink then
32          -- set the length of the route
33          new_route.Route_Length := Byte(counter);
34          -- insert new route into route-list of train
35          Insert(new_route);
36          -- clear the visited flag
37          visited(Integer(Position)) := false;
38          -- decrease the recursion-level-counter
39          counter := counter - 1;
40          return true;
41       end if;
42
43       if nodeAhead /= Source AND nodeAhead /= 0 the
44          if visited(Integer(nodeAhead)) = false then
45             -- increase the recursion-level-counter
46             counter := counter + 1;
47             -- set the correct position
48             new_route.Route_Position(Position) := POS_AHEAD
                  ;
49             -- go to the next level
50             retValue := FindRouteRecursive(nodeAhead,
51                         Source,Sink);
52          end if;
53       end if;
54
55       if nodeTurn /= Source AND nodeTurn /= 0 then
56          if visited(Integer(nodeTurn)) = false then
```

```
57              -- increase the recursion-level-counter
58              counter := counter + 1;
59              -- set the correct position
60              new_route.Route_Position(Position) := POS_TURN;
61              -- go to the next level
62              retValue := FindRouteRecursive(nodeTurn,
63                          Source,Sink);
64           end if;
65        end if;
66
67        -- decrease the recursion-level-counter
68        counter := counter - 1;
69        return retValue;
70
71     end FindRouteRecursive;
```

Korrektheit des Algorithmus

1. Sofern eine Route von X (Startknoten) nach Y (Zielknoten) existiert, wird diese gefunden.
2. Es werden alle Routen von X (Startknoten) nach Y (Zielknoten) gefunden.

ad (1): Beweis mittels Widerspruch (Es gibt eine Route, sie wird aber nicht gefunden). Hierfür wird angenommen, dass vom Startknoten aus bereits $n - 1$ Knoten betrachtet (in beiden Fahrtrichtungen und beiden Weichenpositionen) und der Zielknoten noch nicht gefunden wurde. Für den letzten Knoten (n-ten Knoten) gibt es nun die folgenden Möglichkeiten:

- Der Knoten entspricht dem Startknoten: Somit gäbe es keine Route von X nach Y im System, was ein Widerspruch zur Annahme wäre.
- Es existiert kein Folgeknoten mehr: Auch hier gäbe es keine Route von X nach Y, was ebenfalls zu einem Widerspruch zur Annahme führen würde.
- Der Knoten entspricht dem Zielknoten, was ebenfalls zu einem Widerspruch führen würde.

Über diese drei Punkte konnte gezeigt werden, dass – falls die Route existiert – sie auch gefunden wird.

ad (2): Die Funktion `FindRouteRecursive` wird nur in folgenden Fällen beendet:

- Der aktuelle Knoten entspricht dem Zielknoten. Somit wurde eine Route gefunden und es kann eine Rekursionsstufe zurück gewechselt und die Suche nach weiteren Routen fortgesetzt werden. Auf dem aktuellen Pfad gibt es keine weiteren Routen.

- Der aktuelle Knoten entspricht dem Startknoten. Somit wurde ein Kreis gefunden, auf dem sich keine gesuchte Route befindet. Es muss die Rekursionsstufe verlassen und darunter weiter gesucht werden.
- Der aktuelle Knoten enthält die Adresse 0, was bedeutet, dass dieser Knoten im System nicht existiert und das Ende eines Pfades erreicht wurde.

In den folgenden Fällen wird in die nächst höhere Rekursionssufe gewechselt und so lange weitergesucht bis eine der drei zuvor beschriebenen Bedingungen eintritt. Somit sind alle Fälle abgedeckt und es kann davon ausgegangen werden, dass alle möglichen Routen gefunden werden.

- Der aktuelle Knoten entspricht nicht dem Zielknoten und wurde bei der Suche noch nicht behandelt (`visited` an der entsprechenden Adresse = `false`).
- Der aktuelle Knoten entspricht nicht dem Startknoten und wurde bei der Suche noch nicht behandelt (`visited` an der entsprechenden Adresse = `false`).
- Der aktuelle Knoten besitzt eine eindeutige Adresse (ungleich 0) und wurde bei der Suche noch nicht behandelt (`visited` an der entsprechenden Adresse = `false`).

4.2.2 Kurzschlusserkennung und -behandlung an den Knoten

Der zuvor erklärte Algorithmus berechnet alle zur Verfügung stehenden Routen, die von einem Zug befahren werden können. Bevor nun die Verteilung der dafür nötigen Informationen (Nutz- und Konfigurationsdaten) an die Knoten erklärt wird, muss die Kurzschlusserkennung erläutert werden, die für das Befahren von Abschnitten bzw. den Übertritt von einem in einen anderen Abschnitt eine wesentliche Rolle spielt.

Im digitalen Betriebsmodus kann es zu folgender Situation kommen: Ein Zug führt gerade von einem Abschnitt in den nächsten ein, was dazu führt, dass er von beiden Abschnitten mit Strom versorgt wird. Da die digitale Paketkommunikation Pegel von +12 V und -12 V verwendet, kommt es hier zu Kurzschlüssen bzw. eventuell zu Spannungsspitzen, was dazu führt, dass der Decoder keine korrekten Datenpakete empfängt und die Weiterfahrt behindert wird. Um eine derartige Situation zu vermeiden, wird immer nur jener Abschnitt mit Strom versorgt, der gerade vom Zug befahren wird. Alle anderen Abschnitte sind galvanisch getrennt – nähere Informationen zur Funktionsweise können dem Kapitel 4.2.4 entnommen werden.

4.2.3 Vermeidung von Kollisionen

Übersicht

Einer der wichtigsten Aspekte bei der Steuerung einer Vielzahl von Zügen in einem gemeinsamen Eisenbahnnetz, stellt die Vermeidung von Unfällen dar. Um dies zu gewährleisten, wurde folgende Anforderung an das System getroffen:

- Das Eisenbahnnetz unterteilt sich in Abschnitte.
- Es darf sich auf einem Abschnitt nur ein einziger Zug befinden.

Szenario

Diese zwei Punkte sind zwar sehr wichtig für die Vermeidung von Kollisionen, aber unter folgender Situation noch nicht ausreichend:

- Es befinden sich zumindest zwei Züge im Netz.
- Die Züge befinden sich in zwei benachbarten Abschnitten und steuern auf einander zu.
- Beide Züge treffen zur selben Zeit am Übergang zwischen den beiden Abschnitten ein.

Überprüfung

In dem obigen Fall kommt es zu einer Kollision beim Übergang der beiden Streckenabschnitte. Um eine derartige Situation zu vermeiden, muss bereits beim Befahren eines Abschnitts folgendes überprüft werden:

- Ist der nachfolgende Abschnitt bereits von einem Zug belegt?
- Ist der übernächste Abschnitt bereits von einem Zug belegt?

Kann eine dieser beiden Abfragen mit *Ja* beantwortet werden, so muss der Zug gestoppt werden. Eine Beschreibung dieser Funktionalität kann dem nächsten Kapitel (4.2.4) entnommen werden.

4.2.4 Verteilen der Routeninformationen

Übersicht

Nachdem die Routen wie bereits erwähnt, ermittelt und vom Anwender ausgewählt wurden, muss im nächsten Schritt jedem Knoten mitgeteilt werden, welche Daten er einem Zug in seinem Abschnitt bereitstellen muss. Die folgenden Voraussetzungen müssen für den einwandfreien Betrieb gegeben sein:

- Es wurden alle Routen ermittelt und eine Route vom Anwender für eine bestimmte Zug-Adresse gewählt.
- Die Routendaten wurden in der entsprechenden Datenstruktur gespeichert.
- Bei der Initialisierung der Knoten wurde der Controller-Status auf `STATUS_STOP` gesetzt und somit die Spannung von den Gleisen genommen.
- Der gewählte Zug wurde am Startknoten platziert bzw. es wurde jener Knoten als Startknoten ausgewählt, in dessen Abschnitt sich der Zug gerade befindet.
- Es wurde der `[S T A R T]`-Knopf betätigt.

Belegt-Flag

Für den Austausch von Daten zwischen dem Master und den Knoten ist es erforderlich, dass der Master weiß, auf welchen Abschnitten sich gerade Züge befinden. Diese Information ist aus zwei Gründen wichtig:

- Über einen Abschnitt, der gerade von einem Zug befahren wird, kann über die entsprechenden Datenstrukturen ermittelt werden, welcher der nächste Abschnitt der Route ist und die benötigten Nutz- und Konfigurationsdaten können an den entsprechenden Knoten geschickt werden.

- Es kann vermieden werden, dass zwei Züge gleichzeitig in einen Abschnitt einfahren.

Um festzustellen, ob sich ein Zug am Abschnitt befindet, wird das Errorflag (EF) des TLEs verwendet, das unter den folgenden Bedingungen einen Fehler (EF=0) liefert:

- Kurzschluss, Überlast, zu hohe Temperatur

- Kein Verbraucher vorhanden

Der zweite Punkt der obigen Aufzählung führt nur dann zu einem Fehler (EF=0), wenn beide Eingänge des TLEs (IN1 und IN2) auf 1 gesetzt wurden. Das Flag wird auf 1 zurückgesetzt, sobald sich die Ausgänge ändern. Um nun festzustellen, ob ein Verbraucher vorhanden ist, wird der Zustand des Errorflags für die beiden Betriebsmodi überprüft:

- Digitalbetrieb: Nachdem eine bestimmte Anzahl von Paketen (zu jeweils 40 Bits) vollständig an den Decoder übermittelt wurden, werden die beiden Eingänge des TLEs auf 1 gesetzt, anschließend mindestens $50\mu s$ gewartet und dann der Zustand des Errorflags überprüft.

- Analogbetrieb: Da hier keine Datenpakete übertragen werden, muss ein anderer Zeitpunkt für die Überprüfung gewählt werden. Die Überprüfung wird gestartet, nachdem insgesamt 100 Impulse am Ausgang des TLEs erzeugt wurden. Es werden wieder beide Eingänge auf 1 gesetzt und $50\mu s$ gewartet, bis der Zustand des Flags überprüft werden kann.

Die Zeitverzögerung von $50\mu s$ ergibt sich aus dem Aufbau des TLEs. Weitere Informationen können dem Datenblatt [3] entnommen werden.
Das Ergebnis der Überprüfung wird nun im Belegt-Flag des Controllers gespeichert und der Datenaustausch zwischen Knoten und Zügen wieder bis zur nächsten Überprüfung normal ausgeführt.

Änderung des Belegt-Flags

Wenn das Belegt-Flag des Controllers bei einer Überprüfung seinen Wert ändert, muss entsprechend darauf reagiert werden:

- Änderung von 1 auf 0: Der aktuelle Abschnitt wurde verlassen. Somit gibt es keinen Verbraucher mehr und der Controllerstatus wird auf `STATUS_STOP` gesetzt. Somit ist die Versorgung der Schienen getrennt und es kann kein Strom mehr fließen. Zusätzlich muss dem Master mitgeteilt werden, dass es zu einer Änderung des Status gekommen ist. Um dies durchzuführen, sendet jener Knoten, der nicht belegt ist, eine Busy-Nachricht (siehe Kapitel 4.1.1) an den Master.

- Änderung von 0 auf 1: Der aktuelle Abschnitt wurde befahren. Es müssen keine weitere Aktionen durchgeführt werden und der Betrieb im jeweiligen Modus (digital oder analog) kann fortgesetzt werden.

Vorgehensweise am Master

Für die Beschreibung der Vorgehensweise wird folgende Konvention für die Knotenadresse verwendet:

- *X*: Knoten, der befahren wurde (aktueller Knoten)

- *X-1*: Knoten, der verlassen wurde (Vorgänger in der gewählten Route)

- *X+1*: Nächster Knoten (Nachfolger in der gewählten Route)

- *X+2*: Übernächster Knoten der gewählten Route

Die folgenden Schritte werden nun beim Verteilen der Routeninformationen am Master ausgeführt.

- Wird eine Route gestartet (siehe Bedingungen in der Übersicht), so müssen den Abschnitten die nötigen Informationen (Nutz- und Konfigurationsdaten) mitgeteilt werden. Im ersten Schritt werden die Daten für den Startknoten aus den gespeicherten Routeninformationen ermittelt und versendet. Der Startknoten kann nun mit dem Betrieb beginnen.

- Der zweite Schritt besteht darin, auf die Änderung des Belegt-Flags der einzelnen Abschnitte zu reagieren. Dies erfolgt über einen Task, der die ankommenden Nachrichten analysiert. Trifft eine Busy-Nachricht am Master ein, so enthält diese als Absenderadresse, jenen Knoten, dessen Flag auf 0 gesetzt wurde, der also vom Zug verlassen wurde (Knoten *X-1*). Für diesen Knoten wird in der Datenstruktur am Master das Busy-Flag auf `false` gesetzt. Über die Adresse *X-1* kann über die gespeicherten Routeninformationen der aktuelle Knoten *X* ermittelt werden, dessen Flag auf `true` gesetzt wird.

- Um einen sicheren Betriebszustand zu wahren, muss zu diesem Zeitpunkt ermittelt werden, ob ein Zug weiterfahren darf, oder aufgrund einer möglichen Kollision zum Stillstand gebracht wird. Um diesen Schritt durchzuführen, ist es erforderlich, dass der Zustand es Busy-Flags der Knoten *X+1* und *X+2* ermittelt wird. Die folgenden Situationen können dabei auftreten:

- Beide Knoten sind frei: Der Master erzeugt eine Nachricht mit Nutz- und Konfigurationsdaten und sendet diese an den entsprechenden Knoten. Dieser speichert die Daten und lässt den Zug mit der gegebenen Geschwindigkeit, Fahrtrichtung, Beleuchtung und Weichenstellung im entsprechenden Betrieb (digital oder analog) fahren.
- Mindestens ein Knoten ist belegt. Wie auch in der zuvor beschriebenen Situation wird eine Nachricht erzeugt, allerdings wird der Wert für die Geschwindigkeit auf $0x00$ gesetzt, um den Zug anhalten zu lassen. Diese Nachricht wird an den entsprechenden Knoten gesendet. Zusätzlich wird eine zweite Nachricht mit der korrekten Geschwindigkeit und den Adressen der beiden Nachfolgeknoten als Listenelement am Ende der sogenannten *Blocking-List* abgelegt. Diese Liste beinhaltet alle Nachrichten, welche die Knoten erhalten würden, wenn sie ihren Betrieb normal (wie im zuvor beschriebenen Punkt) ausführen könnten. Ein zweiter Task läuft in einer Endlosschleife und untersucht jedes Element der Liste, ob die beiden gespeicherten Knoten *X+1* und *X+2* frei sind. Ist diese Überprüfung positiv, so wird die gespeicherte Nachricht aus der Liste gelöscht, an den Knoten gesendet und dieser kann den Betrieb mit den korrekten Parametern ausführen. Andernfalls verbleibt die Nachricht in der Liste und wird beim nächsten Durchlauf wieder überprüft.

- Wenn ein Abschnitt von einem Zug befahren wird, muss in den Routeninformationen ermittelt werden, ob das Ziel bereits erreicht wurde. Ist dies der Fall, so wird – so wie bei belegten Knoten – eine Nachricht mit Nutz- und Konfigurationsdaten an den Knoten gesendet, ebenfalls mit einem Geschwindigkeitswert von $0x00$. Da der Zug sein Fahrtziel erreicht hat, wird auch die Beleuchtung deaktiviert.

4.2.5 Ausfallserkennung

Übersicht

Ein wesentlicher Faktor bei sicherheitsrelevanten Systemen ist die Erkennung von Ausfällen. Bei der Steuerung von Modelleisenbahnen mit dem entworfenen System gibt es vier Teile des Systems, die ausfallen könnten:

- Sollte es einen Defekt bei den Zugmaschinen geben, so könnten diese ungewollt auf einem Abschnitt stehen bleiben und somit eine Gefahr für andere Züge darstellen, die diesen Abschnitt befahren wollen.

- Der Master-PC kann aufgrund diverser Probleme ausfallen. Diese könnten eine fehlerhafte Software oder ein Defekt in der PC-Hardware sein, der zum Absturz des Rechners, aber auch zu Problemen bei der Datenübertragung führen kann.

- Auch die Controller können aufgrund eines Softwarefehlers oder defekter Hardwarekomponenten ausfallen oder zu unerwünschten Betriebszuständen führen

- Es können Probleme bei der Kommunikation aufgrund defekter Übertragungsmedien auftreten. Als Folge defekter Kabel könnten Daten gänzlich verloren gehen, Pakete unvollständig oder fehlerhaft übermittelt werden.

Ausfall eines Zuges

Um während des Betriebs trotz eines Ausfalls nicht in einen unsicheren Betriebszustand zu kommen, muss während der Laufzeit der Software auf mögliche Fehler geprüft werden. Der einfachste zu behandelnde Fall wäre eine defekte Zugmaschine, die in einem Abschnitt hängen bleibt, weil die Software automatisch dafür sorgt, dass pro Abschnitt maximal ein Zug vorhanden sein darf. Da der Master weiß, welche Abschnitte von Zügen belegt sind, kann dieser verhindern, dass weitere Züge in diesen Abschnitt einfahren. In diesem Fall ist es für den Master nicht relevant, ob es sich hierbei um einen defekten oder funktionstüchtigen Zug handelt.

Ausfall des Masters, der Knoten oder der Übertragungsinfrastruktur

Kommt es während des Betriebs zu einem Ausfall des Masters, eines oder mehrerer Knoten oder auch der Medien zur Übertragung der Daten, so kann nicht mehr sichergestellt werden, dass die nötigen Informationen zur Gewährleistung eines sicheren Zustands an den entsprechenden Empfängern (Knoten oder Master) korrekt und vollständig eintreffen. Da in diesem Fall nur schwer bis gar nicht ermittelt werden kann, ob ein Soft- oder Hardwarefehler vorherrscht, wird sofort in einen sicheren Zustand gewechselt, in diesem Fall in den Zustand `ST_RUNNING_ERROR` am Master (siehe Kapitel 4.3.1) bzw. `ERROR` an den Knoten (siehe Kapitel 4.3.2). Es wird der Betrieb sofort gestoppt, das bedeutet alle Züge werden angehalten, indem die Versorgung der Gleise unterbrochen wird. Somit fließt kein Strom und alle Züge kommen zum Stillstand. Ein Fehler, der zu einem solchen Zustand führt, wird bei der Übertragung der Daten festgestellt, wenn die Checksumme der empfangenen Daten fehlerhaft ist, ein Datenpaket unvollständig übermittelt wird oder über einen längeren Zeitraum keine Kommunikation zwischen Master und den Knoten stattgefunden hat. Um die fehlerfreie Kommunikation zu testen, schickt der Master in gewissen Zeitabständen eine Nachricht auf den Ring. Besteht kein Fehler im System, so wird diese Nachricht weitergeleitet und trifft nach einer bestimmten Zeit wieder am Master korrekt ein. Ist dies nicht der Fall, so wird von einem Fehler im System ausgegangen und der Master wechselt in den Fehlerzustand. Zusätzlich sendet er eine Stop-Nachricht (der Nachrichtentyp ist `MSG_TYPE_STOP`) an den Ring. Diese Vorgehensweise hat den Vorteil, dass ein Knoten, der diese Nachricht erhält, sofort in den Fehlerzustand übergehen kann und nicht selbst auf den Ablauf eines Intervalls warten muss. Erhält ein Knoten diese Nachricht nicht vom Master und auch keine Nachricht eines anderen Typs oder unkorrekte Nachrichten, so wird in den Fehlerzustand gewechselt und eine Stop-Nachricht an den Ring gesendet.

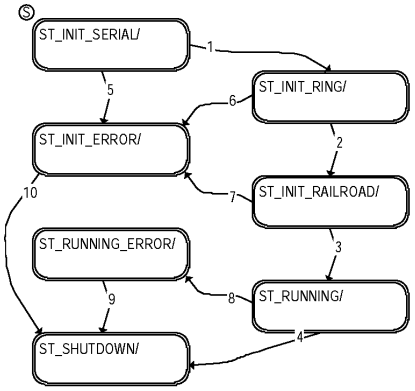

Abbildung 4.3: Statemachine (Master)

4.3 Statemachine

4.3.1 Statemachine am Master

Übersicht

Die Abbildung 4.3 zeigt eine Übersicht über die im System vorhandenen Zustände und Transitions, die im Folgenden erklärt werden. Screenshots und weitere Informationen können dem Kapitel 4.5 entnommen werden.

Initialisierung

Im ersten Schritt nach dem Programmstart versucht der Master die Log-Datei, die für die Protokollierung verwendet wird, zu öffnen. Ist dieser Versuch nicht erfolgreich, so wird die Ausgabe in die Datei deaktiviert und lediglich über die Konsole protokolliert. Anschließend erfolgt die Initialisierung des Masters, die in drei Schritten abläuft:

- Initialisierung der seriellen Schnittstelle (`ST_INIT_SERIAL`, Startzustand)
- Initialisierung des Kommunikationsrings (`ST_INIT_RING`)
- Initialisierung des Schienensystems (`ST_INIT_RAILROAD`)

Jeder dieser Schritte erfolgt nach demselben Schema: Für jeden Zustand gibt es eine eigene Statemachine, die sich aber nur in einem einzigen Zustand unterscheiden (zu sehen in den Abbildungen 4.4 - 4.6). Im Startzustand (`STATE_RUNNING`) wird ein Timer gestartet. Bis zum Ablauf dieses Timers muss nun die aktuelle Aufgabe abgeschlossen sein, dann wird in den Zustand `STATE_FINISHED` gewechselt. Läuft der Timer ab, bevor die Aktion abgeschlossen wurde, so wird in den Fehlerzustand `STATE_ERROR_TIMEOUT` gewechselt. Für jeden der drei Initialisierungszustände gibt es innerhalb der Aufgabe noch einen weiteren Fehlerzustand in den – falls es zu einem Fehler innerhalb der Abarbeitung der Aufgabe kommt (z.B. maximale Anzahl der Versuche wurde erreicht) – gewechselt wird (z.B. `STATE_ERROR_FILE`).

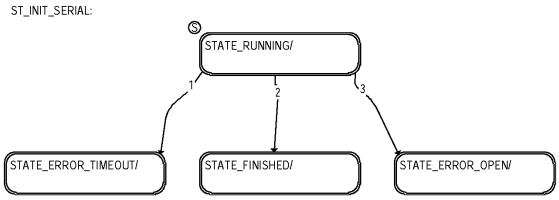

Abbildung 4.4: Initialisierung der seriellen Schnittstelle (Statemachine)

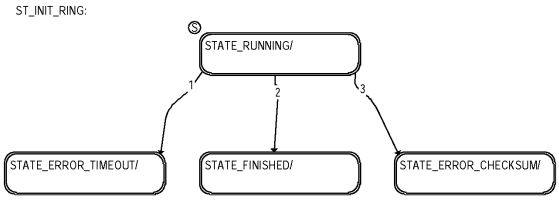

Abbildung 4.5: Initialisierung der Kommunikationsrings (Statemachine)

Die Zeitspanne des jeweiligen Timers kann über die Optionen eingestellt werden. Weiters kann noch die Anzahl der Wiederholungen für einen Initialisierungsschritt eingestellt werden. Ist keiner der eingestellten Versuche erfolgreich, so gilt die Initialisierung als gescheitert, es wird in den entsprechenden Fehlerzustand gewechselt und die Software beendet.
Ist ein Schritt erfolgreich abgeschlossen, so wird in den Folgezustand gewechselt und der Initialisierungsdialog entsprechend aktualisiert.

Initialisierung der seriellen Schnittstelle Im ersten Schritt der Initialisierung wird die serielle Schnittstelle, die über das Package `Gnat.Serial_Communications` angesprochen wird, geöffnet. Über die Optionen kann das Port eingestellt werden. Die Abbildung 4.4 zeigt die Statemachine für diesen Zustand.

Initialisierung des Kommunikationsrings Für die Kommunikation zwischen dem Master und den Knoten ist es erforderlich, dass jedem Knoten eine eindeutige Adresse zugeordnet ist – dies erfolgt im zweiten Initialisierungsschritt. Die Statemachine kann der Abbildung 4.5 entnommen werden.
Der Master sendet eine Initialisierungs-Nachricht über die serielle Schnittstelle an den Ring, diese wird vom ersten Knoten empfangen. Im Datenfeld wird der Wert 0 vom Master eingetragen. Der Knoten inkrementiert diesen Wert, speichert ihn als seine eigene Adresse und schickt die Nachricht mit dem neuen Wert am Ring weiter. Dies wird von jedem Knoten durchgeführt, bis die Nachricht letztendlich wieder am Master empfangen wird. Im Datenfeld ist jetzt die Adresse des letzten Knoten enthalten, was der Anzahl der Ringteilnehmer entspricht.

Initialisierung des Schienesystems Im vorherigen Schritt wurde der Kommunikationsring für die Verwendung im Betrieb initialisiert, jedoch sagt dieser noch nichts über das physikalische

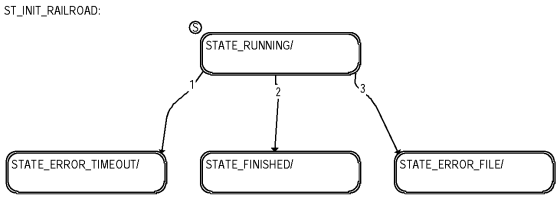

Abbildung 4.6: Initialisierung des Schienensystems (Statemachine)

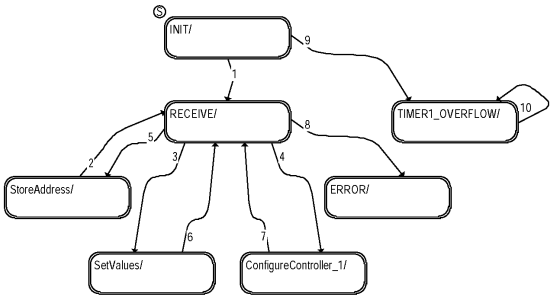

Abbildung 4.7: Statemachine am Controller

Eisenbahnnetz aus (Statemachine: siehe Abbildung 4.6). Um festzustellen, wie die Knoten miteinander verbunden sind, wird diese Information über eine Textdatei eingelesen (für weitere Informationen siehe Kapitel 4.5). Für jeden Knoten kann es vier Nachbarn geben (vorwärts mit gerader Weichenstellung, vorwärts mit gestellter Weiche, rückwärts mit gerader Weichenstellung, rückwärts mit gestellter Weiche), die – falls vorhanden – zur Knotenadresse eingetragen sind.
In diesem Initialisierungsschritt wird die Textdatei (einstellbar über die Optionen) geladen und allen Knoten ihre Nachbarn zugeordnet und in der entsprechenden Datenstruktur am Master gespeichert.

Hiermit ist die Initialisierung am Master abgeschlossen und der normale Betrieb kann gestartet werden.

4.3.2 Statemachine an den Knoten

Übersicht

Die Abbildung 4.7 zeigt die Statemachine am Controller. Im Folgenden werden die einzelnen Zustände nun beschrieben.

Initialisierung (INIT)

UART Die Initialisierung des Controllers beginnt beim UART. Hierfür wird ein vordefiniertes Package verwendet und es muss lediglich die Baudrate eingestellt werden. Diese berechnet sich für den asynchronen Modus über die Formel [5]

$$BAUD = \frac{f_{OSC}}{16(UBRR + 1)} \tag{4.10}$$

was bei einer Baudrate von BAUD = 9600 bps, einer Taktfrequenz des Prozessors von f_{OSC} = 3.686.390 Hz und der entsprechenden Umformung zu folgender Rechnung führt:

$$UBRR = \frac{3.686.390}{153.600} - 1 \approx 23 \tag{4.11}$$

Im ersten Schritt wird das Datenregister des UART gelöscht, um zu vermeiden, dass undefinierte Daten versendet oder bearbeitet werden.

LEDs Der nächste Teil der Initialisierung besteht darin, die LEDs für den Betrieb einzustellen. Dabei wird die Datenrichtung festgelegt und anschließend sichergestellt, dass alle vier LEDs ausgeschaltet sind. Die LEDs haben im Betrieb folgende Bedeutung:

- LED1: Diese LED leuchtet, wenn sich ein Zug am Abschnitt befindet.
- LED2: Während der Datenübertragung (sowohl beim Senden als auch beim Empfangen) leuchtet diese LED.
- LED3: Eine leuchtende LED3 signalisiert den digitalen Betriebsmodus, andernfalls befindet sich der Controller im analogen Betrieb.
- LED4: Die vierte LED wird im Fehler-/Notfall aktiviert. In diesem Fall wurde der Betrieb sofort unterbrochen und ein *Emergency stop (NOT AUS)* durchgeführt.

Die Tabelle 4.19 zeigt die Konfiguration der Ein- und Ausgangspins.

Pin	Zustand	Beschreibung
DDC1	`true`	Ausgangspin LED4
DDC2	`true`	Ausgangspin LED3
DDC3	`true`	Ausgangspin LED2
DDC4	`true`	Ausgangspin LED1

Tabelle 4.19: Initialisierung der LEDs

Exkurs: Emergency stop Ein *Emergency stop* ist der sofortige Betriebsstopp, das heißt, die Versorgung der Gleise wird unterbrochen. Ein derartiger Stopp kann vom Anwender über die Software am Master ausgelöst werden, woraufhin eine Nachricht vom Type `MSG_TYPE_STOP` an die Controller geschickt wird. Beim Erhalt der Nachricht wird in den Zustand *ERROR* gewechselt (siehe Kapitel 4.3.2) und die Nachricht am Ring weitergeleitet. Läuft der sogenannte *Alive-Timer* (siehe Kapitel 4.2.5) ab, so wurde über ein definiertes Zeitintervall keine Nachricht am Controller empfangen. Dies deutet auf einen Fehler (physikalisches Kommunikationsmedium, Absturz des Masters) hin. Dadurch kann ein sicherer Betrieb nicht mehr gewährleistet werden und alle Züge müssen den Betrieb stoppen.

Pin	Zustand	Beschreibung
DDD3	`false`	IN1 des TLE
DDD4	`false`	IN2 des TLE
DDD5	`false`	VSS des TLE
DDD2	`false`	Errorflag des TLE
PORTD2	`true`	Pull-Up für DDD2

Tabelle 4.20: Initialisierung der Schienenansteuerung

Pin	Zustand	Beschreibung
DDB0	`true`	IN1 des TLE
DDB2	`true`	IN2 des TLE
DDB1	`true`	Errorflag des TLE
PORTB1	`true`	Pull-Up für DDB1

Tabelle 4.21: Initialisierung der Weichenansteuerung

Schienen- und Weichensteuerung Um die TLEs für die Schienen- und Weichensteuerung verwenden zu können, müssen die Ein- und Ausgangspins entsprechend den Tabellen 4.20 und 4.21 konfiguriert werden.

Betriebsmodus Der Betriebsmodus wird bei der Initialisierung auf Digitalbetrieb gestellt. Hierfür wird die LED3 aktiviert, der Timer1 für die Steuerung der digitalen Datenpakete für die Decoder konfiguriert und anschließend gestartet. Eine Beschreibung des digitalen Betriebsmodus kann im Kapitel 4.4.1 gefunden werden. Hier wird auch die Berechnung der Timer-Werte erklärt. Somit ist die Initialisierung des Controllers abgeschlossen und der normale Betrieb wird gestartet.

RECEIVE

Dieser Zustand kann als zentrales Element der Statemachine betrachtet werden, in dem die Nachrichten über die serielle Schnittstelle empfangen werden. Wurde eine Nachricht korrekt (Checksumme der Nachricht und berechnete Checksumme sind gleich) empfangen, wird – abhängig vom Typ der Nachricht – in einen der möglichen Folgezustände gewechselt. Die Tabelle 4.22 zeigt eine Übersicht über die Folgezustände und unter welcher Bedingung in diese gewechselt wird. Nach Abarbeitung der Funktionen in den Folgezuständen, wird – ausgenommen vom Zustand *ERROR* – wieder zurück in den Zustand *RECEIVE* gewechselt.

StoreAddress

Dieser Zustand wird dann erreicht, wenn eine Initialisierungsnachricht empfangen wurde. Es wird das Datenfeld der Nachricht (`MSG_Data`) gelesen und um den Wert 1 inkrementiert. Die-

Zustand	Übergangsbedingung
StoreAddress	Nachrichtentyp = `MSG_TYPE_INIT` (Initialisierungsnachricht)
SetValues	Nachrichtentyp = `MSG_TYPE_DATA` (Nutzdatennachricht)
ERROR	Nachrichtentyp = `MSG_TYPE_STOP` (Abbruchsnachricht)

Tabelle 4.22: Folgezustände mit Übergangsgbedingungen

ser Wert wird als eigene Adresse (Controlleradresse) gespeichert und die Nachricht mit dem modifizierten Wert weitergeleitet.

SetValues

Erhält ein Controller eine Nutzdatennachricht, so wird in diesen Zustand gewechselt. Der erste Schritt besteht darin, zu prüfen, ob die empfangene Nachricht für den Controller bestimmt ist. Dabei können folgende Fälle unterschieden werden:

- Die Empfängeradresse der Nachricht entspricht der Controlleradresse: Die empfangenen Daten (Nutz- und Konfigurationsdaten) werden gespeichert.
- Die Empfängeradresse hat den Wert $0xFF$ (es handelt sich um eine Broadcastnachricht): Die Daten werden gespeichert und die Nachricht anschließend weitergeleitet.
- Die Empfängeradresse hat keinen der beiden zuvor beschriebenen Werte: Die Nachricht wird weitergeleitet.

ERROR

Dieser Zustand wird nach dem Empfang einer Abbruchsnachricht erreicht. Diese Nachricht wird im ersten Schritt weitergeleitet und im zweiten wird die Software beendet.

TIMER1_OVERFLOW

In diesen Zustand wird beim Ablauf des Timer1 gewechselt. Hier wird entweder die Funktion zur Motorsteuerung im Analog- oder jene zur Steuerung im Digitalbetrieb aufgerufen. Nach Abarbeitung dieser Funktion wird der normale Betrieb im vorherigen Zustand fortgesetzt. Die Beschreibung der beiden Betriebsmodi sind unter Kapitel 4.4.1 (Digitalbetrieb) bzw. 4.4.2 (Analogbetrieb) zu finden.

4.4 Betriebsmodi

4.4.1 Digitalbetrieb

Im Digitalbetrieb kommuniziert der Controller über die Schienen mit dem Digitaldecoder der Lokomotive. Dieser wandelt die empfangenen Daten um und ändert die Geschwindigkeit, den

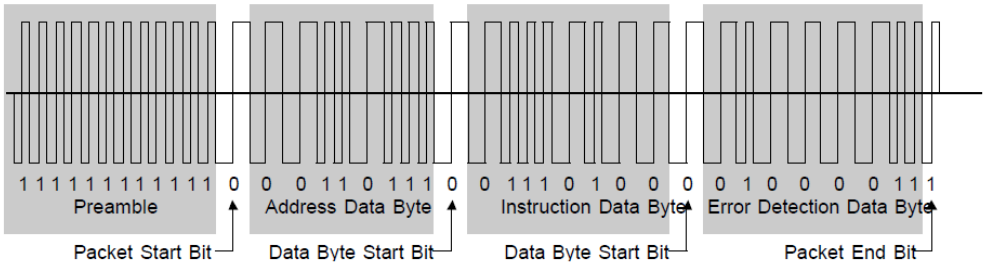

Abbildung 4.8: Digitales Paket [6]

Scheinwerfer und die Fahrtrichtung entsprechend. Beim Kommunikationsprotokoll handelt es sich um den NMRA Standard S-9.2 [6], die Elektronik unterliegt dem NMRA Standard S-9.1 [7].

Datenpaket

Ein Datenpaket besteht aus 40 Bit, wobei es hier folgende Aufteilung gibt (siehe Abbildung 4.8) [6]:

- Die ersten zwölf Bit enthalten die Präambel, wobei hier jedes Bit auf 1 gesetzt ist, gefolgt vom Startbit mit dem Wert 0
- Die nächsten 8 Bit enthalten die Adresse des digitalen Decoders (Adresse der Zugmaschine), mit einem weiteren Startbit (0). Die Adresse 0x00 und 0xFF sind für bestimmte Datenpakete (Idle, Broadcast) reserviert und dürfen somit nicht verwendet werden. Somit ergibt sich ein maximale Decoderanzahl von 254.
- Bit 23 bis Bit 30 enthalten die Fahrdaten, wobei die ersten beiden Bit (01) dafür stehen, dass nachfolgend ein Bit für die Fahrtrichtung, eines für die Beleuchtung und vier Bit für die Geschwindigkeit folgen, gefolgt von einem Startbit (0)
- Schlussendlich folgen acht Bit, welche die Checksumme (Exklusives Oder der Adresse und der Fahrdaten) enthalten. Abgeschlossen wird das Paket mit einem Stopbit (1)

Beispiel

Als Beispiel werden folgende Werte angenommen:

- Decoderadresse: 3
- Fahrgeschwindigkeit: 10
- Fahrtrichtung: `DIR_FORWARD` (0)
- Beleuchtung: `LIGHT_ON` (1)

Aus den gegebenen Daten ergibt sich ein Datenpaket mit folgenden Werten:

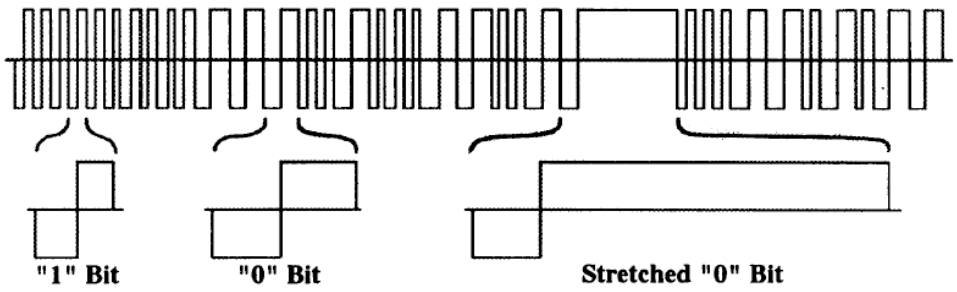

Abbildung 4.9: Bit-Codierung [7]

- Byte 1: $255_{10} = 11111111_2$
- Byte 2: $240_{10} = 11110000_2$
- Byte 3: $025_{10} = 00011001_2$
- Byte 4: $104_{10} = 01101000_2$
- Byte 5: $179_{10} = 10110011_2$

Kodierung

Jedes zu übertragende Bit wird laut NMRA-Standard [7] so modelliert, dass für den Decoder folgende Kriterien gelten (siehe Abbildung 4.9), wobei $|A|$ und $|B|$ für die Dauer eines negativen (-12 V) bzw. positiven (+12 V) Impulses stehen:

- $52\mu s \leq |A| \leq 64\mu s$
- $|A| - |B| \leq 6\mu s$

1-Bit Laut [7] müssen die folgenden Kriterien für die Kodierung des Bitwerts 1 gelten:

- Der erste und zweite Teil des kodierten Bits sollten dieselbe Länge haben ($|A| = |B|$)
- Die Dauer jedes Abschnitts sollte $58\mu s$ betragen, was eine Gesamtperiodendauer von $116\mu s$ ergibt

0-Bit Laut [7] müssen die folgenden Kriterien für die Kodierung des Bitwerts 0 gelten:

- Der erste und zweite Teil des kodierten Bits müssen nicht dieselbe Länge haben, der zweite Teil kann in der Länge variiert werden um den Gleichanteil des Signals zu verändern, damit auch gleichzeitig analoge Zugmaschinen betrieben werden können (siehe Abbildung 4.9).
- Die Dauer jedes Abschnitts sollte mindestens $100\mu s$ betragen, was eine Gesamtperiodendauer von mindestens $200\mu s$ ergibt

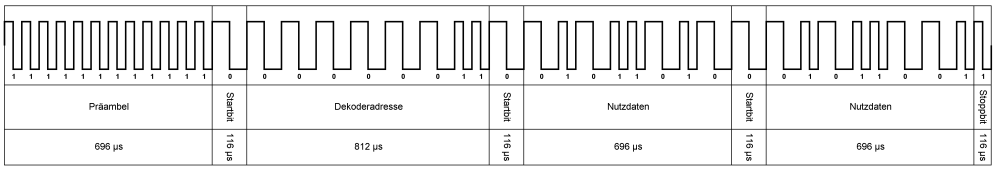

Abbildung 4.10: Digitales Paket (Beispiel)

Berechnung der optimalen Timer-Werte

Um eine Zeit von $\tau = 58\mu s$ pro Pegel für den Bit-Zustand 1 zu erreichen, wird in der Software ein Timer mit dem Wert 65410 gestartet. Mit einer Prozessorfrequenz (über externe Quarz) von $3.6869MHz$ kann dieser Werte über folgende Berechnung ermittelt werden

$$f = 3.6869MHz = 3686900Hz = 3686900s^{-1} \tag{4.12}$$

$$T = \frac{1}{3686900s^{-1}} = 2.7 \cdot 10^{-7}s = 0.27\mu s \tag{4.13}$$

$$C = 65535 - \frac{34\mu s}{0.27\mu s} = 65535 - 125 = 65410 \tag{4.14}$$

In der Gleichung 4.14 wurde der Wert $34\mu s$ für den Timer angenommen, was bedeutet, dass $34\mu s$ nach dem Aktivieren des Timers die Interrupt-Service-Routine aufgerufen wird. Da in dieser Routine im ersten Schritt Bedingungen geprüft und Berechnungen durchgeführt werden, kann der Timer erst danach neu gesetzt werden was zu einer Verzögerung von ca. $34\mu s$ führt. Somit ergibt sich eine Gesamtzeit von $58\mu s$, was den Vorgaben des Protokolls entspricht. Für den Bit-Wert 0 wurde der Timer-Wert $65285 = 65535 - 2 \cdot 125$ gewählt.

4.4.2 Analogbetrieb

Der Analogbetrieb funktioniert über eine Pulsweitenmodulation, mit der die Geschwindigkeit eines Zuges variiert werden kann. Die Ansteuerung von Komponenten wie Licht oder Signalgeräten eines Zuges steht nicht zur Verfügung, da keine Kommunikation mit dem digitalen Decoder des Zuges erfolgt, sondern lediglich eine Spannung von -12V und 0V für eine Fahrtrichtung oder +12V und 0V für die andere Fahrtrichtung an den Schienen angelegt wird und der Decoder somit kein digitales Decodersignal erkennt. Über die Modulation kann eine feinere Abstufung der Geschwindigkeitswerte erfolgen als beim digitalen Betriebsmodus, bei dem insgesamt nur 16 Werte (0 bis 15) für die Geschwindigkeit möglich sind. Um eine einheitliche Steuerung sowohl für den digitalen als auch für den analogen Betrieb zu gewährleisten, wurde allerdings die Einschränkung auf die Geschwindigkeitsstufen 0 bis 15 getroffen.

Der allgemeine Ablauf zur Steuerung von Zügen, die Informationen über Routen und den Austausch der benötigten Daten kann im Kapitel 4.2.4 gefunden werden. Die beschriebenen Algorithmen werden in beiden Betriebsmodi verwendet, der einzige Unterschied besteht in der Modellierung der Spannung an den Ausgängen des TLE für die Motorsteuerung.

4.5 User Interface

Dieser Abschnitt beschäftigt sich mit dem Aufbau und der Funktionsweise des User Interface. Für die Entwicklung diente der User-Interface-Editor *Glade*, basierend auf der *GTK+* Umgebung. Die Gründe für diese Wahl waren

- Ada-Unterstützung
- Bei Glade und GTK+ handelt es sich um Open-Source-Software, wodurch keine Kosten für die Entwicklung des User-Interface entstehen.
- Glade ist ein intuitiv bedienbarer Editor, der lediglich zum Erstellen des User Interface dient. Die Funktionalität dahinter erfolgt unabhängig.
- Glade bietet eine große Auswahl an vordefinierten Widgets, die ohne weitere Modifikation verwendet werden können.

4.5.1 Protokollierung

Einen wesentlichen Teil des User Interface stellt die Protokollierung der wichtigsten Schritte im Programmablauf und der gesamten Kommunikation dar. Dabei werden alle Pakete die vom Master gesendet und auf diesem empfangen werden, sowie die wichtigsten Informationen zu den aktuellen Zuständen, in einer Textdatei gespeichert. Sollte es während des Betriebs zu einer Störung, zu einem Defekt oder einem anderen unerwünschten Verhalten kommen, so kann über diese Log-Datei eventuell die Ursache des Fehlers ermittelt werden. Wie bereits im Kapitel 4.3.1 erwähnt, ist das Öffnen dieser Datei der erste Schritt beim Start der Software. Über die Programmeinstellungen (Kapitel 4.5.2) kann das Verzeichnis für die Ablage dieser Datei gewählt werden.

Unter diesem Absatz ist ein Auszug aus einem Log-File zu finden. In diesem Fall wurde die Initialisierung erfolgreich abgeschlossen und der Hauptdialog der Anwendung gestartet. Für den Knoten `<01>` und die Zugadresse `<03>` wurde der Geschwindigkeitswert $0x04$ gespeichert. Die Beleuchtung wurde für diesen Zug deaktiviert. Anschließend wurde eine Message an diesen Knoten gesendet und eine Busy-Nachricht von diesem Knoten empfangen.
Zur Orientierung im Protokoll dienen die Schlüsselwörter `Step` bzw. `Result` die den aktuellen Schritt der Software bzw. das zugehörige Ergebnis anzeigen. Zeilen die mit `TX` oder `RX` beginnen stehen für gesendete bzw. empfangene Nachrichten. Anschließend stehen die sieben Byte der Nachricht in hexadezimalen Werten (Erklärung siehe Kapitel 4.1.1). `Save` zeigt an, dass Daten gespeichert wurden.

```
+-----------------------------------------------------------+
|  S M S - Sicherheitskritische Modelleisenbahnsteuerung    |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| Step:   Initialize the serial interface                   |
```

```
| Result: Finished                                          |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| Step:   Initialize the Ring                               |
| TX:     16#00# 16#00# 16#01# 16#00# 16#00# 16#00# 16#01# |
| RX:     16#00# 16#00# 16#01# 16#00# 16#02# 16#00# 16#03# |
| Result: Finished                                          |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| Step:   Initialize the railroad system                    |
| Result: Finished                                          |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| Step:   Start the railroad control software               |
| Save:   ID Node=<01> ID Train=<03> Speed=04 Light=OFF     |
| TX:     16#00# 16#01# 16#03# 16#03# 16#04# 16#04# 16#01# |
| RX:     16#01# 16#00# 16#04# 16#03# 16#00# 16#00# 16#06# |
```

4.5.2 Dialoge

Die Software besteht aus vier Dialogen, die dem Anwender zur Verfügung stehen:

- Dialog beim Starten der Software zur Visualisierung des aktuellen Initialisierungs-Fortschritts
- Hauptprogramm-Dialog zur Verwaltung von Zügen und Routen
- Dialog zum Hinzufügen neuer Routen
- Dialog zum Anpassen der Programmeinstellungen

Diese Dialoge werden nun in den folgenden Abschnitten im Detail beschrieben. Screenshots zu den einzelnen Kapiteln finden sich in den Grafiken 4.11 bis 4.14.

Initialisierung der Soft- und Hardware

Wie bereits in Kapitel 4.3.1 beschrieben, unterteilt sich die Initialisierung in drei Phasen:

- Initialisierung der seriellen Schnittstelle (Startzustand)
- Initialisierung des Kommunikationsringes
- Initialisierung des Schienensystems

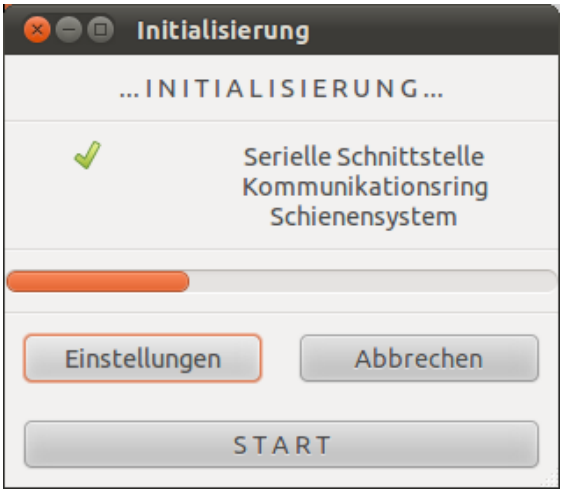

Abbildung 4.11: Dialog während der Initialisierungsphase

Mit dem Programmstart öffnet sich ein Dialog, der den Fortschritt der Initialisierung und das Ergebnis dieser drei Phasen zeigt. Ist der Initialisierungsvorgang erfolgreich, kann der Hauptdialog (nächstes Kapitel) geöffnet werden, im Fehlerfall können die Programmeinstellungen (siehe Kapitel 4.5.2) geändert werden. Folgende Buttons stehen in diesem Dialog zur Verfügung:

- `[Einstellungen]`: Es wird der Dialog zur Anpassung der Programmeinstellungen geöffnet.
- `[Abbrechen]`: Die Initialisierung wird abgebrochen und die Software beendet.
- `[S T A R T]`: Der Hauptdialog der Steuerungssoftware wird gestartet.

Verwaltung und Steuerung von Zügen und Routen

Dieser Dialog ist der Hauptdialog der Software. Er wird gestartet, wenn die Initialisierung abgeschlossen ist und `[S T A R T]` gedrückt wurde. Dieser Dialog verfügt über ein Menü, das die folgenden Einträge enthält:

- Beenden: Die Software wird beendet. Davor wird noch eine Stop-Nachricht (vom Typ `MSG_TYPE_STOP`) an den Ring gesendet um den Knoten zu signalisieren, dass der Betrieb beendet wird und alle Züge gestoppt werden müssen.
- Routen
  - Neue Route hinzufügen: Über diesen Punkt öffnet sich der Dialog zum Hinzufügen neuer Routen.
  - Route löschen: Dieser Menüpunkt dient dazu, die in der Liste markierte Route zu löschen.

Abbildung 4.12: Hauptdialog zur Verwaltung und Steuerung

- Einstellungen: Dieser Menüeintrag öffnet den Dialog zum Anpassen der Programmeinstellungen.
- Info: Es erscheint eine Meldung über die Programmversion
- NOT-AUS: Dieser Menüpunkt dient im Notfall zum Stoppen aller Züge. In diesem Fall wird – wie auch beim Beenden der Software – eine Stop-Nachricht gesendet. Der Unterschied zum Beenden besteht darin, dass die Software am Master nicht beendet wird.

Die Elemente unterhalb des Menüs können in die folgenden Teile aufgeteilt werden:

- Routen: Hier befindet sich eine Combobox, die alle Routen, die derzeit vom System verwendet werden, enthält. In Abbildung 4.12 wurde der Eintrag

  `<03> (FW): <01><02>`

  selektiert, was folgende Bedeutung hat:

  – `<03>`: Adresse des digitalen Decoders
  – `(FW)`: Fahrtrichtung (`DIR_FORWARD`)
  – `<01><02>`: Verlauf der Route von Knoten 01 zum Knoten 02

  Darunter existieren zwei Buttons:

  – `[Neue Route hinzufügen]`

- [Route löschen]

Diese beiden Buttons rufen dieselben Funktionen auf wie die gleichnamigen Menüpunkte.

- Abschnitte: Unter den Routen befindet sich eine weitere Combobox, in der alle Abschnitte der gerade ausgewählten Route, beginnend beim Startknoten bis zum Ziel, enthalten sind.

- Abschnittsdaten: Bei den Abschnittsdaten kann zwischen vier ($0x00$ bis $0x0F$) oder fünf ($0x00$ bis $0x1F$) Bit für die Geschwindigkeitswerte gewählt werden. Für jeden ausgewählten Abschnitt können die folgenden Daten den Bedürfnissen angepasst werden:

  - Geschwindigkeit, mit welcher der gewählte Zug am aktuellen Abschnitt fährt.
  - Status der Beleuchtung (ein- oder ausgeschaltet).

  Die beiden Werte können geändert werden (über die Buttons [<<] bzw. [>>] für die Änderung des Geschwindigkeitswertes und die Checkbox für den Zustand der Beleuchtung). Für die Speicherung der beiden Werte steht der Button [Daten speichern] zur Verfügung. Wird eine andere Route und/oder ein anderer Abschnitt gewählt, so aktualisieren sich die Werte entsprechend der gespeicherten Daten. Nicht gespeicherte Werte für die Fahrtgeschwindigkeit und der Status des Lichts werden dabei verworfen.

Über die Schaltfläche [S T A R T] kann die Fahrt des gewählten Zuges auf der Route gestartet werden. Dabei werden die dafür nötigen Informationen mit den Knoten ausgetauscht. Der genaue Ablauf kann der Beschreibung des Algorithmus in Kapitel 4.2.4 entnommen werden. Wurde diese Schaltfläche betätigt, wird sie für diesen Zug nicht mehr angezeigt. Sobald der Zug am Zielknoten eingetroffen ist, wird die Route aus der Liste entfernt und steht somit nicht mehr zu Verfügung.

Hinzufügen von neuen Routen

Um die Züge von einem Start zu einem Zielknoten fahren zu lassen, müssen dafür Routen ausgewählt werden. Über einen Dialog erfolgt die Wahl, indem die Adresse des Zuge bzw. dessen digitalen Decoders, die Adresse des Start- und des Zielknotens in den zur Verfügung stehenden Auswahllisten gewählt werden. Über den Button [Routen berechnen] werden anschließend alle Routen, die den Angaben entsprechen, berechnet und in der Combobox des Dialog eingetragen. An dieser Stelle kann anschließend ein Eintrag ausgewählt werden und über den Button [Route übernehmen] wird diese Route dann gespeichert, der Dialog geschlossen und zum Hauptdialog zurückgekehrt. Darin wird nun die Liste mit den Routen aktualisiert und enthält den zuvor selektierten Eintrag. Über den Button [Abbrechen] kann der Dialog zur Berechnung der Routen wieder geschlossen werden, ohne eine Route zu übernehmen. Die Berechnung der möglichen Routen wird im Kapitel 4.2.1 und die verwendeten Datenstrukturen in Kapitel 4.1.7 beschrieben. Jeder Eintrag der Auswahlliste enthält die Adresse des Zuges, die ID der Route, die Fahrtrichtung und die Abschnitte der Route.

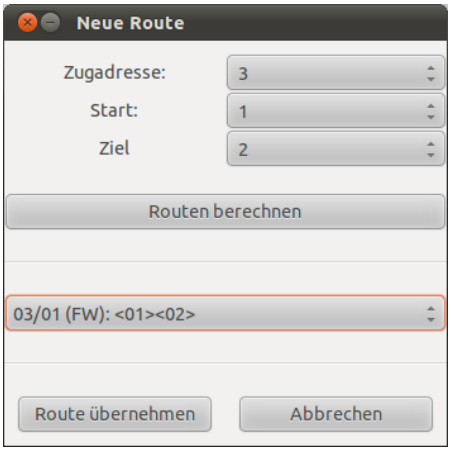

Abbildung 4.13: Dialog zum Hinzufügen von neuen Routen

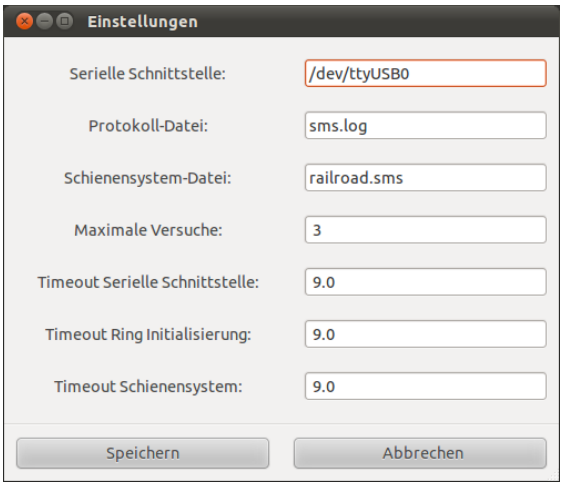

Abbildung 4.14: Dialog zum Ändern der Programmeinstellungen

Programmeinstellungen

Der Dialog zum Ändern der Programmeinstellungen kann zum Einen im Initialisierungsdialog aufgerufen werden. Dies ist insofern nötig, da eine falsche Konfiguration zu Problemen bei der Initialisierung führen könnte. Sollte dies der Fall sein, können Änderungen vorgenommen und die Initialisierung neu gestartet werden. Zum Anderen kann der Programmeinstellungs-Dialog auch über das Menü im Hauptfenster aufgerufen werden.
Die folgenden Einstellungen stehen dem Anwender zur Verfügung:

- Serielle Schnittstelle: Hier kann die serielle Schnittstelle ausgewählt werden (z.B. `/dev/ttyUSB0` für den USB0-Port).
- Protokoll-Datei: Hier kann der Dateiname (inklusive Pfad) der Protokoll-Datei ausgewählt

werden (z.B. `SMS.log`).

- Schienensystem-Datei: In diesem Feld kann jene Datei (inklusive Pfad) eingetragen werden, welche die Informationen über das Schienensystem enthält (z.B. `railroad.sms`).
- Maximale Versuche: Dieser Wert steht für die Anzahl der maximalen Versuche (im Fehlerfall) während der Initialisierung (siehe auch Kapitel 4.3.1).
- Timeout Serielle Schnittstelle: Hier kann das Timeout-Intervall für das Öffnen der seriellen Schnittstelle in Sekunden eingestellt werden, d.h. ist der Versuch die serielle Schnittstelle zu öffnen nach dieser Zeit nicht erfolgreich abgeschlossen, so wird in den Fehlerzustand gewechselt.
- Timeout Ring Initialisierung: Auch in diesem Feld steht der Wert für ein Timeout – in diesem Fall für die Initialisierung des Kommunikationsringes.
- Timeout Schienensystem: Für die Initialisierung des Eisenbahnnetzes ist es nötig, eine Datei zu lesen, welche die nötigen Informationen enthält (siehe Schienensystem-Datei). Dieses Intervall steht für die maximale Zeit, die zum Öffnen dieser Datei aufgewendet werden darf. Kann dieser Vorgang nicht innerhalb der eingestellten Zeitspanne abgeschlossen werden, wird in den Fehlerzustand gewechselt.

Die Datei mit den Informationen über den Aufbau des Schienensystems enthält für jeden Knoten, der im System vorhanden ist, genau eine Zeile mit den Informationen über die

- eigene Knotenadresse
- die Adresse des Nachbarn in Fahrtrichtung `DIR_FORWARD` mit der Weichenstellung `POS_AHEAD`
- die Adresse des Nachbarn in Fahrtrichtung `DIR_FORWARD` mit der Weichenstellung `POS_SWITCH`
- die Adresse des Nachbarn in Fahrtrichtung `DIR_BACKWARD` mit der Weichenstellung `POS_AHEAD` und
- die Adresse des Nachbarn in Fahrtrichtung `DIR_BACKWARD` mit der Weichenstellung `POS_SWITCH`

wobei die Werte in genau dieser Reihenfolge, zweistellig und in hexadezimaler Schreibweise eingetragen werden müssen. Der Wert 00 steht dafür, dass an dieser Position kein Nachbar vorhanden ist.
Ein Beispiel für ein kleines Eisenbahnnetz wäre eine Datei mit folgendem Inhalt:

```
1 01 02 00 02 04
2 02 01 03 01 03
3 03 02 04 02 00
4 04 01 00 03 00
```

Die erste Zeile steht in diesem Fall für den Knoten mit der Adresse 01 mit dem Nachbarn 02 in Fahrtrichtung `DIR_FORWARD` und Weichenstellung `POS_AHEAD` und in Fahrtrichtung `DIR_BACKWARD` mit derselben Weichenstellung. Für die andere Weichenstellung ist in Richtung `DIR_FORWARD` kein Nachbarknoten vorhanden, in die entgegengesetzte Richtung der Knoten mit der Adresse 04.

Die oben genannten Programmeinstellungen werden in einer einfachen Textdatei (`options.sms`) abgelegt. Das bedeutet, dass die Werte auch ohne Programmstart angepasst werden könnten. Kann die Datei nicht gefunden oder gelesen werden oder enthält sie falsche Informationen, so werden Standardwerte für die Parameter verwendet. Die Änderungen in den Einstellung werden immer erst beim Programmstart übernommen, weil dies der einzige Zeitpunkt ist, wo aus den Einstellungen gelesen wird. Änderungen während der Laufzeit haben somit keinen Einfluss auf den aktuellen Betrieb.

KAPITEL 5

Ausblick

Dieses Kapitel befasst sich mit Weiterentwicklungen dieses Systems in der Hard- und Software.

5.1 Hardware

Im Großen und Ganzen kann die Hardware als fertige Lösung angesehen werden. Für die folgenden Punkte gibt es aber noch mehr oder weniger große Spielräume für die Weiterentwicklung:

- Gehäuse: Um die Lebensdauer der Platine inklusive der Bauteile zu verlängern, würde sich ein Gehäuse empfehlen. Allerdings müsste gewährleistet werden, dass es zu keiner Stauung der abgegebenen Wärme des TLEs innerhalb des Gehäuses kommt. Dafür müsste eventuell noch ein Ventilator installiert und angesteuert werden. Das Gehäuse müsste Öffnungen für die Anschlüsse (Versorgung, Schiene, Weiche, Kommunikation und LEDs) besitzen.

- TLE5205-2: Dieser Bauteil erfüllt zwar alle Anforderungen, allerdings wurde die Produktion bereits eingestellt und im Handel sind nur mehr Restbestände erhältlich. Somit müsste nach einer Alternative gesucht werden. In diesem Fall wäre es optimal, wenn die Pins der neuen Motorsteuerung mit denen des TLE5205-2 übereinstimmen würden, da in diesem Fall keine Änderung des Hardware-Designs nötig wäre.

- Anzeigeeinheit: Eine hilfreiche Weiterentwicklung wäre das Installieren eines Displays (7-Segment-Anzeige oder LCD-Display) an den Knoten, die zum Beispiel die Adresse, aktuelle Daten wie Controllerstatus oder Betriebsmodus etc. visualisieren würden. Allerdings sollte abgewogen werden, ob Kosten und Nutzen in diesem Fall noch in Relation stehen würden.

5.2 Software

Im Bereich der Software bestehen mehr Möglichkeiten einer Weiterentwicklung im Sinne einer Verbesserung der Performance und auch der Visualisierung der Daten.

5.2.1 User Interface

Das aktuelle User Interface bietet dem Anwender die wichtigsten Daten in sehr kompakter Form. Der Nachteil dieser Variante besteht darin, dass der Anwender genau wissen muss, wie der Aufbau des Schienensystems aussieht und wo sich die Knoten befinden. Weiters werden die Informationen über den Aufbau des Systems über eine Textdatei der Software zur Verfügung gestellt, was eine Fehlerquelle darstellt. Aus diesen beiden Faktoren ergeben sich die folgenden Verbesserungsmöglichkeiten der Mastersoftware im Bereich der Visualisierung:

- Dem Anwender könnte ein Editor zur Verfügung gestellt werden, in dem er das Eisenbahnnetz nachbilden kann. Die unterschiedlichen Schienen- und Weichentypen müssten als Drag-and-Drop-Elemente verfügbar sein, mit denen auf einfache Weise ein Eisenbahnnetz erstellt werden kann. Es müsste ein entsprechendes Interface definiert werden, um die Informationen der Software bereitzustellen – ähnlich wie die Textdatei mit den Informationen über Knoten und Nachbarn in der aktuellen Version.

- Während des normalen Betriebs könnte eine grafische Abbildung des Eisenbahnnetzes eingesetzt werden. Dabei könnten die freien und belegten Abschnitte farblich hervorgehoben werden. Außerdem könnten die Informationen über Geschwindigkeit, Fahrtrichtung etc. in optisch ansprechender Form dem Anwender zur Verfügung gestellt werden.

Da das Ziel dieser Arbeit aber nicht ein umfangreiches User Interface war, wurde auf derartige Funktionen verzichtet.

5.2.2 Deadlocks

In der aktuellen Arbeit steht dem Anwender die Suche nach Routen zur Verfügung. Anschließend können diese ausgewählt und befahren werden. Allerdings gibt es noch keine automatische Vermeidung von Deadlocks.

Szenario

Die Software verwaltet ein Eisenbahnnetz, das aus mehreren Abschnitten einen einfachen Ring bildet und von zwei Zügen befahren wird. Haben die beiden Züge unterschiedliche Richtungen, bewegen sie sich im Ring aufeinander zu. Die Software verhindert eine Kollision, da immer die nächsten zwei Abschnitte überprüft werden und nur dann weitergefahren wird, wenn diese frei sind. In diesem Fall werden beide Züge zum Stillstand kommen. Es wird in beiden Fällen gewartet, bis die nötigen Abschnitte frei von Zügen sind. Dies erfolgt in der aktuellen Software aber nicht automatisch. Dem Anwender bleibt somit nur mehr die Möglichkeit, eine neue Route zu wählen und einen der beiden Züge in die andere Richtung fahren zu lassen.

Verbesserung

Das gezeigte Szenario behandelt eine Situation in der es zu einem Deadlock kommt. Nur über einen Anwendereingriff lässt sich diese Situation bereinigen. Eine wesentliche Verbesserung wäre ein Algorithmus, der schon bei der Suche, aber spätestens bei der Auswahl einer Route, eingreift und somit Deadlock-Situation vermieden werden. Dies kann über komplizierte Matrizenoperationen gelöst werden, für die auch die Datenstrukturen zur Speicherung der Routen angepasst werden müssten.

5.2.3 Scheduling

Für einen automatischen Betrieb über eine längeren Zeitraum, würde sich eine zeitliche Zugsteuerung empfehlen, bei der für einen Zug mehrere Routen gewählt werden können, wobei für jede Route noch eine Startzeit angegeben wird. Somit könnten reale *Fahrpläne* abgebildet werden.

5.2.4 SVN-Ablage

Im Repository *svn+ssh://thor.auto.tuwien.ac.at/home/svn/amrc* wurden die Source-Dateien der Software, das Hardware-Layoute, die Dokumentation und das zugehörige Poster abgelegt, um es für die Weiterentwicklung zur Verfügung zu stellen.

Literaturverzeichnis

[1] Datasheet L78L00. Datenblatt, 1999.

[2] AVR STK500 User Guide. Datenblatt, 06 2001.

[3] Datasheet 5-A H-Bridge for DC-Motor Applications. Datenblatt, 06 2001.

[4] Datasheet +5V RS-232 Transceivers. Datenblatt, 10 2003.

[5] Datasheet Atmel ATmega8 ATmega8L. Datenblatt, 03 2003.

[6] Inc. National Model Railroad Association. Communication Standards For Digital Command Control. *NMRA STANDARD*, (S-9.2), 2004.

[7] Inc. National Model Railroad Association. Electrical Standards For Digital Command Control. *NMRA STANDARD*, (S-9.1), 2004.