



A Layout Control System for Model Railroads

Helmut Fieres

January 8, 2025

Contents

1	Introduction	1
1.1	Elements of a Layout Control System	2
1.2	Standards, Components and Compatibility	3
1.3	This Book	3
1.4	Parts and Chapters	3
1.5	A final note	4
2	General Concepts	7
2.1	Layout Control Bus	7
2.2	Hardware Module	8
2.3	Nodes	8
2.4	Ports	9
2.5	Attributes	9
2.6	Events	9
2.7	DCC Subsystem	10
2.8	Analog Subsystem	10
2.9	Configuration Mode	11
2.10	Operation Mode	11
2.11	Summary	12
3	Message Formats	13
3.1	LCS Message Format	14
3.2	General Management	14
3.3	Node and Port Management	15
3.4	Event Management	16
3.5	DCC Track Management	16
3.6	DCC Locomotive Decoder Management	16
3.7	DCC Accessory Decoder Management	18
3.8	RailCom DCC Packet management	18
3.9	Raw DCC Packet Management	18
3.10	DCC errors and status	19
3.11	Analog Engines	19
3.12	Summary	20

CONTENTS

4	Message Protocols	21
4.1	Node startup	21
4.2	Switching between Modes	22
4.3	Setting a new Node Id	22
4.4	Node Ping	22
4.5	Node and Port Reset	23
4.6	Node and Port Access	23
4.7	Layout Event management	24
4.8	General LCS Bus Management	25
4.9	DCC Track Management	25
4.10	Locomotive Session Management	26
4.11	Locomotive Configuration Management	27
4.12	Configuration Management using RailCom	28
4.13	DCC Accessory Decoder Management	28
4.14	Sending DCC packets	28
4.15	Summary	29
5	The LCS Runtime Library RtLib	31
6	RtLib Storage	33
6.1	Node Map	33
6.2	Port Map	34
6.3	Node and Port Items	35
6.4	Event Map	35
6.5	User defined Maps	36
6.6	Periodic Task Map	36
6.7	Pending Request Map	36
6.8	Driver Function Map	36
6.9	Driver Map	37
6.10	Summary	37
7	RtLib Call Interface	39
7.1	Library initialization	39
7.2	Obtaining node information	39
7.3	Controlling a node aspect	40
7.4	Controlling extension functions	40
7.5	Reacting to events	40
7.6	Sending messages	40
7.7	Summary	40

CONTENTS

8	RtLib Callbacks	41
8.1	General Callbacks	41
8.2	Node and Port Initialization Callback	41
8.3	Node and Port Request Reply Callback	42
8.4	Node and Port Control and Info Callback	42
8.5	Inbound Event Callback	42
8.6	Console Command Line Callback	43
8.7	DCC Message Callback	43
8.8	RailCom Message Callback	44
8.9	LCS Periodic Task Callback	44
8.10	Summary	44
9	RtLib Command Interface	47
9.1	Configuration Mode Commands	47
9.2	Event Commands	47
9.3	Node Map and Attributes Commands	47
9.4	Send a raw Message	47
9.5	List node status	48
9.6	Driver commands	48
9.7	LCS message text format	48
9.8	Summary	48
10	RtLib Usage Example	49
11	The DCC Subsystem	51
11.1	Locomotive session management	51
11.2	Stationary Decoders	51
11.3	DCC packet generation	52
11.4	Sending a DCC packet	53
11.5	DCC Track Signal Generation	53
11.6	Power consumption monitoring	53
11.7	Decoder programming support	54
11.8	RailCom support	54
11.9	DCC Track sections	55
11.10A	short Glimpse at Software Implementation	56
11.11	Summary	56
12	The Analog Subsystem	59
12.1	Requirements	59
12.2	Overall concept	60

CONTENTS

12.3 Locomotive session management	61
12.4 Analog Track Signal Generation	61
12.5 Analog Track Blocks and Track subsections	61
12.6 A short Glimpse at Software Implementation	61
12.7 Summary	62
13 LCS Hardware Module Design	63
13.1 Selecting the controller	63
13.2 The Controller Platform	64
13.3 Hardware Module Schematics	65
13.4 Controller and Extension Board	65
13.5 LCS Bus connector	66
13.6 LCSNodes Extension Board Connector	66
13.7 Track Power Connectors	69
13.8 Summary	69
14 Controller Dependent Code	71
14.1 The big picture	71
14.2 Configuring the pins	72
14.3 CDC Library setup	73
14.4 General Controller Attributes and Functions	73
14.5 Power Fail detect	73
14.6 External Interrupt	73
14.7 Status LEDs	74
14.8 Timer	74
14.9 Digital IO	74
14.10Analog Input	74
14.11PWM Output	75
14.12UART Interface	75
14.13I2C	75
14.14SPI	76
14.15Extension Connector and hardware pins	76
14.16Summary	76
15 LCS Node Firmware Design	77
15.1 General Thoughts - Nodes, Ports and Events	77
15.2 General thoughts - Software layers	78
15.3 Node Functions and Attributes	79
15.4 Port Functions and Attributes	79

CONTENTS

15.5 Command Line and Display	80
15.6 Event handling	80
15.7 Periodic Tasks	80
15.8 Configuration	80
15.9 The main code	81
15.10Summary	81
16 Power Module Design	83
16.1 DCC Track Power Modules	83
16.2 Dual Power Module - L6205	84
16.3 Mono Power Module - L6205	84
16.4 Power Module - Breakout Boards	85
16.5 Summary	85
17 Railcom Signal Detector	87
17.1 Summary	87
18 The Base Station	89
18.1 Key Requirements	89
18.2 Module hardware	90
18.3 Base Station PCB	91
18.4 Summary	91
19 The Cab Handheld	93
19.1 Requirements	93
20 Signaling Block Control	95
20.1 Requirements	96
20.2 Block Element Concept	96
20.3 Turnouts	97
20.4 Signals	97
20.5 <i>ABS and APB</i>	97
20.6 Block Control Algorithms	98
20.6.1 Single Direction Line	98
20.6.2 Double Direction Line	98
20.6.3 Absolute Permissive Blocks	98
20.7 Summary	98
A LCS Nodes and EasyEda	101
A.1 Symbols and Footprints	101
A.1.1 Symbols	101

CONTENTS

A.1.2	Main Controller Board Footprints	101
A.2	Extension Boards Footprints	102
A.3	Links	104
B	Tests	105
B.1	Schematics	105
B.1.1	part 1	105
B.1.2	part 2	105
B.1.3	part 3	106
B.2	Pictures	107
B.2.1	Variable word layout	107
B.3	Protocol boxes	107
B.4	Split rectangle	108

1 Introduction

Model railroading. A fascinating hobby with many different facets. While some hobbyist would just like to watch trains running, others dive deeper into parts of their hobby. Some build a realistic scenery and model a certain time era with realistic operations. Others build locos and rolling equipment from scratch. Yet others enjoy the basic benchwork building, electrical aspects of wiring and control. They all have in common that they truly enjoy their hobby.

This little book is about the hardware and software of a layout control system for managing a model railroad layout. Controlling a layout is as old as the hobby itself. I remember my first model railroad. A small circle with one turnout, a little steam engine and three cars. Everything was reachable by hand, a single transformer supplied the current to the locomotive. As more turnouts were added, the arm was not long enough any more, simple switches, electrical turnouts and some control wires came to the rescue. Over time one locomotive did not stay alone, others joined. Unfortunately, being analog engines, they could only be controlled by electric current to the track. The layout was thus divided into electrical sections. And so on and so on. Before you know it, quite some cabling and simple electrical gear was necessary.

Nearly four decades ago, locomotives, turnouts, signals and other devices on the layout became digital. With growing sophistication, miniaturization and the requirement to model operations closer and closer to the real railroad, layout control became a hobby in itself. Today, locomotives are running computers on wheels far more capable than computers that used to fill entire rooms. Not to mention the pricing. Turnout control and track occupancy detection all fed into a digital control system, allowing for very realistic operations.

The demands for a layout control system can be divided into three areas. The first area is of course **running** locomotives. This is what it should be all about, right? Many locomotives need to be controlled simultaneously. Also, locomotives need to be grouped into consists for large trains, such as for example a long freight train with four diesel engines and fifty boxcars. Next are the two areas **observe** and **act**. Track occupancy detection is a key requirement for running multiple locomotives and knowing where they are. But also, knowing which way a turnout is set, the current consumption of a track section are good examples for layout observation. Following observation is to act on the information gathered. Setting turnouts and signals or enabling a track section are good examples for acting on an observation.

Running, observing and acting requires some form of **configurations** and **operations**. What used to be a single transformer, some cabling and switches has turned into computer controlled layout with many devices and one or more bus systems. Sophisticated layouts need a way to configure the locomotives, devices and manage operations of layouts. Enter the world of digital control and computers.

After several decades, there is today a rich set of product offerings and standards available. There are many vendors offering hardware and software components as well as entire systems. Unfortunately they are often not compatible with each other. Furthermore, engaged open software communities took on to build do it yourself systems more or less compatible with vendors in one or the other way. There is a lively community of hardware and software designers building hardware and software layout control systems more or less from scratch or combined using existing industry products.

1.1 Elements of a Layout Control System

Before diving into concept and implementation details, let's first outline what is needed and what the resulting key requirements are. Above all, our layout control system should be capable to simultaneously run locomotives and manage all devices, such as turnouts and signals, on the layout. The system should be easy to expand as new ideas and requirements surface that need to be integrated without major incompatibilities to what was already built.

Having said that, we would need at least a **base station**. This central component is the heart of most systems. A base station needs to be able to manage the running locomotives and to produce the DCC signals for the track where the running locomotive is. There are two main DCC signals to generate. One for the main track or track sections and one for the programming track. This is the track where a locomotive decoder can be configured. A base station could also be the place to keep a dictionary of all known locomotives and their characteristics. In addition to interfaces to issues commands for the running locomotives, there also need to be a way to configure the rolling stock.

Complementing the base station is the **booster** or **block controller** component that produce the electrical current for a track section. The booster should also monitor the current consumption to detect electrical shortages. Boosters comes in several ranges from providing the current for the smaller model scales as well as the larger model scales which can draw quite a few amps. There could be many boosters, one for each track section. The base station provides the signals for all of them.

The **cab handheld** is the controlling device for a locomotive. Once a session is established, the control knobs and buttons are used to run the locomotive. Depending on the engine model, one could imagine a range of handhelds from rather simple handhelds just offering a speed dial and a few buttons up to a sophisticated handheld that mimics for example a diesel engine cab throttle stand.

With these three elements in place and a communication method between them, we are in business to run engines. Let's look at the communication method. Between the components, called nodes, there needs to be a **communication bus** that transmits the commands between them. While the bus technology itself is not necessarily fixed, the messaging model implemented on top is. The bus itself has no master, any node can communicate with any other node by broadcasting a message, observed by all other nodes. Events that are broadcasted between the nodes play a central role. Any node can produce events, any node can consume events. Base station, boosters and handhelds are just nodes on this bus.

But layouts still need more. There are **signals**, **turnouts** and **track detectors** as well as **LEDs**, **switches**, **buttons** and a whole lot more things to imagine. They all need to be connected to the common messaging bus. The layout control system needs to provide not only the hardware interfaces and core firmware for the various device types to connect, it needs to also provide a great flexibility to configure the interaction between them. Pushing for example a button on a control field should result in a turnout being set, or even a set of turnouts to guide a train through a freight-yard and so on.

Especially on larger layouts, **configuration** becomes quite an undertaking. The **configuration model** should therefore be easy and intuitive to understand. The elements to configure should all follow the same operation principles and be extensible for specific functions. A computer is required for configuration. Once configured however, the computer is not required for operations. The capacity, i.e. the number of locomotives, signals, turnouts and other devices managed should be in the thousands.

Configuration as well as operations should be possible through sending the defined messages as well as a simple ASCII commands send to the base station which in turn generates the

messages to broadcast via the common bus. A computer with a graphical UI would connect via the USB serial interface using the text commands.

1.2 Standards, Components and Compatibility

The DCC family of standards is the overall guiding standard. The layout system assumes the usage of DCC locomotive decoder equipped running gear and DCC stationary decoder accessories. Beyond this set of standards, it is not a requirement to be compatible with other model railroad electronic products and communication protocols. This does however not preclude gateways to interact in one form or another with such systems. An example is to connect to a LocoNet system via a gateway node. Right now, this is not in scope for our first layout system.

All of the project should be well documented. One part of documentation is this book, the other part is the thoroughly commented LCS core library and all software components built on top. Each lesson learned, each decision taken, each tradeoff made is noted, and should help to understand the design approach taken. Imagine a fast forward of a couple of years. Without proper documentation it will be hard to remember how the whole system works and how it can be maintained and enhanced.

With respect to the components used, it uses as much as possible off the shelf electronic parts, such as readily available microcontrollers and their software stack as well as electronic parts in SMD and non-SMD form, for building parts of the system. The concepts should not restrict the development to build it all from scratch. It should however also be possible to use more integrated elements, such as a controller board and perhaps some matching shields, to also build a hardware module.

1.3 This Book

This book will describe my version of a layout control system with hardware and software designed from the ground up. The big question is why build one yourself. Why yet another one? There is after all no shortage on such systems readily available. And there are great communities out there already underway. The key reason for doing it yourself is that it is simply fun and you learn a lot about standards, electronics and programming by building a system that you truly understand from the ground up. To say it with the words of Richard Feynman

“What I cannot create, I do not understand. – Richard Feynman”

Although it takes certainly longer to build such a system from the ground up, you still get to play with the railroad eventually. And even after years, you will have a layout control system properly documented and easy to support and enhance further. Not convinced? Well, at least this book should be interesting and give some ideas and references how to go after building such a system.

1.4 Parts and Chapters

The book is organized into several parts and chapters. The first chapters describe the underlying concepts of the layout control system. Hardware modules, nodes, ports and events and their interaction are outlined. Next, the set of messages that are transmitted between the components

and the message protocol flow illustrate how the whole system interacts. With the concepts in place, the software library available to the node firmware programmer is explained along with example code snippets. After this section, we all have a good idea how the system configuration and operation works. The section is rounded up with a set of concrete programming examples.

Perhaps the most important part of a layout control system is the management of locomotives and track power. After all, we want to run engines and play. Our system is using the DCC standard for running locomotives and consequently DCC signals need to be generated for configuring and operating an engine. A base station module will manage the locomotive sessions, generating the respective DCC packets to transmit to the track. Layouts may consist of a number of track sections for which a hardware module is needed to manage the track power and monitor the power consumption. Finally, decoders can communicate back and track power modules need to be able to detect this communication. Two chapters will describe these two parts in great detail.

The next big part of the book starts with the hardware design of modules. First the overall outline of a hardware module and our approach to module design is discussed. Building a hardware module will rest on common building blocks such as a CAN bus interface, a microcontroller core, H-Bridges for DCC track signal generation and so on. Using a modular approach the section will describe the building blocks developed so far. It is the idea to combine them for the purpose of the hardware module.

With the concepts, the messages and protocol, the software library and the hardware building blocks in place, we are ready to actually build the necessary hardware modules. The most important module is the base station. Next are boosters, block controllers, handhelds, sensor and actor modules, and so on. Finally, there are also utility components such as monitoring the DCC packets on the track, that are described in the later chapters. Each major module is devoted a chapter that describes the hardware building blocks used, additional hardware perhaps needed, and the firmware developed on top of the core library specifically for the module. Finally, there are several appendices with reference information and further links and other information.

1.5 A final note

A final note. "Truly from the ground up" does not mean to really build it all yourself. As said, there are standards to follow and not every piece of hardware needs to be built from individual parts. There are many DCC decoders available for locomotives, let's not overdo it and just use them. There are also quite powerful controller boards along with great software libraries for the micro controllers, such as the CAN bus library for the AtMega Controller family, already available. There is no need to dive into all these details.

The design allows for building your own hardware just using of the shelf electronic components or start a little more integrated by using a controller board and other breakout boards. The book will however describe modules from the ground up and not use controller boards or shields. This way the principles are easier to see. The appendix section provides further information and links on how to build a system with some of the shelf parts instead of building it all yourself. With the concepts and software explained, it should not be a big issue to build your own mix of hardware and software.

I have added most of the source files in the appendix for direct reference. They can also be found also on GitHub. (Note: still to do...) Every building block schematic shown was used and tested in one component or another. However, sometimes the book may not exactly match the material found on the web or be slightly different until the next revision is completed. Still, looking at portions of the source in the text explain quite well what it will do. As said, it is

CHAPTER 1. INTRODUCTION

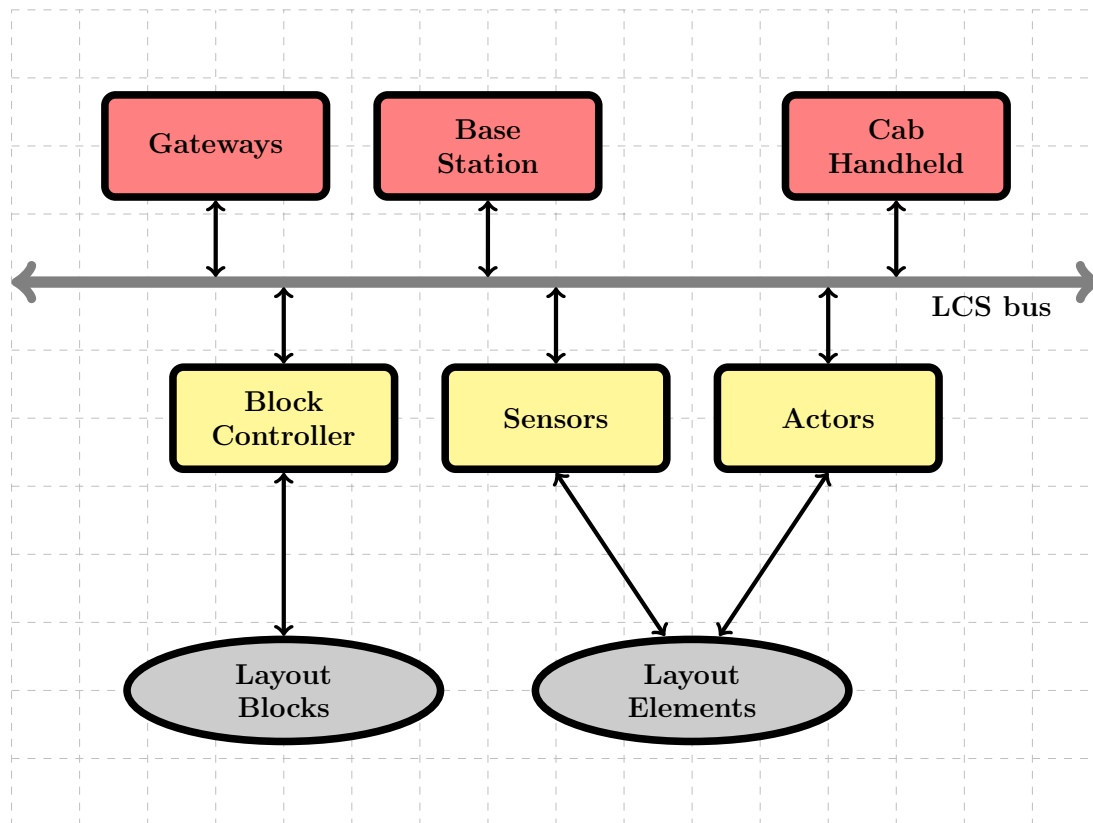
the documentation that hopefully in a couple of years from now still tells you what was done so you can adapt and build upon it. And troubleshoot.

The book hopefully also helps anybody new to the whole subject with good background and starting pointers to build such a system. I also have looked at other peoples great work, which helped a lot. What I however also found is that often there are rather few comments or explanations in the source and you have to partially reverse engineer what was actually build for understanding how things work. For those who simply want to use an end product, just fine. There is nothing wrong with this approach. For those who want to truly understand, it offers nevertheless little help. I hope to close some of these gaps with a well documented layout system and its inner workings.

In the end, as with any hobby, the journey is the goal. The reward in this undertaking is to learn about the digital control of model railroads from running a simple engine to a highly automated layout with one set of software and easy to build and use hardware components. Furthermore, it is to learn about how to build a track signaling system that manages analog and digital engines at the same time. So, enjoy.

2 General Concepts

At a higher level, the layout control system consists of components and a communication scheme. This chapter will define the key concepts of a layout system. At the heart of the layout control system is a common communication bus to which all modules connect. The others key elements are node, events, ports and attributes. Let's define these items first and then talk about how they interact. The following figure depicts the high level view of a layout control system.



2.1 Layout Control Bus

The layout control bus is the backbone of the entire system. The current implementation is using the industry standard CAN bus. All hardware modules connect to this bus and communicate via messages. All messages are broadcasted and received by all other hardware modules on the bus. The classic CAN bus standard limits the message size to 8 bytes and this is therefore the maximum message size chosen for the LCS bus. The CAN bus also has a hardware module limit of about 110 modules for bandwidth reasons. But even for a large layout this should be sufficient. And for really large layouts, another bus system or a system with CAN bus routers, could be envisioned. The software should therefore be designed to manage thousands of connected modules. While the CAN bus technology could be exchanged, the message format and size defined as well as the broadcasting paradigm are fixed in the overall design and will not change.

2.2 Hardware Module

Everything connected to the LCS bus is a **hardware module**, which is the physical entity connected to the bus. Typically it is a micro controller with the bus interface and hardware designed for the specific purpose. For example, a CAN bus interface, an AtMega Controller, and digital output drivers could form a hardware module to control railroad turnouts and signals. Base stations, handhelds and gateways are further examples of a hardware module. Hardware modules are expected to be physically located near their use and thus spread throughout the layout. Some hardware modules could be at locations that cannot be reached easily. So all interaction for configuration and operations needs to be possible through the messages on the bus. Nevertheless, putting local controls on a hardware module should not be prohibited.

A hardware module consists of a controller part and a node specific part. The controller part is the **main controller**, which consists of the controller chip, a non-volatile memory to retain any data across power down, a CAN bus interface and interfaces to the node specific hardware. The node specific hardware is called the **node extension**. Conceptually, both parts can be one monolithic implementation on one PCB board, but also two separate units connected by the extension connector. There are defined connectors between the boards. The hardware chapter will go into more detail on the board layouts and hardware design options.

2.3 Nodes

A hardware module is the physical implementation. A **node** is the software entity running in the firmware of the hardware module. Nodes are the processing elements for the layout. Conceptually, a hardware module can host more than one node. The current implementation however supports only one node on a given hardware module. A node is uniquely identified through the **node identifier**. There are two ways to set a `nodeId`. The first is to have central component to assign these numbers on request. The second method sets the number manually. Although a producer consumer scheme would not need a `nodeId`, there are many operations that are easier to configure when explicitly talking to a particular node. Both nodes and event identifiers are just numbers with no further classification scheme. A configuration system is expected to provide a classification grouping of nodes and event number ranges if needed.

A node also has a **node type**, to identify what the node is capable of. Examples of nodes types are the base station, a booster, a switch module, a signal control module, and so on. While the node number is determined at startup time and can change, the node type is set via the module firmware. As the node type describes what the hardware module can do the type cannot change unless the module changes. Once the node has an assigned node number, configuration tools can configure the node via configuration messages to set the respective node variables.

A node needs to be configured and remember its configuration. For this purpose, each node contains a **node map** that keeps all the information about the node, such as the number of ports, the node unique Id and so on. There is also a small set of user definable attributes to set data in a node map specific to the node. The data is stored in non-volatile memory space and on power up the node map is used to configure the node. If the module is a new module, or a module previously used in another layout, or the firmware version requires a new data layout of the node map, there is a mechanism to assign a new node number and initialize the node map with default values.

2.4 Ports

A node has a set of receiving targets, called ports. Ports connect the hardware world to the software world, and are the connection endpoints for events and actions. For example, a turnout digital signal output could be represented to the software as a port on a node. The node registers its interest in the event that target the signal. An event sent to the node and port combination then triggers a callback to the node firmware to handle the incoming events. Although a node can broadcast an event anytime by just sending the corresponding message, the event to send is typically associated with an outbound port for configuration purposes. In addition to the event immediate processing, the event handling can be associated with a timer delay value. On event reception the timer value will delay the event callback invocation or broadcast.

A node has a **port map** that contains one entry for each defined port. **port map entries** describe the configuration attributes and state of the port such as the port type. There is also a small set of user definable attributes to set data in a port map entry specific to the port. These attributes can be used by the firmware programmer to store port specific data items such as a hardware pin or a limit value in the port map.

2.5 Attributes

Node attributes and **port attributes** are conceptually similar to the CV resources in a DCC decoder. Many decoders, including the DCC subsystem decoders, feature a set of variables that can be queried or set. The LCS layout system implements a slightly different scheme based on items. In contrast to a purely decoder variable scheme an item can also just represent just an action such as setting an output signal. Items are passed parameter data to further qualify the item. Items are just numbers assigned. The range of item numbers is divided into a reserved section for the layout system itself, and a user defined range that allows for a great flexibility to implement the functions on a particular node and port. The meaning of user defined items is entirely up to the firmware programmer. If it is desired to have a variables, a combination of items and attributes can provide the traditional scheme as well. In addition, there are node local variables, called attributes, available to the firmware programmer for storing data items.

2.6 Events

The LCS message bus, hardware module, node and ports describe layout and are statically configured. For nodes to interact, **events** and their configuration is necessary. An event is a message that a node will broadcast via the bus. Every other node on this bus will receive the event and if interested act on the event. The sender is the producer, the receiver is the consumer. Many producers can produce the same event, many consumers can act on the same event. The **event Id**, a 16 bit number, is unique across the layout and assigned by a configuration tool during the configuration process. Other than being unique, there is no special meaning, the number is arbitrary. There are in total 65536 events available.

In addition to the event Id, an event message contains the node Id of the sender. While most events will be an ON/OFF event, events can also have additional data. For example an overload event sent by a booster node, could send the actual current consumption value in the event message. A consumer node registers its interest in an event by being configured to react to this event on a specific port. The node maintains an **event map**, which contains one entry for each event id / port id combination. For the eventing system to work, the nodeID is not required. Any port on any node can react to an event, any node can broadcast an event.

To connect producers to consumers, both parties need to be told what to do with a defined event. A producer node outbound port needs to be told what event to send for a given sensor observation. For example, a simple front panel push button needs to be told what event to send when pushed. Likewise, a consumer node inbound port needs to be told what events it is interested in and what the port should do when this event is received. Both meet through the event number used. While an inbound port can be configured to listen to many event Ids, an outbound port will exactly broadcast one eventId.

Any port on any node can react to an event, any node can broadcast an event. Still, addressing a node and port combination explicitly is required for two reasons. The first is of course the configuration of the node and port attributes. Configuration data needs to go directly to the specified node and port. The second reason is for directly accessing a resource on the layout. For example, directly setting a turnout connected to one node. While this could also be implemented with associated an event to send when operating a turnout, it has shown beneficial and easier to configure also directly access such a resource through a dedicated node/port address.

2.7 DCC Subsystem

The node, ports and events are the foundation for building a layout system based on the producer / consumer scheme. The scheme will be used heavily for implementing turnout control, signals, signal blocks and so on. In addition, there is the management of the mobile equipment, i.e. locomotives. The DCC subsystem is the other big part of our layout control system. In a sense it is another bus represented by the track sections.

LCS messages for DCC commands are broadcasted from controlling devices. For example, a handheld broadcasts a speed setting DCC command. In a layout there is one base station node which is responsible to produce the DCC signals for the track. The DCC signals are part of the physical LCS bus. While a base station design could directly supply the signal current to the track, larger layouts will typically have one or more boosters. They take the DCC signal from the LCS bus lines and generate the DCC signal current for their track section. All LCS messages for DCC operations are broadcasting messages, all nodes can send them, all nodes can receive them. Handhelds, base station and boosters are thus just nodes on the LCS bus. Only the base station will however generate the DCC signal.

The DCC standard defines mobile and stationary decoders. The DCC signal could also be used to control for example a set of turnouts via a stationary decoder. The LCS DCC message set contains messages for addressing a stationary decoder. Since the commands for stationary equipment are just DCC commands, they will be transmitted via the track as well and take away bandwidth on the track. A layout will therefore more likely use the LCS bus for implementing the management of stationary equipment. Besides, the producer / consumer model allows for a much greater flexibility when building larger and partially automated layouts.

2.8 Analog Subsystem

The layout control system is primarily a digital control system. There are however layout use cases where there are many analog locomotives that would represent a significant investment when converting to DCC or that cannot easily be equipped with a DCC decoder. In a DCC subsystem the decoder is in the locomotive and many locomotives can run therefore on the same track. In an analog system, the locomotive has no capabilities and therefore the track needs to be divided into sections that can be controlled individually. One locomotive per section is the

condition. In a sense the decoder becomes part of the track section. The layout control system offers support for building such a track section subsystem. Often the sections are combined into blocks and build the foundation for a block signaling system. Note that the rest of the layout control system is of course digital. What is typically the booster to support a section of track, is the block controller for an analog layout. We will see in the later chapters that booster and block controller are very similar and design a block controller to accommodate both use cases.

2.9 Configuration Mode

Before operations the nodes, ports and events need to be configured. Once a node has an assigned valid nodeId, the node configuration is the process of configuring a node global information, the event map information and the finally the port information. The information is backed by non-volatile storage, such that there is a consistent state upon node power up. During operations, these value can of course change, but are always reset to the initial value upon startup.

The primary process of configuration is inventing events numbers and assigning them producers and consumers. The process follows the general "if this then that" principle. On the producer side the configuration process assigns a port to an event, i.e. the push of a button to an event to send. If this button is pushed then send that event. On the consumer side the configuration process is to assign the event to a port. If this event is received then execute that port action.

After the node is up and running with a valid node Id, there are event configuration messages than can be send to the node to set the event mapping table with this information. The event map table is the mapping between the event and the port associated. Events are thus configured by "teaching" the target node what port to inform about an occurring event.

2.10 Operation Mode

Besides the basic producer/consumer model with the event messages as communication mechanism, there are several LCS control and info messages used for managing the overall layout with signals turnouts and so on as well as the physical track and the running equipment. In a layout, the track typically consist of one or more sections, each managed by a booster or block controller node. Track sections are monitored for their power consumption to detect short circuits. Back communication channels such as RailCom are handled by the booster node and provide information about the running equipment. Stationary equipment such as turnouts and signals as well as detectors, such as track occupancy detectors or turnout setting detectors are monitored and controlled through LCS messages and the event system. Conceptually any node can send and receive such event, info or control messages. Some nodes, however have a special role.

For example, the key module for layout operations is the **base station**. The base station, a node itself, is primarily responsible for managing the active locomotives on the layout. When a control handheld wants to run a locomotive, a cab session for that locomotive is established by the base station. Within the session, the locomotive speed, direction and functions are controlled through the cab handheld sending the respective messages. The base station is responsible for generating the DCC packets that are sent by the booster or block controller power module to the actual track sections. Booster and block controller module are - you guessed it - node themselves.

Finally, there are LCS nodes that represent cab handhelds to control a locomotive or consists, layout panel connectors, gateways to other layout protocols, sensors and actors to implement for example turnout control, signaling, section occupancy detections and many more. All these components share the common LCS bus and use ports and events to implement the capabilities for operating a layout.

In a layout with many track sections the **block controller** is a special node that will manage a block on the layout. Like all other nodes, a block controller itself is a node that can react to events and is controller and monitored by LCS messages. There will be several chapters devoted to this topic later.

2.11 Summary

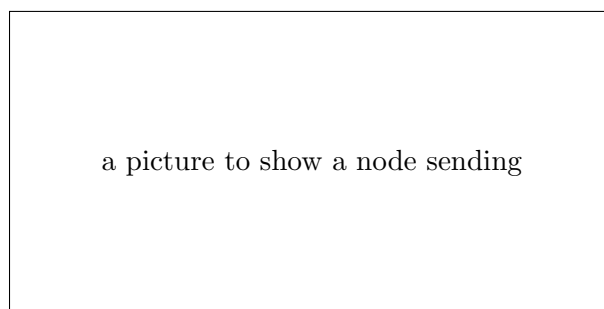
This chapter introduced the basic concepts of the layout control system described in this book. It follows very few overall guiding principles. Above all, there is the clear separation of what needs to be available for operating the mobile equipments, i.e. locomotives, and the stationary layout elements. Controlling mobile decoders are left to the DCC subsystem, all other communication takes place via the LCS bus, which is the bus to which all of the hardware modules connect. Hardware modules host the nodes. Currently, a hardware module hosts exactly one node. A node can contains one or many ports, which are the endpoints for the event system. There is a set of user allocated attributes available to node and ports. Node, port and attribute data are backed by non-volatile memory, so that a restart will use defined initial values. Nodes and their ports are also directly addressable, which is needed for configuration purposes and the directly addressable components model. Using the producer / consumer paradigm, sensors generate events and interested actors just act on them. The configuration process is simply to assign the same event to the producer node and consumer node / port id when they should work together.

The communication bus should rest on a reliable bus with a sufficient bandwidth. Although the CAN bus is used in the initial implementation, it is just one option and other technologies can be considered. In all cases however, the message format should be available for a variety of bus technologies. Our messages are therefore short, up to eight data bytes. This causes on the one hand some complexity for data items larger than a few bytes on the other hand no messages blocks the bus for a longer period. The bus technology is expected to reliably deliver a message but does not ensure its processing. This must be ensured through a request reply message scheme built on top.

3 Message Formats

Before diving into the actual design of the software and hardware components, let us first have a look at the message data formats as they flow on the layout control bus. It is the foundation of the layout control subsystem. This chapter will provide the overview on the available messages and give a short introduction to what they do. Later chapters build on it and explain how the messages are used for designing LCS node functions.

All nodes communicate via the layout control bus by broadcasting messages. Every node can send a message, and every node receives the message broadcasted. There is no central master.



Since all nodes receive all messages, a node needs to decide whether to react to a message or not. General management and emergency type messages are handled by all nodes. A reply to a specific request will only be handled by the requesting node. The layout control system defines a fairly large set of messages, which can be grouped into several categories:

- General management
- Node and Port management
- Event management
- DCC Track management
- DCC Locomotive Decoder management
- DCC Accessory Decoder management
- RailCom DCC Packet management
- Raw DCC Packet management

The current implementation is using the CAN bus, which ensures by definition that a message is correctly transmitted. However, it does not guarantee that the receiver actually processed the message. For critical messages, a request-reply scheme is implemented on top. Also, to address possible bus congestion, a priority scheme for messages is implemented to ensure that each message has a chance for being transmitted.

3.1 LCS Message Format

A message is a data packet of up to 8 bytes. The first byte represents the operation code. It encodes the length of the entire packet and opcode number. The first 3 bits represent the length of the message, the remaining 5 bits represent the opCode. For a given message length, there are 32 possible opcode numbers. The last opcode number in each group, 0x1F, is reserved for possible extensions of the opcode number range. The remaining bytes are the data bytes, and there can be zero to seven bytes.

The message format is independent of the underlying transport method. If the bus technology were replaced, the payload would still be the same. For example, an Ethernet gateway could send those messages via the UDP protocol. The messages often contain 16-bit values. They are stored in two bytes, the most significant byte first and labeled “xxx-H” in the message descriptions to come. The message format shown in the tables of this chapter just presents the opCode mnemonic. The actual value can be found in the core library include file.

The byte fields names in an LCS message are explained in greater detail when we discuss the runtime library. For this chapter, the term **npId-x** will refer to node/port identifier, the term **sId** to a locomotive session. The remaining message field names, such as **UID** or **spDir** are fairly self-explaining.

3.2 General Management

The general management message group contains commands for dealing with the layout system itself. The reset command (**RESET**) directs all hardware modules, a node, or a port on a node to perform a reset. The entire bus itself can be turned on and off (**BUS-ON**, **BUS-OFF**), enabling or suppressing the message flow. Once the bus is off, all nodes wait for the bus to be turned on again. Finally, there are messages for pinging a node (**PING**), perform data synchronization operations (**SYNC**) and request acknowledgement (**ACK/ERR**).

Table 3.1: General Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
RESET	npId-H	npId-L	flags				
BUS-ON							
BUS-OFF							
SYS-TIME	arg1	arg2	arg3	arg4			
LCS-INFO	arg1	arg2	arg3	arg4			
PING	npId-H	npId-L					
SYNC	npId-H	npId-L	arg				
ACK	npId-H	npId-L					
ERR	npId-H	npId-L	code	arg1	arg2		

Additional Notes

- Do we need a message for a central system time concept?
- Do we need a message for a message that describes the global LCS capabilities?
- Do we need an emergency stop message that every node can emit?

3.3 Node and Port Management

When a hardware module is powered on, the first task is to establish the node Id in order to broadcast and receive messages. The (REQ-NID) and (REP-ID) messages are the messages used to implement the protocol for establishing the nodeId. More on this in the chapter on message protocols. A virgin node has the hardware module-specific node type and a node Id of NIL also be set directly through the (SET-NID) command. This is typically done by a configuration tool.

Table 3.2: Node and Port Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
REQ-NID	nId-H	nId-L	nUID-4	nUID-3	nUID-2	nUID-1	flags
REP-NID	nId-H	nId-L	nUID-4	nUID-3	nUID-2	nUID-1	flags
SET-NID	nId-H	nId-L	nUID-4	nUID-3	nUID-2	nUID-1	flags
NCOL	nId-H	nId-L	nUID-4	nUID-3	nUID-2	nUID-1	

All nodes monitor the message flow to detect a potential node collision. This could be for example the case when a node from one layout is installed in another layout. When a node detects a collision, it will broadcast the (NCOL) message and enter a halt state. Manual interaction is required. A node can be restarted with the (RES-NODE) command, given that it still reacts to messages on the bus. All ports on the node will also be initialized. In addition a specific port on a node can be initialized. The hardware module replies with an (ACK) message for a successful node Id and completes the node Id allocation process. As the messages hows, node and port ID are combined. LCS can accommodate up to 4095 nodes, each of which can host up to 15 ports. A Node ID 0 is the NIL node. Depending on the context, a port Id of zero refers all ports on the node or just the node itself.

The query node (NODE-GET) and node reply messages (NODE-REP) are available to obtain attribute data from the node or port. The (NODE-SET) allows to set attributes for a node or port for the targeted node. Items are numbers assigned to a data location or an activity. There are reserved items such as getting the number of ports, or setting an LED. In addition, the firmware programmer can also define items with node specific meaning. The firmware programmer defined items are accessible via the (NODE-REQ) and (NODE-REP) messages.

Table 3.3: Node and Port Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
NODE-GET	npId-H	npId-L	item	arg1-H	arg1-L	arg2-H	arg2-L
NODE-PUT	npId-H	npId-L	item	val1-H	val1-L	val2-H	val2-L
NODE-REQ	npId-H	npId-L	item	arg1-H	arg1-L	arg2-H	arg2-L
NODE-REP	npId-H	npId-L	item	arg1-H	arg1-L	arg2-H	arg2-L

Nodes do not react to attribute and user defined request messages when in operations mode. To configure a node, the node needs to be put into configuration mode. The (OPS) and (CFG) commands are used to put a node into configuration mode or operation mode. Not all messages are supported in operations mode and vice versa. For example, to set a new nodeId, the node first needs to be put in configuration mode. During configuration mode, no operational messages are processed.

Table 3.4: Node and Port Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
OPS	npId-H	npId-L					
CFG	npId-H	npId-L					

3.4 Event Management

The event management group contains the messages to configure the node event map and messages to broadcast an event and messages to read out event data. The (SET-NODE) with the item value to set and remove an event map entry from the event map is used to manage the event map. An inbound port can register for many events to listen to, and an outbound port will have exactly one event to broadcast. Ports and Events are numbered from 1 onward. When configuring, the portId NIL has a special meaning in that it refers to all portIds on the node.

Table 3.5: Event Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
EVT-ON	npId-H	npId-L	evId-H	evId-L			
EVT-OFF	npId-H	npId-L	evId-H	evId-L			
EVT	npId-H	npId-L	evId-H	evId-L	arg-H	arg-L	

3.5 DCC Track Management

Model railroads run on tracks. Imagine that. While on a smaller layout, there is just the track, the track on a larger layout is typically divided into several sections, each controlled by a track node (centralized node or decentralized port). The system allows to report back the track sections status (in terms of occupied, free, and detecting the number of engines currently present). These messages allow the control of turnouts and monitoring of sections' status.

Table 3.6: DCC Track Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
TON	npId-H	npId-L					
TOF	npId-H	npId-L					

3.6 DCC Locomotive Decoder Management

Locomotive management comprises the set of messages that the base station uses to control the running equipment. To control a locomotive, a session needs to be established (REQ-LOC). This command is typically sent by a cab handheld and handled by the base station. The base station allocates a session and replies with the (REP-LOC) message that contains the initial settings for the locomotive speed and direction. (REL-LOC) closes a previously allocated session. The

base station answers with the (REP-LOC) message. The data for an existing DCC session can be requested with the (QRY-LOC) command. Data about a locomotive in a consist is obtained with the (QRY-LCON) command. In both cases the base station answers with the (REP-LOC) message.

Table 3.7: DCC Locomotive Decoder Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
REQ-LOC	adr-H	adr-L	flags				
REP-LOC	sId	adr-H	adr-L	spDir	fn1	fn2	fn3
REL-LOC	sId						
QRY-LOC	sId						
QRY-LCON	conId	index					

Once the locomotive session is established, the (SET-LSPD), (SET-LMOD), (SET-LFON), (SET-LOF) and (SET-FGRP) are the commands sent by a cab handheld and executed by the base station to control the locomotive speed, direction and functions. (SET-LCON) deals with the locomotive consist management and (KEEP) is sent periodically to indicate that the session is still alive. The locomotive session management is explained in more detail in a later chapter when we talk about the base station.

Table 3.8: DCC Locomotive Decoder Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SET-LSPD	sId	spDir					
SET-LMOD	sId	flags					
SET-LFON	sId	fNum					
SET-LFOF	sId	fNum					
SET-FGRP	sId	fGrp	data				
SET-LCON	sId	conId	flags				
KEEP	sId						

Locomotive decoders contain configuration variables too. They are called CV variables. The base station node supports the decoder CV programming on a dedicated track with the (REQ-CVS), (REP-CVS) and (SET-CVS) messages. The (SET-CVM) message supports setting a CV while the engine is on the main track. (DCC-ERR) is returned when an invalid operation is detected.

Table 3.9: DCC Locomotive Decoder Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SET-LSPD	sId	cv-H	cv-L	mode	val		
REQ-CVS	cv-H	cv-L	mode	val			
REP-CVS	cv-H	cv-L	val				
SET-CVS	cv-H	cv-L	mode	val			

The SET-CVM command allows to write to a decoder CV while the decoder is on the main track. Without the RailCom channel, CVs can be set but there is not way to validate that the operation was successful.

3.7 DCC Accessory Decoder Management

Besides locomotives, the DCC standards defines stationary decoders, called accessories. An example is a decoder for setting a turnout or signal. There is a basic and an extended format. The (SET-BACC) and (SET-EACC) command will send the DCC packets for stationary decoders. Similar to the mobile decoders, there are POM / XPOM messages to access the stationary decoder via RailCom capabilities.

Table 3.10: DCC Accessory Decoder Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SET-BACC	adr-H	adr-L	flags				
SET-EACC	adr-H	adr-L	val				

These commands are there for completeness of the DCC control interfaces. There could be devices that are connected via the DCC track that we need to support. However, in a layout control system the setting of turnouts, signals and other accessory devices are more likely handled via the layout control bus messages and not via DCC packets to the track. This way, there is more bandwidth for locomotive decoder DCC packets.

3.8 RailCom DCC Packet management

With the introduction of the RailCom communication channel, the decoder can also send data back to a base station. The DCC POM and XPOM packets can now not only write data but also read out decoder data via the RailCom back channel. The following messages allow to send the POM / XPOM DCC packets and get their RailCom based replies.

Table 3.11: RailCom DCC Packet management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SET-MPOM	sId	ctrl	arg1	arg2	arg3	arg4	
REQ-MPOM	sId	ctrl	arg1	arg2	arg3	arg4	
REP-MPOM	sId	ctrl	arg1	arg2	arg3	arg4	
SET-APOM	adr-H	adr-L	ctrl	arg1	arg2	arg3	arg4
REQ-APOM	adr-H	adr-L	ctrl	arg1	arg2	arg3	arg4
REP-APOM	adr-H	adr-L	ctrl	arg1	arg2	arg3	arg4

The XPOM messages are DCC messages that are larger than what a CAN bus packet can hold. With the introduction of DCC-A such a packet can hold up to 15 bytes. The LCS messages therefore are sent in chunks with a frame sequence number and it is the responsibility of the receiving node to combine the chunks to the larger DCC packet.

3.9 Raw DCC Packet Management

The base station allows to send raw DCC packets to the track. The (SEND-DCC3), (SEND-DCC4), (SEND-DCC5) and (SEND-DCC6) are the messages to send these packets. Any node can broadcast

such a message, the base station is the target for these messages and will just send them without further checking. So you better put the DCC standard document under your pillow.

Table 3.12: RRaw DCC Packet Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SEND-DCC3	arg1	arg2	arg3				
SEND-DCC4	arg1	arg2	arg3	arg4			
SEND-DCC5	arg1	arg2	arg3	arg4	arg5		
SEND-DCC6	arg1	arg2	arg3	arg4	arg5	arg6	

The above messages can send a packet with up to six bytes. With the evolving DCC standard, larger messages have been defined. The XPOM DCC messages are a good example. To send such a large DCC packet, it is decomposed into up to four LCS messages. The base station will assemble the DCC packet and then send it.

Table 3.13: RRaw DCC Packet Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
SEND-DCCM	ctrl	arg1	arg2	arg3	arg4		

3.10 DCC errors and status

Some DCC commands return an acknowledgment or an error for the outcome of a DCC subsystem request. The (DCC-ACK) and (DCC-ERR) messages are defined for this purpose.

Table 3.14: RRaw DCC Packet Management

Opcode	Data1	Data2	Data3	Data4	Data5	Data6	Data7
DCC-ACK							
DCC-ERR	code	arg1	arg2				

3.11 Analog Engines

The messages defined for the DCC locomotive session management as outlined above are also used for the analog engines. An analog engine will just like its digital counterpart have an allocated locomotive session and the speed/dir command is supported. All other commands will of course not be applicable. The speed/dir command will be sent out on the bus and whoever is in control of the track section where the analog engine is supposed to be, will manage that locomotive. In the following chapters we will answer the question of how exactly multiple analog engines can run on a layout.

3.12 Summary

The layout system is a system of nodes that talk to each other. At the heart are consequently messages. The message format is built upon an 8-byte message format that is suitable for the industry standard CAN bus. Although there are many other standards and communication protocols, the CAN bus is a widely used bus. Since all data is encoded in the message, there is no reason to select another communication media. But right now, it is CAN.

4 Message Protocols

This chapter will present how the messages presented in the previous chapter are used to form the protocols for layout configuration and operations. We begin with node management and port management. Next, the event system is described. Finally, the DCC locomotive and track management related commands and messages round up this chapter. The protocols are described as a set of high level messages flow from requestor to receiver and back.

4.1 Node startup

Node startup includes all the software steps to initialize local data structures, hardware components and whatever else the hardware module requires. To the layout system, the node needs to be uniquely identified across the layout. A configuration software will use the nodeId to manage the node. The (REQ-NID) and (REP-NID) messages are used to establish the nodeId on node startup. On startup the current nodeId stored in the module non-volatile memory is broadcasted. The (REQ-NID) message also contains the node UID. This unique identifier is created when the node is first initialized and all non-volatile data structures are built. The UID will not change until the node is explicitly re-initialized again.

After sending the (REQ-NID) message the node awaits the reply (REP-NID). The reply typically comes from a base station node or configuration software. In fact, any node can take on the role of assigning nodeIds. But a layout can only have one such node in charge of assigning nodeIds. The reply message contains the UID and the nodeId assigned. For a brand new module, this is will the node nodeId from now on.

Table 4.1: Node startup

node	base Station
REQ-NID (nodeId, nodeUID) ->	<- REP-NID (nodeId, nodeUID) or timeout

The nodeUID plays an important role to detect nodeId conflicts. If there are two modules with the same nodeId, the nodeUID is still different. A requesting node will check the (REP-NID) answer, comparing the nodeUID in the message to its own nodeUID. If the UID matches, the nodeId in the message will be the nodeId to set. Note that it can be the one already used, or a new nodeId. If the UIDs do not match, we have two nodes assigned the same nodeId. Both nodes will enter the collision and await manual resolution.

The above nodeId setup scheme requires the presence of a central node, such a base station, to validate and assign node identifiers. In addition, the nodeId can also be assigned by the firmware programmer and passed to the library setup routine. Once assigned, the node is accessible and the node number can be changed anytime later with the (SET-NID) command. All nodes are always able to detect a nodeId conflict. If two or more nodes have the same nodeId, each node will send an (NCOL) message and go into halted state, repeating the collision message. Manual intervention is required to resolve the conflict through explicitly assigning a new nodeId.

4.2 Switching between Modes

After node startup, a node normally enters the operation state. During configuration, certain commands are available and conversely some operational commands are disabled. A node is put into the respective mode with the (CFG) and (OPS) message command.

Table 4.2: Switching between Configuration and Operations mode

base Station	target node
CFG/OPS	-> <- ACK/ERR (nodeId) or timeout

4.3 Setting a new Node Id

A configuration tool can also set the node Id to a new value. This can only be done when the node is configuration mode. The following sequence of messages shows how the node is temporarily put into configuration mode for setting a new node Id.

Table 4.3: Switching between Configuration and Operations mode

Base Station	Node
CFG (nodeId)	-> node enters config mode <- ACK/ERR (nodeId) or timeout
SET-NID (nodeId, nodeUID) ->	<- ACK/ERR (nodeId) or timeout
OPS (nodeId)	-> node enters operations mode <- ACK/ERR (nodeId) or timeout

It is important to note that the assignment of a node Id through a configuration tool will not result in a potential node Id conflict resolution or detection. This is the responsibility of the configuration tool when using this command. The node Id, once assigned on one way or another, is the handle to address the node. There is of course an interest to not change these numbers every time a new hardware module is added to the layout.

4.4 Node Ping

Any node can ping any other node. The target node responds with an (ACK) message. If the nodeId is NIL, all nodes are requested to send an acknowledge (ACK). This command can be used to enumerate which nodes are out there. However, the receiver has to be able to handle the flood of (ACK) messages coming in.

Table 4.4: Node ping

requesting node	target node
PING	->
	<- ACK (nodeId) or timeout

4.5 Node and Port Reset

A node or individual port can be restarted. This command can be used in configuration as well as operations mode. The node or will perform a restart and initialize its state from the non-volatile memory. A port ID of zero will reset the node and all the ports on the node.

Table 4.5: Node and Port Reset

requesting node	target node
RES-NODE (npId, flags)	-> node or port is restarted
	<- ACK (nodeId) or timeout

4.6 Node and Port Access

A node can interact with any other node on the layout. The same is true for the ports on a node. Any port can be directly addressed. Node/port attributes and functions are addressed via items. The are reserved item numbers such as software version, nodeId, canId and configuration flags. Also, node or port attributes have an assigned item number range. Finally, there are reserved item numbers available for the firmware programmer.

The query node message specifies the target node and port attribute to retrieve from there. The reply node message will return the requested data.

Table 4.6: Node and Port Access

requesting node	target node
QRY-NODE (npId, item)	->
	<- REP-NODE (npId, item, arg1, arg2) or timeout if successful else (ERR)

A node can also modify a node/port attribute at another node. Obviously, not all attributes can be modified. For example, one cannot change the nodeId on the fly or change the software version of the node firmware. The (SET-NODE) command is used to modify the attributes that can be modified for nodes and ports. To indicate success, the target node replies by echoing the command sent.

Some item numbers refer to functions rather than attributes. In addition, all firmware programmer defined items are functions. The (REQ-NODE) message is used to send such a request, the (REP-NODE) is the reply message.

Table 4.7: Node and Port Access

requesting node	target node
SET-NODE (npId, item, val1, val2)	->
	<- ACK/ERR (npId) or timeout

Table 4.8: Node and Port Access

requesting node	target node
REQ-NODE (npId, item, arg1, arg2)	->
	<- REP-NODE (npId, item, arg1, arg2) if successful, else ACK/ERR (npId) or timeout

4.7 Layout Event management

Events play a key role in the layout control system. Nodes fire events and register their interest in events. Configuring events involves a couple of steps. The first step is to allocate a unique event Id. The number does not really matter other than it is unique for the entire layout. A good idea would be to have a scheme that partitions the event ID range, so events can be tracked and better managed. Consumer configuration is accomplished by adding entries to the event map. The target node needs to be told which port is interested in which event. A port can be interested in many events, an event can be assigned to many ports. Each combination will result in one event map entry. The (SET-NODE) command is used with the respective item number and item data.

Table 4.9: Layout Event management

requesting node	target node
SET-NODE (npId, item, arg1, arg2)	->
	<- REP-NODE (npId, item, arg1, arg2) if successful, else ACK/ERR (npId) or timeout

An entry can be removed with the remove an event map entry item in the (SET-NODE) message. Specifying a NIL portId in the messages, indicates that all eventId / portId combinations need to be processed. Adding an event with a NIL portID will result in adding the eventID to all ports, and removing an event with a NIL portID will result in removing all eventId / portID combinations with that eventId.

Producers are configured by assigning an eventId to broadcast for this event. The logic when to send is entirely up to the firmware implementation of the producer.

Even a small layout can already feature dozens of events. Event management is therefore best handled by a configuration tool, which will allocate an event number and use the defined LCS messages for setting the event map and port map entry variables on a target node.

Table 4.10: Layout Event management

requesting node	interested node
EVT-ON (npId, item, eventId)	-> receives an "ON" event
EVT-OFF (npId, item, eventId)	-> receives an "OFF" event
EVT (npId, item, eventId, val)	-> receives an event with an argument

4.8 General LCS Bus Management

General bus management messages are message such as (RESET), (BUS-ON), (BUS-OFF) and messages for acknowledgement of a request. While any node use the acknowledgement messages (ACK) and (NACK), resetting the system or turning the bus on and off are typically commands issued by the base station node. Here is an example for turning off the message communication. All nodes will enter a wait state for the bus to come up again.

Table 4.11: General LCS Bus Management

requesting node	any node
BUS-ON (npId, item, eventId)	-> nodes stop using the bus and wait for the (BUS-ON) command
BUS-OFF (npId, item, eventId)	-> nodes start using the bus again

4.9 DCC Track Management

DCC track management messages are commands sent by the base station such as turning the track power on or off. Any node can request such an operation by issuing the (TON) or (TOF) command.

Table 4.12: DCC Track Management

requesting node	any node
TON (npId)	-> nodes or an individual node/port for a track section execute the TON command
TOF (npId)	-> nodes or an individual node/port for a track section execute the TOF command

Another command is the emergency stop (ESTP). It follows the same logic. Any node can issue an emergency stop of all running equipment or an individual locomotive session. The base station, detecting such a request, issues the actual DCC emergency stop command.

In addition, LCS nodes that actually manage the track will have a set of node/port attributes for current consumptions, limits, and so on. They are accessed via the node info and control messages.

Table 4.13: DCC Track Management

requesting node	any node
ESTP(npId)	-> all engines on a node / port for a track section will enter emergency stop mode

4.10 Locomotive Session Management

Locomotive session management is concerned with running locomotives on the layout. The standard supported is the DCC standard. Locomotive session commands are translated by the base station to DCC commands and send to the tracks. To run locomotives, the base station node and the handheld nodes, or any other nodes issuing these commands, work together. First a session for the locomotive needs to be established.

Table 4.14: Locomotive Session Management

sending node	bae station node
REQ-LOC (locoAdr, flags)	->
	<- REP-LOC (sessionId, locoAdr, spDir, fn1, fn2, fn3)

When receiving a REQ-LOC message, the base station will allocate a session for locomotive with the loco DCC address. There are flags to indicate whether this should be a new session to establish or whether to take over an existing session. This way, a handheld can be disconnected and connected again, or another handheld can take over the locomotive or even share the same locomotive. Using the (REP-LOC) message, the base station will supply the handheld with locomotive address, type, speed, direction and initial function settings. Now, the locomotive is ready to be controlled.

Table 4.15: Locomotive Session Management

sending node	base station node
SET-LSPD(sId, spDir)	-> sends DCC packet to adjust speed and direction
SET-LMOD(sId, flags)	-> sends DCC packet to set session options
SET-LFON(sId, fNum)	-> sends DCC packet to set function Id value ON
SET-LFOF(sId, fNum)	-> sends DCC packet to set function Id value OFF
SET-FGRP(sId, sId, fGroup, data)	-> receives DCC packet to set the function group data
KEEP(sId)	-> base station keeps the session alive

The base station will receive these commands and generate the respective DCC packets according to the DCC standard. As explained a bit more in the base station chapter, the base station will run through the session list and for each locomotive produce the DCC packets.

Periodically, it needs to receive a (**KEEP**) message for the session in order to keep it alive. The handheld is required to send such a message or any other control message every 4 seconds.

Locomotives can run in consists. A freight train with a couple of locomotive at the front is very typical for American railroading. The base station supports the linking of several locomotives together into a consist, which is then managed just like a single loco session. The (**SET-LCON**) message allows to configure such consist.

Table 4.16: Locomotive Session Management

sending node	base station node
SET-LCON(sId, conId, flags)	-> send DCC packet to manage the consist

To build a consist, a consist session will be allocated. This is the same process as opening a session for a single locomotive using a short locomotive address. Next, each locomotive, previously already represented through a session, is added to the consist session. The flags define whether the locomotive is the head, the tail or in the middle. We also need to specify whether the is forward or backward facing within the consist.

4.11 Locomotive Configuration Management

Locomotives need to be configured as well. Modern decoders feature a myriad of options to set. Each decoder has a set of configuration variables, CV, to store information such as loco address, engine characteristics, sound options and so on. The configuration is accomplished either by sending DCC packets on a dedicated programming track or on the main track using with optional RailCom support. The base station will generate the DCC configuration packets for the programming track using the (**SET-CVS**), (**REQ-CVS**), (**REP-CVS**) commands. Each command uses a session Id, the CV Id, the mode and value to get and set. Two methods, accessing a byte or a single bit are supported. The decoder answers through a fluctuation in the power consumption to give a yes or no answer, according to the DCC standard. The base station has a detector for the answer.

Table 4.17: Locomotive Session Management

sending node	base station node
SET-CVS(cvId, mode, val)	-> validate session, send a DCC packet to set the CV value in a decoder on the prog track
REQ-CVS(cvId, mode, val)	-> validate session, send a DCC packet to request the CV value in the the decoder on the prog track <- REP-CVS(cvId, val) if successful or (ERR)

Programming on the main track is accomplished with the (**SET-CVM**) message. As there are more than one locomotive on the main track, programming commands can be send, but the answer cannot be received via a change in power consumption. One alternative for programming on the main track (**POM**, **XPOM**) is to use the RailCom communication standard. The base station and booster or block controller are required to generate a signal cutout period in the

DCC bit stream, which can be used by the locomotive decoders to send a datagram answer back. There is a separate section explaining this in more detail.

Table 4.18: Locomotive Session Management

sending node	base station node
SET-CVM(cvId, mode, val)	-> validate session, send a DCC packet to set the CV value in a decoder on the main track <- if not successful DCC-ERR

4.12 Configuration Management using RailCom

Instead of configuring engines and stationary decoders on the programming track, i.e. a separate track or just a cable to the decoder, configuring these devices on the main track would be a great asset to have. A key prerequisite for this to work is the support of receiving RailCom datagrams from the decoder.

??? ****note**** to be defined... we would need LCS messages to support this capability...
 ??? one message could be the channel one message of a RC detector...

4.13 DCC Accessory Decoder Management

The DCC stationary decoders are controlled with the (SET-BACC) and (SET-EACC) commands. A configuration/management tool and handhelds are typically the nodes that would issues these commands to the base station for generating the DCC packets. The following sequence shows how to send a command to the basic decoder.

Table 4.19: DCC Accessory Decoder Management

sending node	base station node
SET-BACC(accAdr, flags)	-> validate decoder address, send the DCC packet to the accessory decoder <- if not successful DCC-ERR

Since the layout control system uses the LCS bus for accessing accessories, these messages are just intended for completeness and perhaps on a small layout they are used for controlling a few stationary decoders. It is also an option to use a two wire cabling to all decoders to mimic a DCC track and send the packets for the decoders. On a larger layout however, the layout control system bus and the node/event scheme would rather be used.

4.14 Sending DCC packets

The base station is the hardware module that receives the LCS messages for configuring and running locomotives. The primary task is to produce DCC signals to send out to the track. In addition to controlling locomotives, the base station can also just send out raw DCC packets.

Table 4.20: Sending DCC packets

sending node		base station node
SEND-DCC3(arg1, arg2, arg3)	->	puts a 3 byte DCC packets on the track, just as is
SEND-DCC4(arg1, arg2, arg3, arg4)	->	puts a 4 byte DCC packets on the track, just as is
SEND-DCC5(arg1, arg2, arg3, arg4, arg5)	->	puts a 5 byte DCC packets on the track, just as is
SEND-DCC6(arg1, arg2, arg3, arg4, arg5, arg6)	->	puts a 6 byte DCC packets on the track, just as is

Sending a large DCC packet will use the ****SEND-DCCM**** message. The "ctrl" byte defines which part of the message is send. The base station will assemble the pieces and then issue the DCC packet.

Table 4.21: Sending DCC packets

sending node		base station node
SEND-DCCM(...)	->	puts a 3 byte DCC packets on the track, just as is

Again, as the DCC packets are sent out without further checking you better know the packet format by heart. Perhaps put the NMRA DCC specification under your pillow.

4.15 Summary

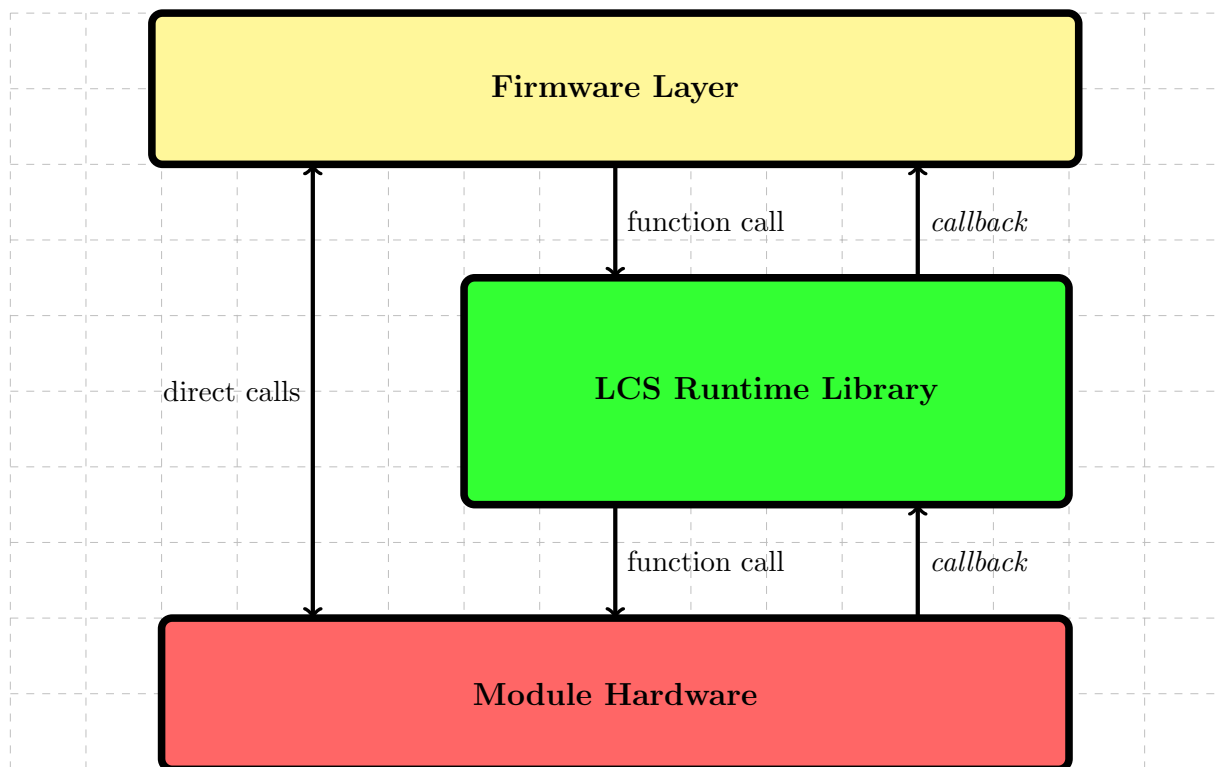
This chapter introduced the general message flow for the layout control bus functions. By now you should have a good idea how the system will work from a message flow between the nodes perspective. Most of the messages dealing with nodes, ports and events follow a request reply scheme using the nodeId as the target address. The DCC messages and protocols implicitly refer to nodes that implement base station and handheld functions. The base station is the only node that actually produces DCC packets to be sent to the track. However, any node implementing DCC functions can act on these messages. All message functions as well as functions to configure and manage nodes, ports and events are available for the firmware programmer through the ****LCS Runtime Library****. The next chapter will now concentrate on the library concepts and functions.

5 The LCS Runtime Library RtLib

Intended for the node firmware programmer, the LCS runtime library is the main interface to the hardware module. The library has methods for node and port configuration, event processing and layout control bus management. Most of the LCS bus management, node, port and port data management is performed transparently to the node firmware programmer. The library also provides convenience methods to send messages to other nodes and allows for a rich set of callback functions to be registered to act on messages and events.

The key design objective for the runtime library is to relieve the LCS nodes firmware programmer as much as possible from the details of running a firmware inside a hardware module. Rather than implementing the lower layers for storage and message processing at the firmware level, the runtime library will handle most of this processing transparently to the upper firmware layer. A small set of intuitive to use and easy to remember functions make up the core library. The library communicates back to the firmware layer via a set of defined callbacks. Throughout the next chapters, the library will be presented in considerable detail. Let's start with the high level view.

The following figure depicts the overall structure of a LCS hardware module and node. At the bottom is the hardware module, which contains the communication interfaces, the controller and the node specific functions. The core library offers a set of APIs and callbacks to the node firmware. The firmware programmer can perform functions such as sending a message or accessing a node attribute through the APIs provided. The library in turn communicates with the firmware solely via registered callbacks.



The firmware has of course also direct access to the hardware module capabilities. This is however outside the scope for the LCS core library. As we will see in the coming chapters, the library has a rich set of functions and does also perform many actions resulting from the protocol implementation transparently to the firmware programmer. It is one of the key ideas, that the firmware programmer can concentrate on the module design and not so much on the inner workings of the LCS layout system. Events, ports, nodes and attributes form a higher level foundation for writing LCS control system firmware. Not all of the functionality will of course be used by every node. A base station and a handheld cab control will for example make heavy use of the DCC commands. A turnout device node will use much more of the port and event system. Size and functions of the various library components can be configured for a node.

As a consequence, the library is not exactly a small veneer on top of the hardware and does take its program memory toll on controller storage. However, with the growing capabilities of modern controllers, this should not be a great limitation. The first working versions required an Arduino Atmega1284 alike version as the controller. The current working version is based on the Raspberry Pi Pico controller. More on the individual requirements and selection later.

The appendix contains the detailed description of all library interfaces. If a picture says more than a thousands words, an excerpt of the data declarations from the implementation says even more to the firmware programmer. At the risk of some minor differences on what is shown in the book and the actual firmware, you will find a lot of declarations directly taken from the "LcsRuntimeLib.h" include file.

6 RtLib Storage

All data of a LCS node is kept in volatile (MEM) and non-volatile (NVM). The data is structured into several data areas which we call `**map**`s. A map is a memory area which can be found in MEM and NVM or only in MEM. The key idea is that a map in MEM is initialized from its NVM counterpart at runtime start. Changes in a MEM map can be synced with its NVM map counterpart. There are also maps that do not have a NVM counterpart. These maps are initialized with default values defined for this map.

Maps do of course have a size. A port map for example will have a number of entries, one for each port. The design choice was whether all map sizes are configurable or rather a fixed size. The current design features a fixed size scheme. There are a few key reasons for this decision. First, there is no configuration need when initializing a node. Second, the total size even when generously sizing the maps is rather small compared to what the hardware can do. A node with 64 node attribute, 15 ports each of which also have 64 port attributes, an event map of 1024 events to manage and space for some miscellaneous data items will be around 8 Kbytes of data. A node with a 32K NVM chip still has plenty of space for user data. A raspberry Pi PICO has 264Kbytes of MEM, so also not an issue. Finally, with a fixed map layout, the NVM data can be copied in one swoop to a memory area on runtime start or reset.

This chapter presents a high level overview of the available maps and their purpose. Instead of painting many pictures, we will directly take code snippets from the runtime include files to show the data found in each map. Note that all maps are only accessible via runtime library routines.

6.1 Node Map

The node map is a node private data structure only accessible to the library firmware. It contains the information about the configured maps, the node options, `nodeId`, `canId` and other data such as the library version. When a node is initially created the configuration descriptor contains all the required information to set up a node map. Nodes need volatile and non-volatile storage. Our design implements a mirroring scheme. For the LCS storage there is a memory and an EEPROM version with the same layout. When a node is running the memory version is the storage to use for performance reasons. Also, it can be expected that the memory contents changes very often during operation. EEPROMs do have a limited number of writes in their lifetime and are not that performant for a write cycle. On the other the other hand the data is stored non-volatile. Information that needs to be changed and available across a restart is therefore synced from MEM to NVM. On restart, the NVM data is just copied to MEM. We always start with a defined state. The following figure shows the `nodeMap` data structure.

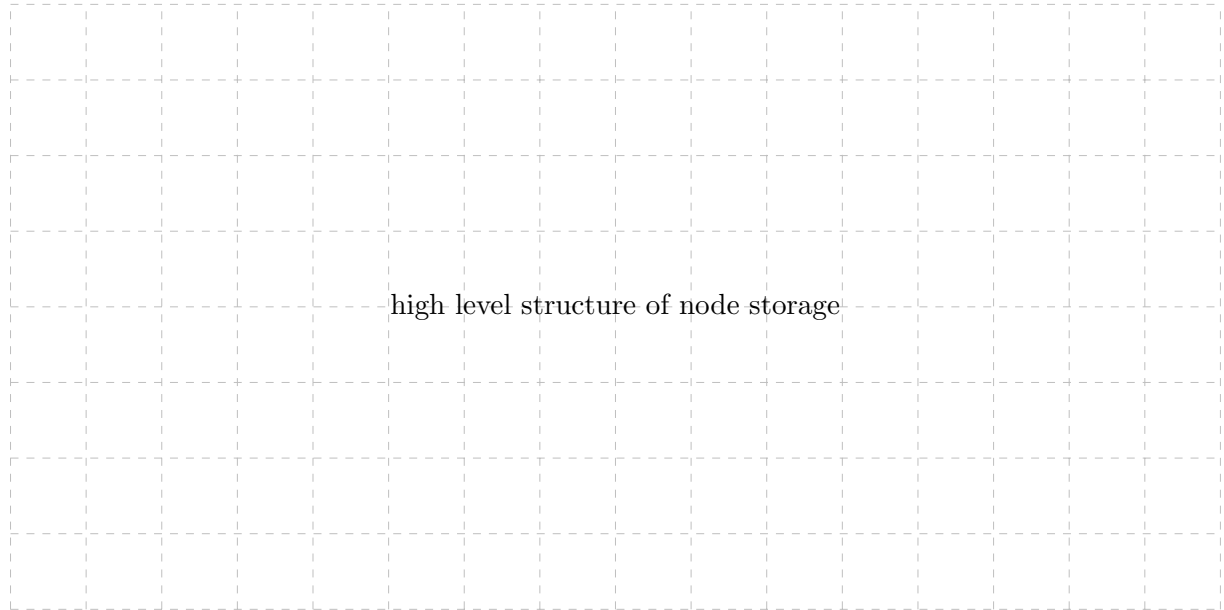


Table 6.1: Node Map NVM storage

Offset	Size	Purpose
0x0	0x200	Node Map. bla bla
0x200	0x200	CDC Map. bla bla
...	...	bla bla

Most of the data items deal with the location and entry sizes of the key maps. In addition, there are the nodeId, the node name, creation options, actual status flags and the set of node map attributes. Finally, the software version of the node version is kept here. For the firmware programmer there are methods to read from and write an item to the node. The library the **nodeGet**, **nodePut** and **nodeReq** routines offer a controlled access to the node map and other node data for node firmware programmers. They both use an item / value concept. Each routine passed an item Id for the data of interest and the data value. We will see an example later in this chapter. There are also three LCS messages, (QRY-NODE), (REP-NODE) and (SET-NODE) which allow for access from another node. Since these messages come from another node, there is also the option to register a callback for access control checks to node data before the operation is performed.

6.2 Port Map

The port map is an array of port map entries. The maximum number of ports are set through the node configuration descriptor values set by the firmware programmer. Changing the number of ports results in a node re-initialization, rebuilding the port map and all non-volatile port map data lost. During runtime there is a non-volatile and a memory version of this map. On node startup or reset, the non volatile port map entries are copied to their memory counterpart.

```

1 struct LcsPortMapEntry {
2
3
4 };

```

The port map entry contains flags that describe the port configuration options and the current operational setting. The event handling fields hold for an inbound port the current event received, the action and value as well as the a possible time delay before invoking the callback. For an outbound port the event fields describe the event to send when the condition for sending that event is encountered. The port map entries are located by just indexing into the port map.

The library **nodeGet**, **nodePut** and **nodeReq** routines presented before, offer a controlled access to the port map entry. The item and portId passed determine whether a node or port item is requested. Depending on the item, a portId of 0 will refer to all ports on the node or the node itself.

6.3 Node and Port Items

The term "item" came up numerous times by now. Nodes and ports features to access their attributes through an **item Id**. An item Id is just a number in the range from 1 to 255. Here is the definition from the library include file. The include file also contains the item numbers for the reserved node info and control items.

Table 6.2: Item ranges

Low	High	Purpose
0		NIL Item
1	63	Reserved items for node and ports
64	127	user defined items passed to the registered callback function
128	255	Node or Port Attributes

The first set of item numbers are reserved by the core library itself for node and port items that are standardized across all nodes. The range 64 to 127 and 128 to 191 describes the set of node or port attributes. The two groups actually represent the same attributes. For example the item number 64 refers to the same attributes as item 128 does. The difference is that the latter group also accesses the NVM storage. Items 192 to 255 are completely user defined. Using these numbers will just result in a callback invocation. Note that a callback can do anything. For example, turning a signal on or off could be an item Id of let's say 205 and sending a node control message with the item 205 and the value of 1 in the first argument would result in invoking a callback which implements how to turn the signal on. In short, a node supports variable access, comparable to the CV concept in DCC, and also a function call concept which allows a great flexibility for the firmware programmer.

6.4 Event Map

The event map is an array of event map entries, each containing the eventId that node is interested in and the port Id to inform when the event is encountered. The maximum number of event map entries is set through the node configuration descriptor values set by the firmware programmer. When a new node is configured, this value is used to construct the empty event map. Any change of this value results in a node re-initialization of the node, rebuilding the event map with all non-volatile event map data lost.

```

1 struct LcsEventMapEntry {
2
3     uint16_t    eventId;
4     uint16_t    portId;
5 };

```

??? explain the SYNC approach for this map...

Like all other maps, the event map is stored in two places. The non-volatile version of the eventMap is an array of event map entries. Whenever a new entry is added, a free entry is used to store this information. The memory version of the event map is a sorted version of all used non-volatile entries. The entries are first sorted by event Id. For entries with the same event Id, the port Id is then sorted in ascending order.

In addition to the search function, event map entries can be added and deleted by specifying the eventId and portId. EventMap entries can also be accessed by their position in the event map. This is necessary to read out the event map for example through a configuration tool. While reading an event map entry from the event map is supported in both node configuration and operation mode, deleting or adding an entry is only supported in node configuration mode.

6.5 User defined Maps

In addition to the runtime maps for node, ports, and events, the LCS runtime offers a user map for the firmware to use. This storage area is simply an unstructured array and the size depends on the capability of the node hardware NVM storage size. The area is the remaining storage available in the NVM chip array.

??? explain the concept and purpose ...

6.6 Periodic Task Map

```

1
2 User map \dots

```

6.7 Pending Request Map

The pending request map, is a small map that keeps track of outstanding reply messages to a previously issued message request. If a node sends a request, an entry is added to this map that indicates that a reply from another node is pending. When a reply messages is detected, the firmware callback is only invoked if this reply matches a previous request. This map is a volatile structure, a restart will clear all outstanding requests.

??? a timeout concept

6.8 Driver Function Map

```

1
2 ... code snippet here ...

```

6.9 Driver Map

for extension boards to be explained later...

```
1  
2 ... code snippet here ...
```

6.10 Summary

??? explain again why this NVM is key and thus important...

To summarize, node storage is organized in maps.

There is the node map, which is the global place for locating all other areas in the node. The port map contains the data for the configured ports. The event map is the mapping mechanism for events to ports. During node startup, the non-volatile data is copied to a newly allocated memory area. After initialization the node will only work from the memory area. All read and write operations use the memory storage area. When setting a value in any map, the flush option allows for setting its non-volatile counter part as well, so that we have a new initial value for the next restart.

Any change to the structure of the maps, for example changing the number of entries in a map, but also a different size of a data structure caused by a new library version, will result in a rebuilding of the non-volatile memory area with all previous data lost. The layout configuration data, such as the mapping of events to the node and port needs to be stored for example in a computer system so that can be reloaded once a node is re-created. A node has no way of keeping stored data across structural changes to its map layout.

7 RtLib Call Interface

??? this chapter needs to be reworked for new library call interface....

The LCS runtime library is the foundation for any module firmware written. The library presents to the firmware programmer a set of routines to configure, manage the LCS node and use the LCS functions, such as sending a message. This chapter will present the key functions used. We will look at library initialization, obtaining node information, controlling a node aspect, reacting to an event and sending message to other nodes. Refer to the appendix for a complete set of available LCS runtime functions.

7.1 Library initialization

The LCS runtime is initialized with the **init** routine. After successful runtime initialization, the firmware programmer can perform the registration of the callback functions needed, as well as doing other node specific initialization steps. This also includes the setup of the particular hardware. The subject of hardware setup will be discussed in a later chapter, "controller dependent code".

While there are many library functions to call, the only way for the library to communicate back to the module firmware when a message is received are the callbacks registered for. Callbacks will be described in the next chapter. A key task therefore is to register call back functions for all events and messages the node is interested. The following code fragment illustrates the basic library initialization.

```
1
2  code snippet here \dots
```

The final library call is a call to **run**. The run function processes the incoming LCS messages, manages the port event handling, reacts to console commands and finally invokes user defined callback functions. Being a loop, it will not return to the caller, but rather invoke the registered callback functions to interact with the node specific code. Before talking about the callback routines, let's have a look at the local functions available to the programmer to call functions in the core library.

7.2 Obtaining node information

Obtaining node or port information is an interface to query basic information about the node or port. A portID or NIL_PORT_ID will refer to the node, any other portID to a specific port on that node. The data is largely coming from the nodeMap and portMap data structures. The LCS library defines a set of data items that can be retrieved.

The return result is stored in one or two 16-bit variables and is request item specific. The nodeInfo and nodeControl routines allow for local access, the (QRY-NODE) and (REP-NODE) messages allow for remote access. The following example shows how the number of configured ports is retrieved from the nodeMap.

```
1
2  code snippet here \dots
```

7.3 Controlling a node aspect

Very similar to how we retrieve node data, the `nodeControl` routine allows for setting node attribute. A node attribute does not necessarily mean that there is a data value associated with the attribute. For example, turning on the "ready" LED is a control item defined for the `nodeControl` routine. There is a detailed routine description in the appendix that contains the items that are defined. The following example turns on the ready LED on the module hardware.

```
1
2  code snippet here \dots
```

The example shows that a node item is not only used to read or write a data item. It can also be used to execute a defined command, such as turning on an LED. In addition to the predefined node items, there is room for user defined items. In order to use them, a callback function that handles these items needs to be registered. This concept allows for a very flexible scheme how to interact with a node.

7.4 Controlling extension functions

```
// ??? the extension and driver stuff...
```

7.5 Reacting to events

```
// ??? rather a callback topic ?
```

7.6 Sending messages

Sending a message represent a large part of the available library functions. For each message defined in the protocol, there is a dedicated convenience function call, which will take in the input arguments and assemble the message buffer accordingly. As an example, the following code fragment will broadcast the ON event for event "200".

```
1
2  code snippet here \dots
```

All message sending routines follow the above calling scheme. The data buffer is assembled and out we go. Transparent to the node specific firmware, each message starts with a predefined messages priority. If there is send timeout, the priority will be raised and the message is sent again. If there is a send timeout at the highest priority level, a send error is reported.

7.7 Summary

A key part of the runtime library is the setup and manipulation of node and port data. A small comprehensive function set was presented in this chapter. That is all there is to invoke the core library functions. There are a few more functions that will be described in the chapters that deal with their purpose. For the other direction of information flow, i.e. the core library sends information back to the firmware layer, callback functions are used, presented in the following chapter.

8 RtLib Callbacks

One key idea in LCS library message processing is the idea of a callback method to interact with the node firmware. The library inner loop function will continuously check for incoming messages, command line inputs and other periodic work to do. Most of this work is handled by the core library code itself transparently to the node firmware. For example, reading a port attribute from another node is done without any user written firmware interaction. There are other messages though that require the node firmware interaction. As an example, consider an incoming event. We check that there is port interested and if so, invoke a callback with the message and port information to handle the event. The same applies to the console command line handler and the generic loop callback. Since the library has complete control over the processing loop, the callbacks are essential to invoke other periodic work. Depending on the callback type, it is invoked before the action is taken or afterwards. For example, switching from configuration mode to operations mode, will first perform the switch and then invoke the bus management callback routine if there was one defined.

8.1 General Callbacks

The general callback routine invokes the registered handler with messages that concern the general working of the node. Those are for example (RESET), (BUS_ON), (BUS_OFF), but also (ACK) and (ERR).

```
1 // ... the busMgt msg handler routine
2 void busMgtMsgHandler( uint8_t *msgBuf ) {
3     //... handle the cases of busMgt messages
4 }
5 ...
6 // during module firmware initialization ...
7 lcsLib -> registerMsgHandler( busMgtMsgHandler )
```

8.2 Node and Port Initialization Callback

Once the library is initialized the various handlers can be registered and all other firmware specific initialization can be done. The last step is the call to the ****run**** method, which will never return. The very first thing the ****run**** method does after some internal setup is to invoke the node and port initialization callback if registered. The callbacks are also invoked whenever a node is restarted with the (RES-NODE) command or the (RESET) command for nodes and ports. The following code snippet shows how to register such a callback.

```
1 // ... the node init msg handler routine
2 void nodeInitHandler( uint16_t nodeId ) { ... }
3 ...
4 // during module firmware initialization ...
5 lcsLib -> registerInitCallback( NIL_PORT_ID, nodeInitHandler )
```

Note that a portID or NIL_PORT_ID will refer to the node. Registering an initialization callback from a port will just pass a non-nil portId instead. The port init callbacks are invoked in ascending portId order.

8.3 Node and Port Request Reply Callback

Node and port attributes can be queried from other nodes. The reply from sending a (QRY-NODE) command to the target node, the (REP-NODE) message, is passed back to the requesting firmware through the node request callback.

```

1 // ... the node query handler routine
2 void nodeReqHandler(    uint16_t nodeId,
3                        uint8_t portId,
4                        uint8_t item,
5                        uint16_t val1,
6                        uint16_t val2 ) { ... }
7 ...
8 // during module firmware initialization ...
9 lcsLib -> registerReqRepCallback( nodeReqHandler );

```

The callback returns in addition to the arguments, the node and port ID of the replying node. Again, a portId of NIL_PORT_ID refers to a node item answer.

8.4 Node and Port Control and Info Callback

The nodeControl and nodeInfo routines offer callbacks for user defined items. There is a callback function for user defined control items and one for the info items.

```

1 uint8_t ( *infoHandler ) ( uint8_t portId,
2                            uint8_t item,
3                            uint16_t *arg1,
4                            uint16_t *arg2 ) { ... }
5
6 uint8_t ( *ctrlHandler ) ( uint8_t portId,
7                            uint8_t item,
8                            uint16_t arg1,
9                            uint16_t arg2 ) { ... }
10 ...
11 // during module firmware initialization ...
12 lcsLib -> registerInfoCallback( portId, infoHandler );
13 lcsLib -> registerCtrlCallback( portId, ctrlHandler );

```

All the callback routines return a status code. When the item is not found or the arguments are not valid, the callback should return an error code. Any other status than ALL_OK is passed back to the caller as the result of the nodeInfo or nodeControl method.

8.5 Inbound Event Callback

The event callback function is invoked when an event was received and the node has an inbound port that is interested in the event. The eventId / portId was previously configured in the event map. A port reaction to the incoming event can be configured to have a delay between the receipt of the event and the actual invocation of the port event callback routine. The callback function is passed the actual event information.

```

1 // ... the inbound event handler routine
2 void eventHandler ( uint16_t nodeId,
3                    uint8_t portId,
4                    uint8_t eAction,
5                    uint16_t eId,
6                    uint16_t eData ) { ... }

```

```

7  ...
8  // during module firmware initialization ...
9  lcsLib -> registerPortEventCallback( eventHandler )

```

If there is more than one port configured to react on the the incoming event, they are invoked in ascending order of portIds. The `***eAction***` parameter specifies whether the event is a simple ON/OFF event or a generic event with optional associated data. Note that only ports can react to events.

8.6 Console Command Line Callback

The LCS library implements a console command interface. Although not typically used during normal operations, it is very handy for tracking down firmware problems during development. Furthermore, troubleshooting in a layout is a good reason for having such an interface. As we will see in the hardware section, a simple serial data line or even an USB connector can be part of the module hardware. Simply connecting a computer to the node allows to query and control the node. Note, that this is also to some degree possible using the LCS bus messages.

In addition to the serial commands defined for the LCS core library, the firmware programmer can implement an additional command interface. Any command not recognizes by the library is passed to the registered command line callback. The callback itself returns a status code about the successful command execution. Any status other than ALL-OK will result in an error message listed to the serial command device connected.

```

1  // ... the command line handler routine
2  uint8_t commandLineHandler( char *line ) { ... }
3  ...
4  // during module firmware initialization ...
5  lcsLib -> registerCommandCallback( commandLineHandler )

```

Why implementing a serial command handler on top of the core library serial commands? The key reason is that a firmware programmer can add additional commands for firmware specific commands. Other than further debug and status commands, nodes such as the base station can implement an entire set of their own commands. A good example is our base station, which implements most of the DCC++ serial command set. Configuring a DCC locomotive decoder can then be handled with decoder programming software such as the JMRI DecoderPro tool, which in turn issues DCC++ commands as one option.

8.7 DCC Message Callback

The LCS Library defines a set of DCC related LCS messages to configure and operate the running equipment and track. These messages are typically used by cab handhelds and the base station, which is in charge to produce the DCC signals for the tracks. The DCC message callbacks are used to communicate these messages to the node firmware. The callback routines are all passed the message buffer. The following code snippet shows the declaration for a DCC type callback.

```

1  // ... the DCC message handler routine for DCC messages
2  void dccMsgHandler( uint8_t *msg ) { ... }
3  ...
4  // during module firmware initialization ...
5  lcsLib -> registerDccMsgCallback( dccTrackMsgHandler )

```

8.8 RailCom Message Callback

Railcom is a concept for the DCC decoders to communicate back. DCC is inherently a broadcast protocol just like a radio station. There was no way to communicate back. Railcom was design to allow for a decoder to send back data when the DCC channel is told to "pause". The chapter on the DCC subsystem will explain DCC and RailCom in greater detail. The Railcom Message callback is the function callback that will be invoked when a RailCom Messages is received.

```

1 // ... the Railcom message handler routine for DCC messages
2 void railComMsgHandler( uint8_t *msg ) { ... }
3 ...
4 // during module firmware initialization ...
5 lcsLib -> registerRailComMsgCallback( dccTrackMsgHandler )

```

8.9 LCS Periodic Task Callback

The LCS core library attempts to handle as much as possible of message and event processing transparent to the user developed firmware. The core library `***run***` method, called last in the firmware setup sequence, will do the internal housekeeping and periodically scan for messages and serial commands. In addition, the run loop will also handle periodic activities outside the library. For example, a booster needs to periodically monitor the current consumption. The library therefore offers a callback registration function for periodic tasks. The example shown below registers a task to be executed every 1000 milliseconds.

```

1 // ... a periodic task to be registered
2 void aTask( ) { ... }
3 ...
4 // during module firmware initialization ...
5 lcsLib -> registerPeriodicTask( aTask, 1000 );

```

The runtime library `***run***` routine never returns. All interaction between the library is done through previously registered callbacks and calls to the library from within those callbacks. It is also important to realize that a callback runs to completion. In other words, the library inner working is put on hold when executing a callback. For example, no further LCS messages are processed during callback execution. The same is true for the periodic tasks. It also means that one cannot rely on exact timing. Specifying for example a 1000 milliseconds time interval, could mean that the task is invoked later because of other tasks running for a longer period. A periodic task would however not run earlier than the specific interval. In summary, callback routines should therefore be short, quick and mist of all non-blocking.

Putting the library inner working on hold is however not true for functions that react on hardware interrupts. If there are interrupt routines for let's say a hardware timer, they will of course continue to take place. As we will see in the DCC track signal generation part of the base station, the interrupt driven signal generation is not impacted. Nevertheless, a firmware programmer needs to be aware that the order of callback invocation is fixed and that a callback runs to completion.

8.10 Summary

LCS callbacks are a fundamental concept in the core library. A firmware designer will write code that uses the core library functions to access the lower layers and callback functions that are invoked by the library to communicate back. Well, that is all there is a the core layer.

CHAPTER 8. RTLIB CALLBACKS

Other than functions and callbacks, how can you access the library ? Wouldn't it be nice to have a simple interface to access the node data, set some options and simply test new hardware ? That is the subject of the next chapter.

9 RtLib Command Interface

??? explain the general concept ...

The primary communication method of the layout control system are LCS messages sent via the bus. In addition, each module that offers an USB connector or the serial I/O connector, implements also the serial command console interface. The interface is intended for testing and tracing purposes. LCS console commands are entered through the hardware module serial interface.

Perhaps the most important command is the help command, which lists all available command and their basic syntax.

```
1 <!?>
2 <#?>
```

Any command not recognized is passed to a command line handler....

`!lcs-command-char [arguments] ;`

will be passed to the registered command call back function, if there is one registered. The following summary shows the available LCS serial commands. The appendix contains a detailed description of of the commands implemented by the LCS library.

9.1 Configuration Mode Commands

The configuration mode commands will place a node into either operations or configuration mode.

9.2 Event Commands

Event commands work with the event map. They add and remove an event, search the map for an event/port pair, or locally send an event to the node itself to test the event handling and so on.

9.3 Node Map and Attributes Commands

The node map and attribute map will examine and modify these maps.

9.4 Send a raw Message

For testing the message send mechanism, a command is available to send a raw data packet via the LCS bus.

9.5 List node status

The "s" command will list a great detail on the node data. When debugging a node problem, this is perhaps the most useful command to see what is store locally.

9.6 Driver commands

What about the "xxx" commands? Well, they are used issue commands to the hardware drivers. We have not talked about them so far. This topic is presented when we know more about how the hardware is structured. Stay tuned.

9.7 LCS message text format

Just like the LCS core library accepts simple ASCII command strings, the LCS messages can also be transmitted as an ASCII text line. This is very useful for building communication gateways that transmit the message via another medium, such as an ethernet channel. There is a simple scheme for the ASCII representation of the message:

The message is enclosed in the "i" and "j" delimiters and the first character is the "xxx" sign. Up to 8 hexadecimal values written as "0xdd" follow, where "d" is a hexadecimal digit.

Note: to be implemented. Perhaps to simple library routines to create an ASCII version of a LCS message and convert an ASCII string to an LCS message.

9.8 Summary

The command line interface provides a way to interact with a node at the command line level. This is very useful for initial testing new hardware and software debugging. All that is needed is a USB interface and a computer. As we will see in the main controller chapter, a USB or serial interface is also necessary for downloading new firmware to the boards. Besides that, this interface is normally not used during regular operations.

10 RtLib Usage Example

??? what is a good comprehensive example ?

??? use the diagnostic program to show how to write a program and at the same show how we actually test the runtime.

11 The DCC Subsystem

The LCS runtime library builds the software foundation for implementing the layout control software. So far we have discussed the general working, node and port functions and callbacks. One part that was only touched upon briefly so far is the digital command control (DCC) subsystem. A significant part of the LCS messages deal with the control of running equipment decoder, stationary decoders and the track itself.

This chapter now dives a little deeper into the DCC subsystem. At the heart of this subsystem is the base station node that is in charge for of managing locomotives and tracks. It receives LCS messages from devices such as a cab throttle and translates these commands into a series of DCC packets. The packets are the basis for the DCC track power modules to actually produce the electrical signals on the track. The power module is either a part of the base station or a separate booster. Base station, boosters and throttles are just nodes making use of the DCC commands in the LCS message set. They too can implement reacting to events and send themselves events. First we will look at a base station and what it takes to manage a locomotive session and to generate the DCC packets for mobile and stationary decoders. Next, we will look into how a DCC packet actually gets out on the track.

11.1 Locomotive session management

Digital locomotives are equipped with a mobile decoder. The decoder will analyze the DCC packets on the track and if addressed perform the desired function. For each active locomotive the base station first establishes a locomotive session. Across the layout, a locomotive is uniquely identified by its `**cabId**`. In DCC terms this is the address of the locomotive. The DCC standard defines an address range that all decoders, mobile and stationary, share. Once a session is established for the cabId, the base station accepts LCS DCC commands, such as setting the speed, direction or a function, and produce the corresponding DCC packet. We will see later what happens to the packet.

A base station typically works with two DCC tracks. There is the `**main track**`, which consist of all the track sections of the layout. Commands such as setting a locomotive speed and direction, refer to this track. In addition, there is a `**service track**` which is used to configure an individual locomotive. This track is electrically separated from the main track. However, when it comes to packet transmission, the two tracks are very similar. For the base station functionality there are thus two key functional components. The first is the locomotive session management, the second is the programming of a locomotive mobile decoder. The programming track commands do not need a cabId, i.e. address, as there should only be one locomotive on this track. This has to do with the way a decoder replies the base station and will be discussed when we talk about decoder programming.

11.2 Stationary Decoders

While mobile decoders can be found in a locomotive, a stationary decoder can be found somewhere on the layout. For example, a stationary decoder that is close to a set of turnouts. It is connected to the main track and just like its mobile cousin decodes the DCC packets. Stationary decoders, called accessories in the NMRA standard, are assigned to a part of the address

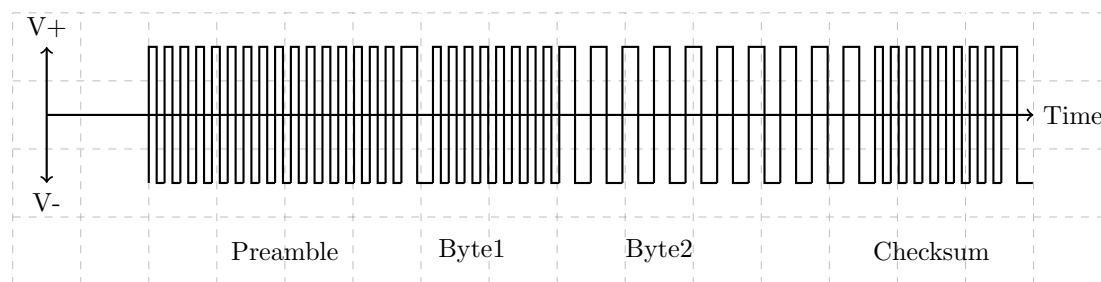
range and react to their configured address. The base station accepts LCS commands for such a decoder and generates the DCC packets for it.

As said before, the trend is to use a layout control system with a dedicated bus for the layout components. The key idea is to offload the track where the engines run from the packets for the accessories. Another approach is to have a dedicated wire to all accessory decoders and send the DCC packets on this. In a sense another track without locomotives. Our layout control system will support generating the stationary decoders packets and send them via the main track. But the feature is only implemented for completeness. Maybe there is still one old decoders that is put to use this way. Our layout will be controlled by the LCS bus.

11.3 DCC packet generation

The key task of the locomotive session management is to generate the DCC packets for running and configuring mobile and stationary decoders. There are also packets, such as RESET or IDLE, that concern all decoders on the track. The DCC packets are described officially in the NMRA specifications. The *RailCommunity* specification documents (RCN-xxx) also have an excellent description of the packets layout and their interpretation. Each bit is either a zero or a one. A "one" bit has a period of 116 microseconds, a zero bit a period of 232 microseconds. The exact timings are listed on the DCC standard, for now, this is a good enough description. The appendix contains links to their web pages for diving into all the details of the DCC packet format and protocol.

The base station part that produces DCC packets is not concerned with how these packets are actually transmitted to the locomotive. This is the task of the DCC track management component, which will be presented shortly. In general, a DCC packet is a stream of bits consisting of the preamble, a decoder address and the command bytes followed by a checksum byte. The preamble is to sync a decoder with the upcoming data stream. The address tells which decoder is address and the command bytes actually tell what needs to be done. Finally, the checksum makes sure that there was no error in transmitting the packet. The following figure depicts a simple packet.



The high level LCS DCC commands are translated by the base station into the corresponding DCC packets. There are two modes of transmission. With the first mode, any incoming command is translated and sent out immediately with an optional repeat count. Consider a locomotive speed stop command. This has of course top priority. The second mode of transmission is a one time fixed sequence of DCC packets for a high level LCS command, such as it is used for programming a decoder.

When no command is pending, the base station will loop through all active session entries and send packets for refreshing the previously sent commands. For example, after sending a speed/direction command, this command will be repeated periodically, until a new command is issued for this locomotive session. While looping through the session table, only a part of

the necessary refresh packets are generated to make sure that all engines get a fair share of the track bandwidth in time. The complete refresh of speed/direction and function keys are spread over a couple of loop iterations. The DCC standard makes recommendations what data to send out how often or periodically. Time to discuss how the DCC packets actually get to the track.

11.4 Sending a DCC packet

The DCC track management software component does not store any DCC packets other than the active packet that is currently being transmitted and the pending next packet. If it is busy with sending a packet and there is already a pending packet queued, the packet loading routine in the locomotive session management component is waiting until the pending packet becomes the current packet and then the next packet is queued. There is one more scenario to address. Suppose there is no packet currently sent from the locomotive management and thus there is no packet to send to the track. In this case, we cannot just stop sending packets, as the locomotives draw their track power from the track signal. DCC track management signal generation then just "invents" a packet to send out. This is the DCC IDLE packet for the main track and the DCC RESET packet for the programming track.

11.5 DCC Track Signal Generation

The primary task of a DCC track signal generator is to receive the DCC packets generated by the base station producing the hardware signals for the packet bits on the track. The other task is to monitor the power consumption and the optional RailCom channel communication. DCC signals are square wave signals with a defined duty cycle period. A duty cycle of 58 microseconds represents a "DCC one", a duty cycle of 116 microseconds a "DCC zero" bit. This signal is sent to the track by reversing the polarity of the two tracks lanes with the respective timing. Typically, a H-Bridge such as found in motor drivers will perform this task. If the H-Bridge is enabled, sending a "DCC One" will mean to set the digital input signals for the H-Bridge to enable the "+" direction, and then reverse the digital signals for the "-" direction. The H-Bridge hardware essentially reverses the track polarity accordingly to digital series and ones. The DCC packet is broken down, bit by bit and the digital signal is produced. That's it, we have a nice signal on the track. How exactly the base station does the digital signal output generation is discussed in more detail in the base station chapter.

11.6 Power consumption monitoring

DCC track management is also responsible for continuously monitoring the track power consumption. Considering that boosters can emit several Amps a short circuit for a longer time will certainly damage track and running equipment. It is therefore paramount to monitor the actual current consumption very closely. Monitoring track power consumption can be done by measuring the voltage drop over a shunt resistor in serial with the H-Bridge. The controller analog input will periodically read the value and process the incoming data. From a software perspective there are a couple of ways when to measure the voltage and how to process it. One way is to measure at defined spots in the bitstream.

During the signal generation, the track power current consumption will be measured at defined spots in the bit stream. A zero bit in a packet is a good place. The hardware just need to make sure that the measurement completes during the 116us half cycle of the zero bit. But certainly, there are other ways of measuring. When exceeding the configured consumption limit,

it is stored in a node variable, DCC track management will broadcast a power overload event and shut down the track. After a configured time a restart is attempted. If the restarting fails for several times, the track is powered down permanently and manual intervention is required.

In addition, care needs to be taken to report a power consumption value that reflects the consumption over a period of time. Most locomotive decoder use a PWM (pulse width modulation) approach to drive the motor in the engine. Depending on when the current consumption measurement takes place a high level value or a zero value is returned. This does of course not reflect the actual power consumption. Therefore, several values sampled need to be used to build the "root mean square" value to indicate the actual power consumption.

11.7 Decoder programming support

There it is. A new locomotive unpacked, sitting on the programming track. At a minimum it needs to be told what its locomotive address will be on our layout. This task is accomplished by writing values to the decoder CV variables. A short locomotive address for example is a writing of this address to CV 1.

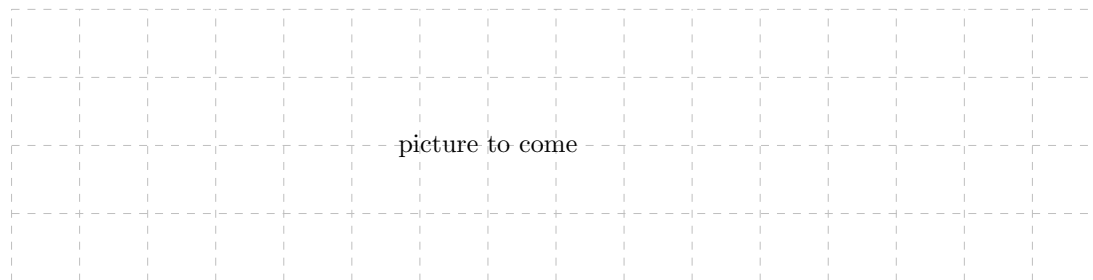
DCC is a broadcasting protocol. Just like a radio station, you can send but not receive. In order to communicate back the decoder raises its consumption power for specific value and time period to indicate an OK. DCC track management needs to be able to detect this consumption power fluctuation on the programming track. The detection is very similar to the previously discussed power consumption monitoring except that is done in two steps. Before accessing a CV variable, the current decoder power consumption is measured to establish a base line. This base line is then compared with the actual power consumption after the CV access. A fluctuation for the value and time specified by the DCC standard is considered a positive answer.

Reading all CV variables from a sophisticated decoder can easily take several minutes this way. Furthermore this communication will not work on the main track, as there are many locomotives running, making it impossible to detect the raise in power consumption of a single locomotive. There had to be a better way and there is. And there is. It is RailCom.

11.8 RailCom support

RailCom was invented to address the problem of effective back communication on the programming track and also on the main track. DCC track management needs to implemented the basic mechanism for this kind of communication. As the DCC is a broadcasting protocol, no other transmission is possible while it is broadcasting. The key idea of RailCom is to briefly turn off the DCC communication and use this moment of quiescence to transmit back data from the decoder. The period of short circuiting the DCC track is called the cutout period. In addition to to generating the DCC zeroes and ones on the track, DCC track management is also implementing the cutout support.

The following figure depicts the overall signal timing for RailCom support. All the details can be found in the NMRA and RailCommunity standard document including a hardware reference implementation for a RailCom decoder and detector. After the last bit of a packet and during the first bits of the DCC packet preamble, the track signal is turned off, the track is short circuited. The decoder can now send out data to the track and a signal detector can receive that data. The signal is a simple serial signal with a baud rate of 250 Kbits. The following figure shows the overall DCC and RailCom signal timing.



The NMRA and RailCommunity standards describe the data format used when sending the RailCom data. There are two channels defined which in total send a maximum 8 bytes during the cutout period. Channel one takes up two bytes, channel two takes up four bytes. To ensure data transmission integrity, the bytes itself are encoded as values with four bits one and four bits zero. This leaves 64 useful values that the byte contains. All else is an invalid data byte. Put together, there are up to 48 bits of data in a RailCom message.

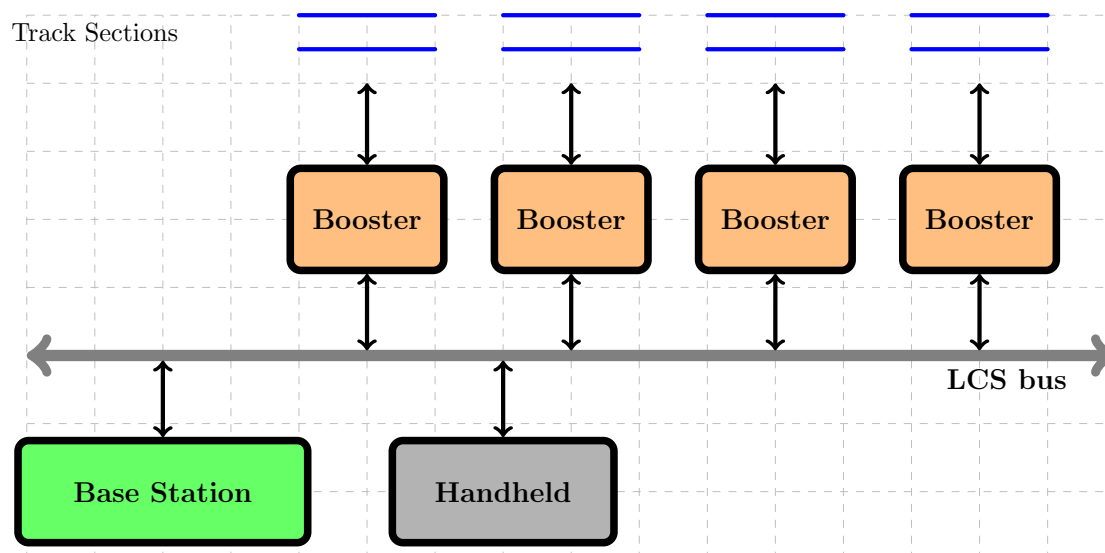
The individual messages available in channel one and two are called datagram. For channel one, a datagram is 12bits, i.e. the six bits encoded in the two raw data bytes, for channel two there are in total 36 bits. Each datagram starts with a four bit identifier followed by the payload. A decoder is required to transmit its address every time it is addressed on channel one. Decoders will send data on channel two only if explicitly requested. This leaves channel one with a bit of chaos more than one decoder transmits. There are options to tell the decoder to stop sending its ID after an initial couple of times.

Channel two is only used when the decoder is explicitly addressed via an POM or XPOM DCC packet. Still, the base station needs to ensure that multiple requests from different encoders are transmitted one at a time and there is enough time for the addressed decoder to answer. Also, the decoder needs to be addressed at least twice to complete a data request via RailCom. The first DCC packet tells what to get, the second DCC packet gives the controller a chance to put the RailCom reply in the next cutout packet. Finally, the DCC-A (RCN218) standard uses the RailCom infrastructure for automatic locomotive registration and fast access to the information in the decoder. For this purpose, channel one and two are combined to a 48bits payload data. More on these topics in the base station chapter.

11.9 DCC Track sections

A base station may have a powerful main track and a less powerful programming track. For smaller layouts this is a typical scenario. In fact, the DCC standard requires for the programming track to limit the maximum current to 100mA after initialization to avoid any decoder damage from misconfiguration when testing a new hardware. Larger layouts however are typically divided into several sections each of which is controlled by a DCC booster. This has the key benefit that a short circuit will only affect a track section. A DCC booster can also be equipped with a RailCom detector to implement for example locomotive detection on a per section basis.

To the DCC track management in a base station a booster managing a track section is largely transparent. All track management is concerned with is that the DCC signals are generated. A base station for a larger layout could just have two H-Bridges with a low current rating. One would produce the DCC signal for the main track, the other for the programming track. The programming track output is directly connected to the programming track. The main track output of the base station however is just a signal line that is then fed via the LCS bus data lines into the booster. All track sections will receive the same DCC signal. All boosters are required to be wired with the same track polarity.



All boosters will measure the power consumption continuously and in the case of exceeding the limits, send an event that the base station is interested in. Boosters are just LCS nodes like anything else. Port variables and events are the mechanism of communication. The actual implementation of a booster with variables and events are described in the hardware module chapter on boosters.

There is one more thing to take care of. If a layout consists of more than one track section there is the situation that the two boosters are not in close sync with respect to polarity and signal generation timing. Again, it is first of all very important that all boosters have a common polarity wiring. If not, short circuits caused by running equipment crossing from one section to the other are likely to happen. If RailCom is enabled, the cutout period acts as a short circuit of one section as well. If one booster section is in cutout mode and the adjacent booster not yet, crossing rolling equipment would effectively short circuit the active booster. To avoid this problem, boosters need not only be in close sync, they also should feature a kind of "security gap" period before starting the cutout period. In this period the booster is put into disconnected mode. This topic is also discussed a bit more in the booster hardware part.

11.10 A short Glimpse at Software Implementation

The DCC base station plays the key role in the DCC subsystem. In addition to manage the locomotive sessions and generating the necessary DCC packets, it is also responsible to manage the two tracks MAIN and PROG. Built on top of the LCS library, the base station will have two key software components, one for session management and one for track management. The session management part is rather straightforward, a table of active locomotive sessions that are processed periodically. The track management part is by nature very close to the hardware. Two interlinked state machines, one for track signal generation and one for track power management build the core of track management. The actual implementation of the two key parts of the base station module is described in more detail in the base station chapter.

11.11 Summary

This chapter gave a high level overview on the DCC subsystem. The base station and booster firmware implement the DCC decoder management and track signal generation. Locomotive session management is concerned with managing the running equipment. The key concept is the

session, which contains all data needed to control a locomotive on the track. DCC Packets for all active locomotives are generated and sent to the track management component and thereafter periodically refreshed. Programming a locomotive decoder sends a DCC packet sequence which the decoder addressed interprets. There are two tracks, the main track and the programming track. While they are different in what they are used for and what hardware capacities they need, both will just as their key function putting out the packets generated by the locomotive session management software.

DCC track management is responsible for the track signal generation and track power management. It takes the DCC packets and sends them out bit by bit. First the preamble and optional cutout period then each data byte of the packet. The track consumption power is monitored for the main track and also used for the programming track decoder acknowledge power consumption fluctuation. Exceeding the configured power consumption limits will result in a shutdown of the signal followed by a number of restarts. The DCC signals produced by the based station are ready to be used and can directly be fed to a track. However, in larger layouts, there will track sections with a DCC boosters for each section. Base station and boosters are, you guessed it, just nodes on the LCS Bus.

12 The Analog Subsystem

Analog? Yes, there is analog. Although the Layout Control System is a digital system with locomotives controlled via DCC, there are cases where implementing a layout based on controlling all rolling stock via DCC would mean to equip all your analog running engines with DCC decoders. Besides that it represents quite a considerable cost and converting some older locomotives is a real project in itself. Also, there are model railroad clubs with literally hundreds of locomotives. These layouts are analog and you will find miles of cables to a central control station. Converting all of the existing infrastructure in one swoop represents a considerable cost.

This chapter presents an overview for a subsystem managing analog locomotives. We will only focus on analog running equipment. Devices such as signals, turnouts and other stationary equipment is managed with the LCS node, port, event system, i.e. digital. This chapter will introduce extensions required to manage an analog and also a potentially hybrid layout.

12.1 Requirements

The first major difference to a DCC based system is that for a given track section there can only be one locomotive or consist. In contrast to a digital signal with a permanent flow of the square wave signal, an analog system will use a pulse width modulated (PWM) approach. A wider pulse width will make the engines run faster, a smaller pulse width makes it run slower. The signal contains no information about the actual engine and just delivers power corresponding to speed desired to the track section.

To still run several engines, the layout needs to be divided into several sections or blocks. Just as we divided a layout into sections with separate DCC boosters, analog layouts will control sections with a separate power module. It is not necessary to have a power module for each section, but the more sections and power modules the more analog engines could run simultaneously. Built on top, an analog layout often has a concept of blocks to run trains automated managed by a block signal control system. The blocks are often divided further into subsections and there are track occupancy detectors to know where the loco is within the block.

Just like their DCC cousin, analog track sections need special consideration when an engine is moving from one block to the next. For DCC track sections, each having a booster receiving the same DCC signal from the base station, there is a short window of power disconnect to address any small booster timing differences. For the analog world, the current section and the following one need to be also in sync for the engine to cross from one block to the next. The PWM signals, which deliver the power to the locomotive, need to be synchronized in order to avoid that one block still delivers power and the next block not quite yet. A locomotive would not run smoothly in that case.

An analog track block would also need to know the actual engine characteristics of the engine in a block. Each engine has different power consumption characteristics, so the speed is a function of engine type and actual train load. This track control mechanism is very closely associated with an overall block signaling control system. In fact, an analog system almost every time has a block signaling system implemented to manage several engines. Note that in an analog the smarts must be in the block controller and not in the engine. In a DCC system, the smarts is in the engine. In A DCC system the booster are there to address the overall power needed, dividing a layout into several power sections. In an analog system, the sections

are there to address the need to run many engine simultaneously. There can only be one engine in a track sections or block.

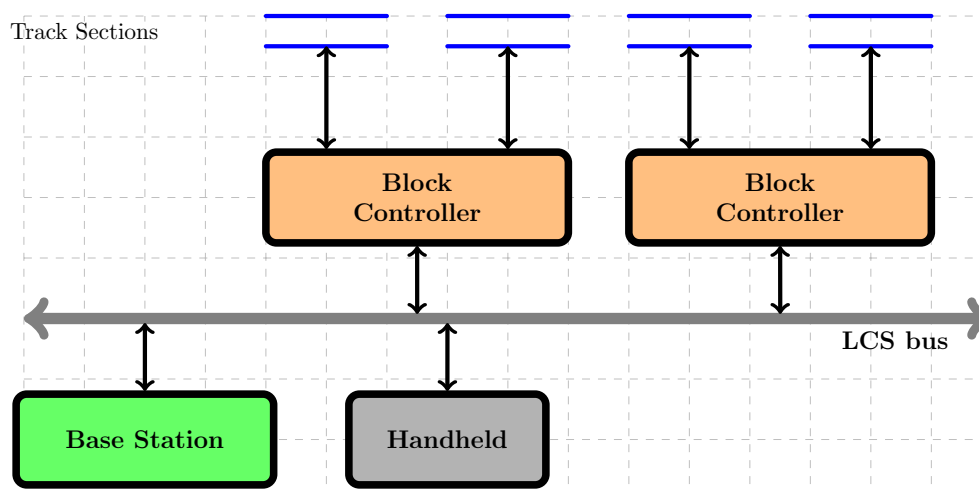
There is also the situation that a layout is in a transition from analog to digital. Wouldn't it be nice to manage both worlds in the same layout? A block could either host a DCC equipped locomotive or an analog locomotive, but never at the same time. Also, it needs to be ensured that the follow-on block where the engine is heading is of the right kind. This brings up several more design questions, and we will talk about it in the following sections.

In any case, an analog system also needs a means of a cab handheld to control the engine. Following the LCS overall concept, the communication between the cab handheld and the layout nodes that ultimately control the engine, is digital. The concept of a locomotive session and a base station that manages all active sessions supports both DCC as well as analog engines. The base station managing locomotive sessions would need to be enhanced slightly to also support analog running equipment. Of course the cab handheld for an analog locomotive is much simpler. All that is needed is the direction and speed control.

12.2 Overall concept

Before diving into details, this section shows how support for an analog system could be implemented. There is the basis station managing all active locomotive sessions. The cab handheld will broadcast the speed / direction LCS message, which is received by the base station and translated to a DCC command packet sent out via the LCS bus. From an overall perspective, there is no difference in managing a locomotive session. There is still a handheld to set the speed and direction and there is a central place that is aware of all active sessions.

However, an analog engine has no concept being directly addressable. The typical solution is to divide the track into sections and control the sections where the locomotive currently is. The section is called a block and a block itself consist of one or more sub sections. The track subsections each have a sensor to detect that there is something drawing current from the track and the block controller has a way to know which locomotive is in which block. The following figure depicts an analog layout using the LCS components.



The LCS base station that manages all active locomotives, will not work differently for a digital or analog locomotive. It will create a session and also emit DCC data packets for controlling among other things speed and direction. The DCC signal is broadcasted via the LCS bus. This way it will also reach all block controllers that manage a block. The block

controller will then decode the DCC packet and if it concerns a locomotive that according to the block controller data is currently in the block will put the respective PWM signal on the track. This is different to a normal DCC booster. A DCC booster just amplifies the incoming DCC signal and puts it onto the track. A block controller will decode the DCC signal and put a corresponding PWM signal on the track.

12.3 Locomotive session management

For each active locomotive the base station first establishes a locomotive session. Across the layout, a locomotive is uniquely identified by its `**cabId**`. Once a session is established for the cabId, the base station accepts LCS commands for setting the speed and direction. This is common to both the DCC digital and analog control of a locomotive as far as the base station is concerned. The only difference is that for an analog engine, only speed and direction can be set. All other capabilities such as sound control and functions for turning on and off a headlight are not available.

12.4 Analog Track Signal Generation

The analog track signal does not contain any information transmitted via the signal. The signal is just a pulse width modulated electrical current. The wider the pulse the faster the engines will go. The direction is determined through the polarity of the track. Just like emitting a DCC signal waveform, the H-Bridge of the power modules can easily also emit a pulse width signal with the right polarity. Short circuit detection and power consumption measurement work independent of the kind of signal emitted.

12.5 Analog Track Blocks and Track subsections

Layouts with analog engines will almost certainly have a number of blocks that can be powered individually. There is a one to one relationship of a power module with a block. A block is further divided into a number of track subsections with occupancy detectors, so the locations where power is drawn within the block can be determined. The chapter on block controller and block signaling will pick up this topic in more detail.

Just like the DCC subsystem, care needs to be taken when a locomotive crosses from one block fed from a power module to the next block fed by another power module. The actual current put on the tracks needs to be in sync, such that there is not awkward jump or worse current flow between the blocks connected via the locomotive wheels when crossing. It needs a way of synchronizing the PWM signals. Classic analog block control system transmitted a separate signal for all block controllers. In our world, the DCC signal emitted to all block controller nodes throughout the LCS layout via the LCS bus is our synchronization method.

12.6 A short Glimpse at Software Implementation

The block controller is the heart of managing a block in an analog layout. It will be responsible for managing the track block with a number of subsections. Using the LCS event system, blocks communicate and broadcast data about the locomotive entering and leaving their block. Using defined node and port attributes, they also communicate about block occupancy. Turnout

control and position feedback as well as track signal control are also be the part of the duties of a block controller.

A part of the block controller firmware will decode LCS messages to determine if there is a command that concerns a locomotive that according to the block controller data is currently managed by that block. Note that there is no way to really know that this is the locomotive until there is some mechanism of identifying a locomotive when entering a block. For example, the sending block where the locomotive is coming from sends an event that this locomotive has left the block. Consequently, the receiving block knows the locomotive ID and broadcasts the event of arrival.

Another part of the block controller firmware needs to manage the power module. Depending on the locomotive characteristics the speed and direction are set. Short circuiting and power consumption are measured just like it is done in a DCC subsystem. In addition the PWM signal phase needs to be the same in all blocks. This is accomplished by a common synchronization signal.

Finally, the firmware will track that a train truly left the block. This information is a combination of the follow-on block indicating that the train entered and a computed time interval where the train should have completely left the previous block has passed. If this is not the case, perhaps the train derailed or a part of it decoupled.

12.7 Summary

Analog systems have their purpose also in a digital world. The approach taken by the Layout Control Systems is to put the smarts of managing the running equipment of such a layout into a set of block controllers with the base station and cab handhelds transparently supporting DCC equipped and analog engines. Both worlds use the power module for managing the track current delivery and consumption measurement. While for DCC the power module generates an amplified copy of the DCC signal, it will generate a PWM signal for an analog engine.

The block controller takes on part of the duties of a DCC locomotive decoder in a digital layout. The DCC signal broadcasted to all nodes in the layout is simply decoded and matched with the locomotive information of the respective block controller. All block controllers constantly broadcast via the LCS event mechanism their current state.

Not discussed yet, there needs to be a central configuration system that keeps all the data about all blocks and their relation to each other. There also needs to be a dictionary of all locomotives and their characteristics. On top configuration software and also panels to set track routines and so on. The requirements will be discussed in the block signaling chapter.

13 LCS Hardware Module Design

So far we covered the general concepts, messages, protocols as well as the LCS core library and a glimpse how all of this might be used. Let's take a break from all that concepts and mostly software talk. For the software to run, hardware modules need to be built. Welcome to the next big part of this book. Here, we will talk about the LCS hardware modules. A hardware module conceptually consist of three key parts.

- communication
- controller
- function block(s)

At the center of a hardware module is the **controller**. There is a great variety of controllers and development environments available. When selecting a controller for LCS, we will talk in a minute which one was picked, its is important that there is enough CPU power and equally important a powerful development environment. A console command line interface and interfaces to load the software is also very handy for configuring, monitoring and debugging. The **communication** part implements at a minimum the LCS message bus interface for the messages to transmit between the modules. Finally, the **function blocks** implement the hardware module specific capabilities.

This chapter is the first in series of chapters on hardware modules. Instead of presenting complete schematics for each major hardware module, such as the base station, we will go a slightly different route. We will first present the basic components an LCS node might need. Definitely we will need a controller and a CAN bus interface. Some LCS nodes might make use of an extended non-volatile storage, others need plenty of digital outputs. Just like Lego Blocks, all these parts should be combined easily to form the desired LCS hardware module. We will tackle each component one at a time to understand how they work. The later chapters will just combine these basic blocks with minor adaptations and perhaps some very dedicated components for their functionality.

13.1 Selecting the controller

The module designs described in this book initially used the AtMega controller platform along with the Arduino IDE to write the software. There is the Arduino IDE and by now a whole set of different processors. Since it was released, the Atmega controller family and boards such as Arduino UNO, Arduino NANO, Arduino MEGA are in widespread use. The LCS core library program and non-volatile storage requirements do place however a higher demand on the controller capabilities.

Meanwhile, the Raspberry PI Pico (PICO) controller joined the club. And it has a lot to offer. The PICO is a dual core controller running at up to 133 Mhz. It features a whopping 16Mbytes of flash and 264 Kbytes of main memory. There are plenty of IO ports, and functional blocks for UARTS, SPI and I2C interfaces. What makes this controller especially interesting are the PIO state machines that allow for implementing your own I/O protocols. There is CAN bus software library built using these state machines. This way no extra CAN bus controller is needed. The PICO comes with its own software development kit and also an Arduino IDE integration is available.

As time goes by, there will be for sure other capable controller entering the market. However, when you want to complete a project versus chasing the latest controllers, you will need to pick. In our case, the PICO is the controller of choice. Its capabilities match our requirements and will be a good choice for the years to come. nevertheless, the LCS library software should be designed as independent of a particular controller as possible. More on this later.

13.2 The Controller Platform

The following table gives some guidance on the capabilities needed in our designs. This list also applies in general to other controllers.

Table 13.1: Controller Attributes

Attributes	Notes
Processor	For a typical module, the PICO offers plenty in terms of CPU power. Since we use a software implementation for the CAN bus, running the software in one core and the CAN bus state machine in the other will well match what the PICO offers.
Memory	Memory depends on the size requirements of the node, port and event maps and the node-specific firmware data demands. A simple module would perhaps get by with 2Kb, a base station could easily use 32Kb or even more.
Program Memory	The LCS library already uses round about 64Kb of code storage. A simple module would get by with 32Kb, a base station could easily use 128Kb and more.
External NVM	Additional NVM storage is allocated in a separate EEPROM or FRAM. The capacity is highly dependent on the module use case. External NVM components typically also require the SPI or I2C interface. Most external EEPROM chips have write cycles of more than a million. At a minimum, a chip size of 32Kb is recommended. The PICO does not offer an internal EEPROM, so an external NVM is always required.
Digital channels	The bulk of control lines is digital and used heavily. For some hardware modules, a subset of the digital pins should also be PWM capable.
Analog channels	Analog input is typically used for the power module for analog voltage measurements. Otherwise, it is perhaps optional. The PICO allows for only three inputs. If more are desired, an external multiplexer needs to be implemented.
I2C	The I2C interface comes in very handy to connect a large variety of chips. Communication to the external NVM and also to chips that implement functions such as a servo controller will require this bus.

Continued on next page

Attributes	Notes
Serial I/O	The serial I/O is used in some hardware modules for implementation of RailCom detectors. The PICO features two hardware UARTS and the option to implement more in software using the PIO state machines.
Console I/O	Serial I/O is used for console I/O. Rather than using dedicated I/O pins and a UART block in the controller, the PICO serial I/O will be implemented via the USB connector.
LEDs, Button and Dip Switches	A hardware module could make use of LEDs to indicate readiness and activity, as well as a set of switches to configure a hardware option. Not really required but certainly useful.
WLAN	WLAN is optional. But there is a PICO version with WLAN capability integrated.

13.3 Hardware Module Schematics

Hardware modules are described to large extent via schematics. The schematics shown in the following chapters are all drawn with the EasyEDA software. It is a great hardware development platform, and you can order PCBs for the final design in one easy step. Following a building block principle, the schematic diagrams will show functional components with many network endpoints where they connect to other building blocks. Each network endpoint is labelled with a name that is unique across all building blocks used in a hardware module schematic drawn. For example, "VCC-3V3" will always refer to the 3.3V power supply line. If two building blocks have an endpoint with the same name, the endpoints will be connected on all building block schematics in the final hardware module design.

A general word to the building blocks. They serve as examples of how the individual parts could be implemented and help to understand how each part works. Parts of the library software assume the presence of these blocks and how they basically work. Although the library has been written with as much as possible independence of the hardware, the final adaption of timers, serial lines, I/O pins and so on is required needs to be considered. Throughout the next chapters, you will find comments on what is perhaps generic and what would require some adaption if moving to another processor family.

13.4 Controller and Extension Board

Each node in the layout control system is a node and hence there is a controller for running the node firmware. Without a question, there will be many different nodes and as time goes by perhaps even a new controller families. However, each node would need at least some form of power supply, the CAN bus interface and depending on the storage demands and controller family, an external NVM. On top there is the node specific hardware. One approach is to design a board for each dedicated purpose. This board would include all the common portion for a LCS node and the hardware module specific portion. Another approach is to design a node controller board with extension boards that can be connected to it. In the remainder of this chapter, we will describe the main controller and extension concept. However, it is also perfectly all-right to design a hardware component with all the components integrated on one board. For a complex node such as the base station, this is a very reasonable solution. The building blocks

shown in this chapter thus also form the basis for a more monolithic hardware module design. But first, let's look at the physical dimension of our boards.

picture

All boards will have a form factor of 10cm wide and 8, 12, and 16cm long. In particular, the 10x16cm board should be very familiar. It has the "Euro PCB" dimensions. The main controller board has on the left side the connectors for the LCS bus and the power input. On the right side, there are two connectors toward an extension board. As described before, there are two types of extension boards. The usage of the individual connector pins are described in the upcoming chapter. To ease the hardware schematic development and ensure that all boards fit together, the PCB boards along with their connectors are available as symbols and PCB footprints in the EasyEDA library.

13.5 LCS Bus connector

Every hardware module needs the LCS bus interface to connect to the bus. Some modules may also draw power from this bus. The modules use an RJ45 connector for connecting to the bus. The bus signals can be grouped in several categories. The CAN bus differential lines represent the CAN bus. The VS line is intended for hardware modules with very little power consumptions so that they can directly be powered by the bus. The DCC signal lines are an exact copy of the DCC signal that would go to a track sent out by the DCC signal generating base station. The signal is intended to be routed from the base station to booster nodes, but also to hardware modules that analyze the DCC signal for some action. Finally there is the STOP signal line. This is a wired OR line that allows a simple button along the layout with access to this line to issue a STOP signal. The base station or any nodes interested in the signal can monitor this line. There are the following signal lines.

Table 13.2: Bus Connector Pins

Pin	Name	Purpose
1	DCC-Sig-1	The DCC signal labelled "+"
2	DCC-Sig-2	The DCC signal labelled "-"
3	GND	Common ground
4	RSV	reserved for future extensions.
5	RSV	reserved for future extensions.
6	PWR	The bus supplied 12V power line. This line is intended for devices with very little power consumption to get their power from. Any other module should connect to its own power supply line.
7	CAN-L	Line L of the differential CAN bus signal.
8	CAN-H	Line H of the differential CAN bus signal.

13.6 LCSNodes Extension Board Connector

For interchangeability of extensions, there is a standardized **extension board connector** between controller and extensions. Extension boards come in two flavors. The first will have

the extension connector on both sides of the board. Main controller boards and extension board will have a female connector on the right hand side. The first flavor extension board will have a matching connector on the left side. This way main controller and extension boards can be placed next to each other, just like a train. The second type of extension boards only have the connectors on the right hand side. They are intended for a backplane style layout where main controller and extension boards are plugged next to each other. The overall concept is very similar to the the shield concept found in the Arduino or Raspberry PI universe, except that we can stack boards, as well as placing them next to each other.

The I2C interface will be the main communication method between the boards. In fact all current extension boards shown in later chapters use the I2C communication channel. Nevertheless, a rather rich set of outputs from the controller should be available to the extension board for flexibility. There should be ports for digital input and output, analog input, PWM outputs, serial outputs and so on. The raspberry pi pico offers a great flexibility on assigning function blocks such as an SPI or I2C interface to pins. The extension connector outlined below offers a set of pins which are mapped to the PICO capabilities. The following table shows the connector pin assignments for the communication between a main controller board and extension boards. All boards will have a 40-pin connector organized as 2 rows of 20 pins.

Table 13.3: Extension Connector

Pin	Name	Pin	Name	Purpose
1	DCC-1	2	DCC-2	The DCC "+" and "-" signal as generated by the DCC Signal Generator. These pins are typically driven by the base station generating the layout DCC signal.
3	GND	4	GND	Common ground pins.
5	ADC-0	6	ADC-1	Analog input pins. The input is not protected. The analog voltage range is 0 to VCC.
7	GND	8	GND	Common ground pins.
9	DIO-0	10	DIO-1	Plain digital Pins, input or output. The pins are protected.
11	DIO-2	12	DIO-3	Plain digital Pins, input or output. The pins are protected.
13	DIO-4	14	DIO-5	Plain digital Pins, input or output. The pins are protected.
15	DIO-6	16	DIO-7	Plain digital Pins, input or output. The pins are protected.
17	DIO-8	18	DIO-9	Plain digital Pins, input or output. The pins are protected.
19	DIO-10	20	DIO-11	Plain digital Pins, input or output. The pins are protected.
21	GND	22	GND	Common ground pins.

Continued on next page

Pin	Name	Pin	Name	Purpose
23	BI-0	24	BI-1	Bus Address input lines. Up to four extension boards can be connected, the BI pins are used to determine the I2C address on the I2C extension bus.
25	BO-0	26	BO-1	Bus Address output lines. The BO lines are computed from the BI lines. If for example BI is 1:0 the BO lines will become 1:1. The starting output pins values are 1:1.
27	SCL	28	SDA	I2C extension bus channel. The lines are protected with a serial resistor and there is a pull-up resistor to VCC.
29	RST	30	EXT	RST is reset line. Active Low. EXT is the external interrupt line which be raised from an extension board. Active low.
31	VCC	32	VCC	VCC 5V supply to extension boards.
33	GND	34	GND	Common ground pins.
35	VS	36	VS	Board Input voltage forward. These connector pins are primarily used by extension boards that need the high power input. Examples are H-Bridges on such a board or boards that have their power supply circuitry.
37	VS	38	VS	Board Input voltage forward.
39	GND	40	GND	Common ground pins.

The extension board connectors on the main controller boards are female connectors placed on the right hand side of the board. Male connectors are used on an extension board to connect into the main controller or a previous extension board. There are EasyEDA symbols and PCB footprints that offer the connector pins without you going through these details. The appendix contains EasyEDA symbols for the most common board dimensions with the connectors placed in the correct location. A new projects can just start with these EasyEDA symbols.

A key question is how many controller pins are available to an extension board. As said, most of the extension boards would just need the I2C bus to drive the I2C capable ICs on an extension board. However, since there might be rather complex extension boards, the IO pins needed from the controller board to the extension are many and should allow not only for digital IO but also the function blocks inside the controller. The DI0-x pins on the connector map to the GPIO pins of the Raspberry Pi PICO in a way that most of the controller capabilities can be used on an extension board. We will discuss this in more detail in the main controller chapter.

For even more complex extension boards, it is perhaps the better idea to combine a main board with an extension board capabilities to one monolithic board but still keep the extension connector for other not so complex extension boards to attach. As a guideline, only the first extension board will benefit from all signals coming from the main controller board. All follow

on extension boards will only get the DCC signals, the interrupt and reset line, the I2C signal and the power lines.

13.7 Track Power Connectors

In addition to the extension board connector, there is the **track power connector**. This connector is only used by the base station, block controller and associated extensions. Its purpose is to pass the track power signals from the H-bridges on the base station or block controller (or booster) board to the extension boards. This connector is described in more detail in the base station and block controller chapter.

Table 13.4: Controller Attributes

Pin	Name	Pin	Name	Purpose
1	DCC-SIG-B0	2	DCC-SIG-B0	Bridge-0 DCC Signal "+" and "-".
3	DCC-SIG-B1	4	DCC-SIG-B1	Bridge-1 DCC Signal "+" and "-".
5	DCC-SIG-B2	6	DCC-SIG-B2	Bridge-2 DCC Signal "+" and "-".
7	DCC-SIG-B3	8	DCC-SIG-B2	Bridge-3 DCC Signal "+" and "-".

When using all four bridge signal output pairs, each output pair is rated up to 3Amps. For high power bridges with up to 6Amps, two pairs can be combined and the number of bridges signals passed on is two.

13.8 Summary

This chapter introduced the basic ideas behind a hardware module, its connectors and board layout. A key concept is the idea of a common component, the main controller, and extensions that can be connected. Nevertheless, there are good cases for combining a main controller and the extension hardware into one monolithic board. But in any case, the connectors and their purposes stay the same from board to board. While the main controller boards always have the LCS bus and power input on the left side, the extension connector and track line connector on the right, extension boards come in two flavors.

The first extension board type has male connectors for track line and extension lines on the left side of the board while the second type has not. Both types have female track line and extension line connectors on their right. The first type can just be plugged into the main controller type boards, additional extension boards are simply plugged into the previous extension board. The second extension type is intended for a backplane type design where main controller boards as well as up to four extension board types are plugged into a backplane board. Throughout the chapter to come, you will see how easy boards can be combined using the two connectors and standards behind them.

Ready for the first hardware work ? All aboard, the train leaves for the next chapter.

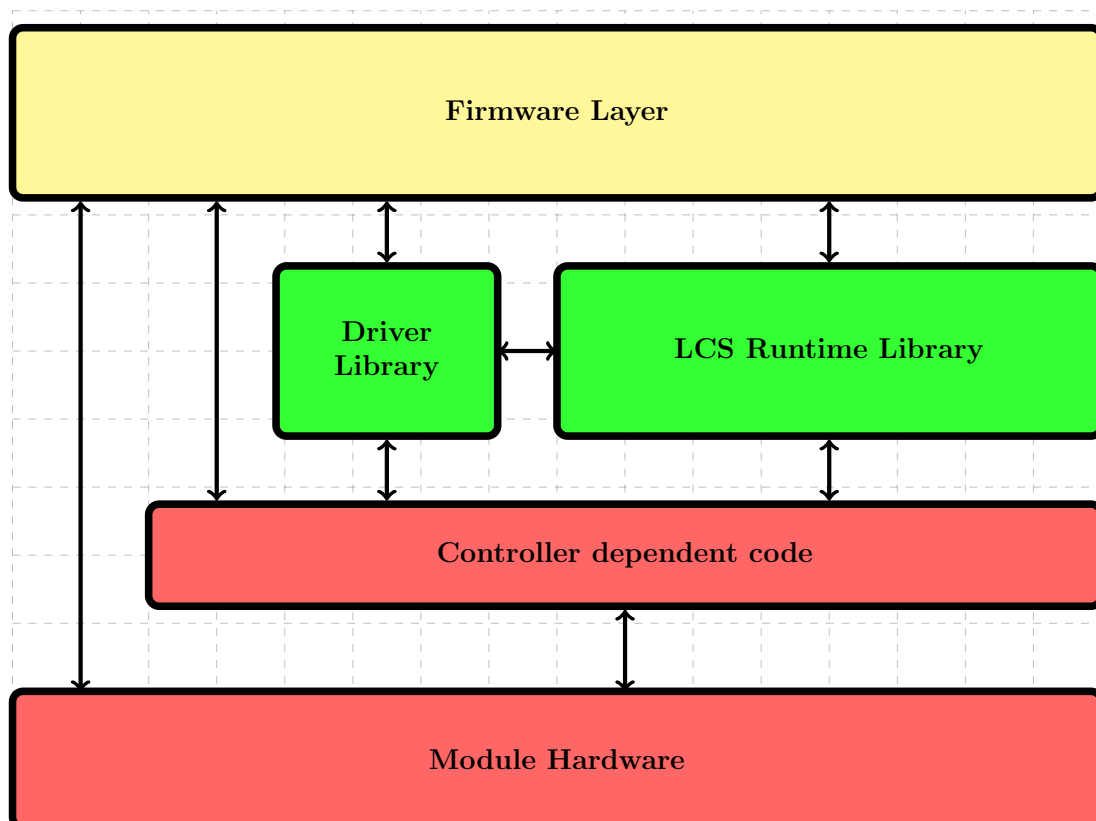
14 Controller Dependent Code

??? rework the text ... extend it...

Enough software talk ? Well not quite. The previous chapter presented the main controller boards and the extension concept. A main controller board features the controller itself, the CAN bus interface, the non-volatile memory and the extension connector with several IO pins of the controller assigned to it. During board initialization the controller hardware needs to be mapped to the actual board. Two processor versions of the main controller board were presented. It can be easily seen that different controller and board versions would place a great configuration and perhaps conditional compile burden on the LCS core library. To address this problem, there is a hardware layer library at the very bottom of the architecture that isolates the controller dependencies.

14.1 The big picture

The figure depicted below shows the refined overall structure of the node firmware. There is the core library discussed in a previous chapters and a new layer, which is the **controller dependent code layer**. The layer essentially encapsulates the required controller functions, such as a timer or a digital I/O pin, and offers a common interface to core library and firmware layer that directly uses the controller functions. The picture also shows a component labelled "extension driver". An extension driver is piece of software that knows how to handle an extension board. We get this this part later when we talk about more about extensions.



Using the CDC layer does not mean that access to the bare bones controller chip is not possible. Any controller HW function can be accessed directly at the expense of that the code will most likely not be portable between different controller families. Still, here may be good reasons for the direct path. The following sections describe will now what the CDC library is offering across all controller platforms.

14.2 Configuring the pins

The CDC library is rather low level hardware abstraction to give access to the pins which will work for the Atmega family and the Raspberry PI Pico and perhaps over time other controllers as well. The key identifier to access a hardware input/output is the pin number. Using the correct pin numbers according to the hardware developed is therefore very important. At the same time the upper layer code should not deal with these details. There needs to be a mapping of actual pin numbers and functions to the controller family and the hardware developed.

Each project therefore needs to define a structure containing some constants and pin number assignments that are used through the node firmware. For this purpose, the CDC library defines a structure with all the pin names and a sensible default setting for a particular controller. Think of the structure a the superset of all pins and HW functions that can be configured for any controller. Using the structure, all the firmware programmer has to do is to set the controller pin numbers of the particular hardware module design to the names predefined in that structure. The upper layer code then just uses these names. The following figure shows the CDC configuration structure.

```
1
2  ConfigInfo ...
```

While the whole idea of the CDC is isolating the firmware programmer from the controller layer and board revisions, the pin assignments cannot be chosen arbitrarily on all supported processor families. An application uses the default configuration routine to obtain an initialized configuration structure with controller specific values set. The application will then fill in the assigned pins and other configuration values. Note that the default configuration setting only fills in what are the controller limitations and leaves all else undefined. An application takes this initial structure and fills in what are board version specific. Throughout the life of the application, this structure is the "single source of truth" for pins. Under their defined name all upper layers refer to the configured IO capabilities. Once all is set, the **init** configuration routine, described later, does the sanity check for the given controller family hardware. For example, on the Atmega the UART pins numbers are fixed if used. On the PICO the pin numbers can be assigned to a couple of different slots. Whatever can be checked is checked. There is however no absolute guarantee that the configuration structure is valid in any cases.

Most CDC routines use pins as one of their input argument. These arguments are not checked again at the level of the configuration validation. For performance reasons, just some quick sanity checks are performed.

For each part of the CDC library, there is a configuration routine. For example, the digital IO configuration will set a particular pin to be an input or output pin and so on. During this configuration only basic checking what a controller can support on that pin will be done. The ATmega is far more restrictive with respect to the IO pins used than the Raspberry. The CDC library will do whatever it can to do such checking. During actual usage of such a pin, i.e. the digital read or write in the example above, no further checking will take place. During initialization, the configuration structure is checked against what the controller is capable. This also includes the pins assigned to UART, SPI and I2C pins.

14.3 CDC Library setup

To the firmware programmer, the library is a set of functions in the name space CDC. A call to any of the routines typically has the form "CDC::xxx". Of course, the name space can be declared upfront so the prefix is not needed. The example shown below will just show the fully qualified signature. The very first thing a node firmware should do is to set up the controller dependent library. If for some reason access to the lower layer is required before the LCS library is initialized, the calls can be made directly from the node firmware. There is also a convenience routine to print the content of the configuration structure.

```
1
2 Declaration part...
```

14.4 General Controller Attributes and Functions

The CDC layer provides a set of common low level functions. There is a function that creates a unique ID for the controller board. It is primarily used when a node needs a hardware module unique identifier. The controller internal memory and any internal EEPROM sizes return the hardware capabilities of processor and installed NVM. The NVM select pin is used for the VNVM memory SPI addressing. Finally, the CANBus controller needs to know the SPI bus select pin and the mode, i.e. baud rate and controller frequency. The library also offers the timestamp routines for milliseconds and microseconds since start.

```
1
2 Declaration part...
```

Some of the routines can also be found in the Arduino IDE and its libraries for the Arduino. As this project perhaps may also be implemented in a non-Arduino environment, this dependency is also hidden behind the CDC layer.

14.5 Power Fail detect

The main controller board features optional power failure detection. The power supply provides a signal line to the controller which goes low when power drops. The power fail input pin is set in the configuration structure and just passed as an input to the configure routine.

```
// ??? has changed...
```

14.6 External Interrupt

The CDC library offers a set of routines for handling an external interrupt. While a controller typically allows for interrupts on almost any IO pin, there are one some controllers dedicated IO pins which offer an interrupt input with flexible setting and high resolution timing. A callback needs to be registered for this interrupt.

```
// ??? has changed ...
```

14.7 Status LEDs

The main controller board features two LEDs. They are the ready and the activity LED. They are accessed via the **writeDio** routine. The LCS core library will use the ready LED to indicate that the node is ready. The activity LED is used to show library activities such as receiving a LCS message. The recommended colors for the LEDs are a green for the READY LED and yellow for the ACTIVE LED. The following just shows an example how to control the LEDs.

```
// ??? this is done via nodeControl calls ...
```

14.8 Timer

Although the LCS core library itself does not make use of a timer, having a general timer with a callback routine interface is an essential component. The DCC signal generation for example makes intensive use of timers to generate that signal. The timer is a repeating timer and accepts a timer value measured in microseconds. In addition, the timer already starts again counting in parallel to the timer interrupt handler code. Finally, the timer limit, i.e. the timer when the next interrupt would occur, can be set without disturbing the already active count.

```
1
2 Declaration part...
```

Timers work slightly different on Atmega and PICO. Nevertheless, both versions allow to set the new limit, i.e. the timer value when the next interrupt would occur, while already counting toward the limit. Care needs to be taken however to set the new limit while the counter is below this limit. If the new limit value is below the timer counter value, we have passed the limit point already and the counter would simply continue to count and wrap around before hitting the new limit. Any carefully designed timer signal is gone. Again, better be quick in the interrupt handler.

14.9 Digital IO

The digital IO routines offer an interface to plain digital input and output operations. If it is an input channel, the input can be set to active low or high input. Also, there is an option to enable the controller internal input pull-up resistor. For configured output pins, two pins can be set in pairs if supported by the actual hardware configuration, i.e. the two IO pins are on the same controller output port. This feature enables the simultaneous setting the two pins. A typical use case is the DCC signal where the two signal levels are set in one call.

```
// ??? add the interrupt stuff...
```

```
1
2 Declaration part...
```

14.10 Analog Input

Analog input configures the respective controller input ports for reading an analog value. The read method offers an asynchronous way in just starting the analog input conversion process and a hardware interrupt when the conversion completes. The registered call back is then passed the value of the conversion process. The **adcRead** function is a blocking ADC measurement call.

```

1
2 Declaration part...

```

The analog to digital converter system for the Raspberry PI PICO is compared to the Atmega really fast. A typical 12-bit conversion takes about 2 microseconds. The PICO ADC unit resolution will be scaled down to 1024 to match the Atmega resolution.

14.11 PWM Output

Depending on the controller, some digital pins can be configured with a time period and pulse width ratio. The capabilities of the underlying processor also determine what kind of PWM is possible. For example, in the main controller board Atmega1284 processor version, Timer 2 is used, allowing for two separate channels. Both channels can be set to either a fast PWM where the timer just counts up, or a phase correct mode, where the timer counts up and then down, essentially dividing the PWM period by two. Since the PWM channels are configured as two independent channels, the PWM period can only be set according to the processor clock frequency divided by the pre-scaler fixed values.

```

1
2 Declaration part...

```

The pre-scale options for the Atmega are limited to what the particular timer allows. In other words, the frequency can only be set to the nearest value of what the pre-scale option will allow. The PICO is again far more flexible allowing for a true frequency setting.

14.12 UART Interface

The UART interface is used to offer a serial communication channel. This is required for the RailCom feature. Currently this interface implements only an asynchronous read into a local data buffer. However, the configuration routine allows to set more parameters that are needed for the current usage. One day, even a write capability may be needed.

```

1
2 Declaration part...

```

The Raspberry PI Pico offers to implement the UART channel on one of the PIO state machines. This allows for more than the two UART blocks of the controller. The **UartMode** parameter selects what kind of UART HW is actually used.

14.13 I2C

Controller offers a serial wire IO block. The I2C will need two pins for clock and data. There is one I2C port on the Atmega1284, and two hardware blocks on the Raspberry Pi Pico. The CDC layer offers an interface to read and write a byte.

```

1
2 Declaration part...

```

14.14 SPI

SPI is the bus used for connecting NVM and CAN controller chips. The CDC library will validate that the configured pins are actually available for the SPI IO block in the respective controller. By nature, the SPI communication exchanges a data item with between two entities. A master sends a byte and in return receives a byte from the slave. To avoid surprises such as filling a buffer sent with whatever is returned from the slave, the transfer routines available will NOT overwrite a buffer sent with whatever data returned. A future version may change this behavior and offer dedicated routines to do a write or read with the transfer semantics.

```
1
2 Declaration part...
```

14.15 Extension Connector and hardware pins

The routines described have been implemented fairly flexible and their main purpose is to shield the upper library and firmware from the controller specific implementation methods. The same routines are also used to control the pins of the extension connector. If you recall, there are two ADC channels, a digital channels and an I2C communication channel available. The first two digital pins can be overlaid with an UART interface, and the last two digital pins allow a PWM capability.

The configuration descriptor structure uses predefined names for the pins of the controller. If the hardware exports these pins via the extension connector, the connector pins can be accessed by these names. However, not all pins need to be provided to the extension connector. If for example, the digital pins DIO 0 .. 4 are used local to the board, the pins are left open on the extension connector. The only mandatory pins available on any extension connector are Power, ground, I2C, reset and E-Stop.

14.16 Summary

The controller dependent code library is the lowest layer in the LCS node software stack. Its purpose is to shield the firmware programmer from the underlying pin assignments and some of the intricacies of the particular controller. At the same time, the layer needs to be rather thin so that it adds very little to the overall path length for performance critical signal management. For special cases, there is always the possibility to access the underlying hardware directly. However, this coding my not be that portable then.

15 LCS Node Firmware Design

The previous chapters introduced the overall Layout Control System architecture, the communication concepts and how an LCS node could be implemented. The hardware design chapter also presented the main controller and extension concept for splitting up the work between the generic controller and a specific extension. We are about to embark on designing specific LCS nodes such as a base station or a block controller. Before we do that, this chapter will outline how in general one does write firmware for the layout control system. This chapter will not focus on the how to use the library. This was explained before in small examples, and the major modules firmware code to follow in the next chapters contains good larger examples for firmware designs. This chapter will rather focus on how you go about designing a firmware for a node and give guidelines and recommendations.

15.1 General Thoughts - Nodes, Ports and Events

It is the general philosophy of any software system to find a good balance between what is coded or written in firmware and what is done with setting configuration values. Without a concept how to enter configuration values, even a simple change would result in downloading the firmware with the changed data. Clearly, we need a better way. The idea should be to update the firmware only when there are new features, fixes to bugs, and new installations of hardware capabilities of the node. This first of all means that each hardware capability is accessible through user defined port and node control and info items. That was a key reason why there is a range of user definable items for these attributes. A configuration system can query and set defined attributes and also execute functions mapped behind a node or port item.

Just using the ability to send messages to a node and port allows already for a very capable control system. Similar to the DCC CV variables, nodes and ports have a set of variables that can be queried and modified from any node. The addition of callbacks in the path of accessing such a variable, allows to implement for example querying a hardware resource and return the result via the variable associated with it. In addition, the firmware designer can define custom items that will invoke a callback function. This function now can do all kind of things. It is completely up to the firmware designer what this function will do on that node. Just using the node and port item concept, a control system could be implemented with a central station containing buttons, switches and status LEDs that send these messages to node which then set the turnout direction acknowledgement, which in turn set the LED. This way we know the turnout direction setting. A central concept, but with few wires. Only the bus lines go across the layout. But that alone would just mean a more economic wiring approach. And the central bus load would perhaps be considerable for larger layouts. A CAN bus frame is between 58 and 114 bits. With a baud rate of 500Kbits and an assumed average packet length of 6 bytes, roughly 5000 frames per second are the limit.

The addition of an eventing system allows for all nodes to react on events broadcasted by any other node. A setting of a turnout, for example, could result in broadcasting an event for all other nodes to see. The central station mentioned before could use this event to set an LED light on the control panel to show the turnout setting. Instead of periodically polling the turnout setting, we now just process the respective event. Furthermore, since all nodes can see all events, many nodes can react to an event sent. All that needs to be done is to configure the

port to react on the given event though invoking a callback. Events provide a great flexibility on top of the query and set scheme outlined before.

Let's take the common example of setting a route. This would involve a node where there is some button or other way to broadcast the "set route" event. Each node that has a turnout belonging to the route was configured to act on this event and the port representing the turnout would then take action via the callback. A simple "event delay" capability allows to set one turnout after the other and not all at once. The nice thing about the event driven approach is that the ports need not be on the same node. And, more importantly, the callback functions are just controlling a piece of hardware and the need for updating the node firmware would only be needed for entire new capabilities, firmware update or bug correction. In short, there is a fine line with what the firmware should do and what is delegated to configuring ports and events.

It would be tempting to also offer a kind of macro capability to execute a macro instead of actual code. Upon receiving an event, the macro would be executed. And going this route we are just about to invent yet another language. Right now, the jury is out whether this is really needed. The whole philosophy of the Layout Control system is that events are produced and visible for all nodes and each port on a node can act on this event. Node local processing just concerns the sensors and actors belonging to that node. If on a node more than one port is registered for an event the ports are triggered in ascending order. It really depends what the callback functions associated with the port are executing.

15.2 General thoughts - Software layers

The layout control system will over time contain many hardware and software components. The software therefore needs to be structured in a way that the firmware designer just will focus on the the tasks at hand and all else is being taken care of by the core library. Well, almost. The great variety of hardware requires a slightly more detailed picture.

Besides whatever the node is designed for, there are the common firmware layers. It all starts with the **controller dependent code**, that shields the actual controller family from the rest of the library. This does not mean that any special hardware feature of the controller is not available it just means that the common capabilities to be found across the implementations are available via a common layer. Examples are the routines to manage a digital I/O pin, the analog input, hardware timers and so on. The **non-volatile memory** CDC library part implements a simple memory that is used by the node firmware but also the firmware designer. Some controllers do have a non volatile memory area, often an EEPROM, others don't. The library offers a simple abstraction for storing non volatile data. The **CAN Bus** CDC library part is responsible for implementing the CAN bus interface, which is either implemented as a separate hardware chip or implemented in software using the controller special capabilities, such as the Raspberry Pi Pico programmable IO blocks.

At the heart of all node specific firmware is the **LCS node library**. It itself uses the aforementioned CDC libraries where needed. The LCS library actually implements the event loop where events and message are handled and manages the node data. Any firmware written for a node sits on top of this library, but still has access to the underlaying hardware. Extension boards need their own piece of software to manage whatever the extension board implements. We will call this a **extension driver**. A driver is access using a defined set of routines available through the node library. Firmware programmers think of such a board as a peripheral device that is access though a set of control, info, read and write functions. The core library contains a set of functions to manage the driver subsystem. Examples are extension board discovery and address mapping.

Finally, there is the **node specific firmware**. This part is written by the firmware designer, i.e. you. It communicates with the lower layers through the set of defined APIs and the set of defined callbacks. In fact, a great deal of firmware specific code development is just writing the callback handlers for the core library and calls to the respective extension driver API. The main routine of a node just consists of call to initialize the library, registers the callbacks and let the library loop do its work. Throughout the chapters that implement specific nodes, you will see this basic structure.

15.3 Node Functions and Attributes

A key idea of designing the firmware is that there is no need to update the firmware every time configuration or other parameters change. It is in a sense quite similar to a DCC decoder which offers a large set of variables that can be set with values that the decoder firmware interprets. LCS nodes have a concept of attributes that is very similar to these variables. A node variable can be queried and set from any other node.

Node variables are accessed via "items". An item is just a number that refers to the variable. Besides a set of fixed, i.e. reserved, items that the system itself offers, a set of items allow to access a variable in memory or NVM. Any values that you want to keep across a power cycle can be stored in these variables. The idea is that during startup the values are simply copied from NVM to memory for usage. In the opposite way, there is a set of items that when used, will store a new value in memory and NVM. So, while the memory value can change quite a bit during operation, startup always used a known fixed value.

A node has up to 64 such variables. That is not a lot, you might say. Well, not quite. The item range has a set of 64 user definable items. These are items that when accessed will result in a callback to the node firmware. And in such a callback function, you can do anything you like, including accessing these local attributes of course. Beyond this way of using a user definable callback, these callbacks offers a very flexible to directly access firmware functions without going through kludges of writing a value to a variable that is interpreted as a functions. Since accessing local attributes is such a common scenario, you will find in the reserved item section also items that allows to directly read and write to local attributes.

Finally, think about where these variable values would come from. There will be the day where a hardware piece breaks down and needs to be restored. It is rather easy to load the node variables and attributes from a central database, as well as to first of all store these values in such a place.

15.4 Port Functions and Attributes

Ports are the higher level endpoints on a node. They too features variables accessible via the items already presented. There are just no local attributes related to a port. But what is a port actually used for? When designing a node firmware, the designer should use ports to group functions that are accessible by a combination of node and port identifiers. As seen in the concepts chapter, they together form a 16-bit value. A good example is a block controller that has four channels to manage a block on the layout. Each block, you guessed it, can be represented as a port. When talking to the block, just the node and port ID is enough to address it.

But ports are also the endpoint for events. Remember each node can broadcast an event observable by all nodes on the system. Events are created to represent an, well, event that somebody would be interested in. For example a track power shortage. Or a block section is

occupied event. Or whatever you see as a reasonable event to broadcast to the world. And again these the event map data can be queried and loaded from another node.

When configuring a node, that node can register its interest in an event by adding an event ID / port ID combo to the event map. How to do this? Well, there is a reserved item function to just do that. A matching incoming event will result in a callback registered for the port. A node offers up to 15 ports. Since they also serve as a logical grouping of physical things on the layout, having 15 ports is a good compromise between number and resources needed by a port. When designing a firmware, the first thinking should be what a node manages and what ports will manage portions controlled by that node.

15.5 Command Line and Display

The LCS library already offers a command line interface. It provides commands for the basic commands to manage a node and access its data. This interface, while not used in regular operation, is very useful for node firmware testing and debugging. All commands are also available via LCS messages, so that troubleshooting and monitoring can be done when the node is already installed. Following this philosophy, implementing node specific commands are a good idea. Writing your own serial commands is just writing a command line callback and registering it at node startup.

15.6 Event handling

The layout control system is an event driven system. A large part of layout configuration consist of creating an event Id, i.e. picking a number, and associating a port with the event. Upon event detection, the port callback is invoked. Configuration is therefore just entering the event/port pair in the node event table. The other direction, a situation at the node results in broadcasting an event, requires that the event ID is known. A good practice is to define port or node attributes that contain the event Id to use. This attribute can be set during node configuration. The LCS core library offers a periodic callback where the firmware designer can implement to manage the local sensor hardware data and decide to broadcast an event.

15.7 Periodic Tasks

Since the core library implement the outer loop, tasks that need to run periodically need to be implemented as callback for the core library to invoke. As always, one can write timer code and management outside of the core library, but this comes at the expense of being dependent on a particular controller hardware. For most cases, a simple callback, it does not even have to be precise to the microsecond, is sufficient. A good example is the checking for power consumption on a track. A firmware designer would provide a callback that is invoked periodically and would make use of the driver interface to obtain the actual data. If the consumption is outside defined bounds, an event will be raised and the track turned off.

15.8 Configuration

Configuration comes in two parts. There are all the items that need to be handled of ramping the hardware actually perform, and there are the items that are at the higher level of configuring a node for its purpose. A great of how to configure the hardware was already presented in the

chapter on the CDC layer. At this layer the hardware setup was place and all capabilities are setup to be used. As a firmware designer, unless you directly access hardware, you may not have to go that deep, the CDC library / Core library will have the configuration data that maps the actual hardware.

What is left is from a low level perspective are the extension boards and their inner workings. These concepts will be explained in great detail in the chapter on the extension board designs. Right now it is sufficient to know that using the driver interface, some setup and configuration needs to be implemented. For example, setting an I/O pin on a GPIO extension board to be an output port, is a typical configuration task. Again, more on this later.

What is left is the high level configuration, which means to set node and port attributes. This task is comparable to a DCC locomotive decoder. It is has tons of variables that can be set. The firmware designer is responsible to make all relevant items of the node firmware configurable by exporting them as node and port attributes. When w look at the first larger node example, the base station, this concept becomes clearer.

15.9 The main code

The main code is just a set of firmware specific code and callback routines that do the firmware specific work. During initialization, register the necessary callbacks and then delegate control to the LCS loop. This scheme is common to all nodes. For smaller nodes, the setup code and callback routines can be all in one file, for example the "main.cpp" file. Larger node firmware will perhaps split the code into several files. The C++ classes or structures are a good way of structuring your code. There is one caveat though. Callbacks are technically just procedure labels and need to be at either the main code, the file local portion of a separate file or a static methods in a C++ class. In other words, they cannot be object instance methods.

15.10 Summary

This chapter gave a brief overview how one would go after writing node firmware. In short it is writing functions that can be registered as callbacks to the core library and that will in turn use the core library functions for implementing their purpose. It also introduced a more detailed picture of the overall software layers. The heavy lifting of message handling, event handling, node data management, extension board management and overall processing is handled by the LCS library. All the firmware programmer has to do is to write the node and specific functions.

// ??? GitHub has all the source code of the nodes i wrote ...

The following chapters will give several examples of how the firmware for the common nodes such as a base station, a cab handheld and a block controller is designed using the principles and guidelines mentioned in this chapter.

16 Power Module Design

LCS hardware modules generally consist of two parts. The first is the main controller portion where the processing is done. The second part typically implements special functions to complete the LCS node. One such function is the power module for the DCC subsystem. This chapter will present the basic components necessary for building the power generation part for a base station or booster and block controller module.

16.1 DCC Track Power Modules

A layout control system needs a form of DCC signal generator. They can be directly integrated with the base station to form one hardware module or be at the heart of a separated booster module. Depending on the model railroad scale, there are different voltages and maximum current considerations. A Gauge N layout has a different power requirement than a Gauge G layout. The NMRA standard defines the voltage ranges for the different gauges. Our assumption is that the layout system has a form of power supply that delivers the voltage and current requirements to the power module. For convenience, all hardware module node local power supplies for the controller and other chips should be able to handle a voltage range between 7V and 24V when drawing power from this power supply. Switching supplies are quite interesting, they do not produce that much heat compared to their analog counterparts. The main task of the DCC track power module is to provide the DCC signal to the track. DCC signals are square wave signals with a defined duty cycle period. A duty cycle of 58 microseconds represents a "DCC one", a duty cycle of 116 microseconds a "DCC zero" bit. A typical solution to this task is a H-Bridge design. The following figure shows the H-Bridge signal convention for LCS power modules.

While a power module can vary in capacity and technology components, our logical design expect to send the same digital signals to a power module, no matter what it's design. It is a key requirements to derive from the DCC signal on the bus connector the state "+", "-", and "short circuit". The latter is needed for the cutout option. There is also the need to detect a logic "one" on both signal inputs and interpret this as setting the bridge to a high impedance state. This requirement comes primarily from the capability to also issue PWM signals, which will depending in the direction alternate between a positive or negative voltage and the high impedance state. There is a whole family of H-Bridge ICs and breakout boards. If a particular H-Bridge IC does not map the inputs to the table below, some gate logic needs to be added. The following table depicts the power module digital input management signals. The table below fits the L6205 H-Bridge nicely.

The standards also specify which side of the track is the positive side, i.e. OUT1 in the table above. The right hand track side, usually connected with a red cable is the positive side, the left track side, connected with a black cable, is the negative side. If the H-Bridge is enabled, sending a "DCC One" will mean to raise the digital output of the controller port DCC-SIG-1 to HIGH and controller port DCC-SIG-2 to LOW for 58 microseconds, followed by the reverse bit setting for another 58 microseconds. The DCC packet is broken down, bit by bit and the digital signal is produced. The H-Bridge hardware then takes the zero and ones and essentially reverses the track polarity accordingly to digital zeroes and ones.

If the power bridge is used for the PWM mode, a "FWD" means to raise the DCC-SIG-1 to HIGH and the controller port DCC-SIG-2 to LOW for the active part of the duty cycle length.

The remainder of the duty cycle, the port signals will be both set to a HIGH, which puts the bridge into the high impedance state. Not all H-Bridge ICs follow the same control signal level standard. Any difference in H-Bridge control signals in/out need to be masked accordingly. The following figure shows the H-Bridge signal standard. All power modules and connectors to the track follow this convention.

All power modules are expected to deliver a voltage proportional to the power consumption of the H-Bridge. This is typically done by putting a shunt resistor between ground and the lower part of the H-bridge. For a better accuracy, the rather low voltage signal is amplified before delivered to the analog input of a controller. Note that most H-Bridge ICs offer short circuit and over temperature protection already as a built-in feature. Based on the analog voltage signal analysis, current protection features such as sending a power overload event can be implemented.

Power consumption measurement is necessary for another important DCC requirement. A DCC decoder in programming mode will raise its power consumption to acknowledge an operation with a raise by about 60mA. This short rise needs to be detected as well. It requires to calibrate the actual power consumption of the decoder, build a base of typical consumption and then detect the temporary raise. With the basics in place now, the next section will show different designs for a power module using the L6205 chip for the implementation. As always, there are other H-Bridges and also breakout boards that could also be used as the heart of a power module building block.

16.2 Dual Power Module - L6205

The representative schematic for a power unit uses the L6205 dual H-Bridge IC for a dual power module unit. There is a serial resistor on bridge ground side to measure the current consumption. The voltage drop over this resistor is amplified and will be passed to an analog controller pin. Both bridges deliver up to 2.8A, which is sufficient for scales up to HO Scale.

When building a base station, one h-bridge is used on the main track and the other for the programming track. The programming track that actually would need a much lower current, why use a design with two equally powered H-bridges? One answer is that you can get such an ICs with two H-Bridges inside. The other answer is that a design with two equal H-Bridges would allow for an interesting feature. Imagine you could feed the main track signal to both the MAIN and the PROG track. A locomotive could drive under its own power onto the PROG track, which is acting as a MAIN track section. Then the DCC signal is switched back from MAIN to PROG and the locomotive configuration can begin. Now, the same could also be accomplished with some relay based logic, but wouldn't this be an elegant approach? More on this idea in the base station chapter.

16.3 Mono Power Module - L6205

The L6205 dual bridge can also be used for a DCC booster hardware module with a higher amperage output. By now, the basic parts of the schematic shown below should be familiar at a high level. The L6205 chip allows to combine the two H-Bridges to deliver up to 5.6Amps. All else is fairly identical to the dual design discussed before.

There is a smaller cousin, the L6225, which delivers two times 1.4 Amps or 2.8 if the two bridges are combined. The electrical control signals and the Pin layout are identical, so it is a good candidate for a smaller mono or dual power module.

16.4 Power Module - Breakout Boards

There are a lot of power module boards readily available. There is a popular Arduino shield version built around the L298 chip that can directly put on top of an Arduino UNO or MEGA. Just to name one. There are also breakout boards that deliver really high power levels, easily up to 10 or more Amps, at a very low cost. There is one very popular bridge out there built upon the BTS7960 half bridge, which is rated up to 30A. There is even a Arduino UNO shield available using the BTS7960 chips. Well, 30A is perhaps a bit too much for a model railroad unless you want to weld engines to the tracks in case of a short circuit. Depending on the breakout board used, some "glue" logic to match our DCC signal standard and a current consumption measurement logic needs to be added. The appendix has a section that describes the PCB layout for an empty board with just the connectors. This board can be used to piggyback a power module breakout board on top.

16.5 Summary

This chapter presented a basic track power module designs. It contains a H-Bridge and a means to return a voltage proportional the current consumption. Depending on the model scale and the layout size, power modules are found in base stations, boosters and block controllers. For interoperability, a power module building block is expected to accept the control signals commands via the digital control lines defined. Any new H-Bridge design needs to make sure that it supports the defined control signals.

17 Railcom Signal Detector

Although a standalone component, the RailCom detector is typically a part of the power section. It sits between the power module and the track. DCC is a broadcasting protocol. To address programming engines on the MAIN track the problem of providing a back channel had to be solved. First, the DCC signal generator needs to be able to include a cutout period in the bitstream. As shown in the DCC subsystem chapter, the cutout period happens right after the last bit of a packet sent as part of the preamble section of the next packet. The power module short circuits the track during this period. The decoder uses the period to send a short bit stream which in turn is detected by the RailCom detector. This section will just discuss the hardware part for receiving the RailCom bitstream in the cutout period. There is of course a software part that receives and decodes the RailCom datagrams. The details of RailCom message processing are described in the base station chapter.

The above schematic shows a dual RailCom detector. As we will see in the block controller chapter, there is the requirement to have two or even four such detectors on one board. This allows for using chips that have four comparators, as shown here. If only one channel is required, a schematic with for example an LM393 would be the solution. The principle of operation is the same. All the examples will rest on the cutout capability of the power module and the detector circuitry shown here.

17.1 Summary

The RailCom channel was an important addition to the DCC standard. With RailCom capable decoders data can be queried and set while the engine is on the main track. This chapter showed a simple building block that allows to detect a RailCom stream. It is actually a very modest hardware addition, so that each H-Bridge delivering DCC signals should be equipped with such a detector. We will make use of this building block for all power modules found in the base station and block controller designs.

18 The Base Station

Take a deep breath. We are about to put together our first major LCS hardware module. The previous chapters introduced the message format and protocols and the core library for implementing the event system as well as the running equipment based on the DCC signal standards. Next, we took a closer look on the major hardware building blocks and power module designs. Just like the LCS core library allows to build a node specific firmware on top, the hardware building blocks are the foundation to build the required hardware modules. We also looked at how one would go after designing the node firmware in general. So here is the first and most important hardware module putting it all together. The base station. Every layout needs to have some kind of a base station that acts as the central place for layout control and signal generation.

Looking at the market, there are plenty of so called base stations. They typically offer support for several standards and communication protocols, such as DCC, mfx, LocoNet, a Can Bus, a S88 sensor bus, and so on. Most base station also have the power module directly integrated. They support the configuration of locomotive and stationary decoders. In short, a one stop all round solution. Their price range is around few hundred Euros. With the advent of Arduino, Raspberry and other controllers there are numerous do it yourself solutions. Just to name one, the DCC++ Arduino base station with a motor shield as a power unit, gets you a base station for well under hundred Euros. The excellent work of the JMRI community to provide a DCC++ interface for configuration software and other utilities to use this inexpensive base station hardware. The DCC-EX group extended and stabilized the original DCC++ work for a wider range of controllers but also with new capabilities. There are many such great projects. The appendix provides some links and pointers to this work.

18.1 Key Requirements

Our base station needs to deliver the following capabilities. At first it needs to be able to assemble the DCC packets and generate the respective DCC hardware signals. This work is split into the base station producing the signal content and the power section driving the hardware. Furthermore, the base station needs to provide a way to manage several locomotive sessions. For each active session the current state of the locomotive is maintained and the DCC packets are produced. When a new locomotive session is established, a dictionary of locomotives could be consulted about the particular locomotive to get the initial function settings, and so on.

A base station implementing locomotive session and track management should also implement a serial command interface for managing a session or sending commands to a locomotive. Although not really necessary, it is very beneficial for testing and debugging. But also, programming DCC decoders will need some form of getting the configuration data to the decoder. There are great openSource tools out there which make use of an ASCII interface to send their commands. An example is DecoderPro from the JMRI teams. It features among other protocols the DCC++ ASCII interface to send and receive commands. Our base station will therefore implement the relevant DCC++ commands.

All configuration settings, such as the number of concurrent sessions or the current consumption limit for a track, should be available as attributes on the node itself and ports. A

track should be represented by a port, so there will be a port for controlling the MAIN track output, and a port for controlling the PROG track output.

Finally, the firmware for the base station could also host LCS management functions, such as a configuration database or display data about layout operations. This is not per se a function of the base station, but as each layout needs to have a base station and perhaps display high level status data, it is a convenient place to put central functionality there as well. This subject will be discussed in another chapter, this chapter will focus on the core base station features.

18.2 Module hardware

The base station is essentially based on a main controller board and a dual power module unit shown in previous chapters. We will use the PICO controller version and the dual H-Bridge building block. Add the RailCom detectors and stir the whole soup for a while. The extension connector of the base station would still export the I2C interface and the DCC signal produced by the base station. So, adding an I2C based extension board for display, switches, etc. is of course still possible. Here is the schematic for the base station. The individual building blocks should be familiar by now. The first page shows the main controller parts. Note that it needs fewer level shifters, as most of the signals are consumed internal to the board.

// schematic.....

-1.png)

The Raspberry PI Pico has enough capacity to implement the CAN bus protocol directly using one of the cores. As a result, only the line driver is necessary to implement the CAN bus interface. There are the level shifters for the I2C bus and a few other external signals. The Power supply needs to have a method to detect that there is an USB cable connected to the PICO and ensure that there is no conflict between the power sources. Like any LCS node, the controller needs a non-volatile memory. The base station features two chip slots with each slot hosting an I2C type NVM with up to 128Kbytes. The key reason for such a high capacity is that a base station might store a lot of data for each engine that it can manage.

The next part shows the power module and RailCom detector unit. The power module exports the DCC signals via the external track power connectors and also as part of the LCS message bus connector. The track power extension connector is used by extension boards that directly use the H-Bridge output. A good example is an occupancy detector board which takes the DCC outputs and routes them to different sections, each equipped with a detectors for power consumption on the track. The power module unit features two identical channels. They are labelled "MAIN" and "PROG". Although the "PROG" channel would not need to deliver a high amperage, the dual H-Bridge is there anyway. And as said before, it would be nice to dynamically treat the PROG track as a type MAIN track too. Both channels therefore also feature a RailCom detector. Refer to the base station firmware chapter on what we actually do with a RailCom detector.

The base station will also offer an optional LocoNet interface. One day. LocoNet is very popular communication network for model railroads and there are a lot of devices such as Cab Handhelds that connect to the LocoNet bus. Wouldn't it be nice to just connect these handhelds and alike via the LocoNet bus such that they can be used as well ? I guess it would. Right now, this is work in progress, but let's already reserve the the space on the board. Here is a first sketch of the interface.

Finally, there are the connectors and the power supplies. While the LCS node runs with 5V as before, the LocoNet interface will offer 12V on its bus to connected devices. This part is

also optional if LocNot is not implemented. In addition to the basic connectors that we have as part of the LCS node design, there are also two power connectors for the two tracks.

Like the main controller, the monolithic base station board makes extensive use of SMD parts. While the previous boards have already been using passive SMD parts such as capacitors and resistors, this board also make use of SMD ICs. The exception is of course the Raspberry PI PICO board and the Dual H-Bridge IC L6205. The H-Bridge is a high power part and in case of a hardware problem it can easily be replaced as a DIP version.

18.3 Base Station PCB

The following picture shows the PCB for the monolithic base station. It is a 12cm by 10cm board, with the standard connectors in the usual place. As said, the LocoNet Interface is optional. Each LCS node needs at least one NVM chip. The second is optional too.

18.4 Summary

The base station, no matter how it is implemented, is a key component in any digital layout. It is primarily responsible for the locomotive session management and track signal generation. There could be designs where the base station is a device in a nice housing with a display, switches and so on. Other designs just put the LCS node somewhere on the layout. All these options will work just fine. And just like any other LCS node, the base station firmware offers through node and port attributes status and control functions.

The base station was also the first major LCS node with a considerable amount of hardware and software and thus went to several iterations and a considerable amount of learning curves. The very first version started with an Arduino Mega and Motor driver breakout board. The software was the original DCC++ software. The next versions actually were just a refactoring effort and work on the software. I learned a great deal on how all this works. The next base station version was a single board with all possible interfaces on an experimental PCB. It already featured CAN BUS, dual H-Bridges, and a controller, the Atmega 1284. This board served a long time for software development and further experiments. The next version of the base station was built with a main controller PCB described in the earlier chapters and a dual power unit, that also featured the RailCom detector. And again further software refactoring, refinement and learning. Finally, after a switch to the Raspberry Pi Pico controller, first on breadboard then on PCBs, the base station shown in this chapter is the current version. The lessons learned proved to be very valuable for the overall concept development and hardware design. And in addition, there was a great deal of reading on chip specifications and, very importantly, the DCC standards. The appendix contains a list of material and web links. I highly recommend to read the electrical and DCC related standards published by the RailCommunity organization.

19 The Cab Handheld

Cab handhelds are used to control a locomotive. Depending on the other capabilities they can also configure a decoder device or set a turnout. Handhelds are connected to the bus, for example LocoNet, sometimes there is a separate bus for just the handhelds. Traditionally a cable connects the handheld to access points on the layout. Just like it is the case for base stations and boosters there is no shortage of cab handhelds. Lately, wireless handhelds have become very popular. And not to forget, some base station integrate handhelds directly in their front panel.

This chapter will describe a general handheld to just control locomotives. It directly connects via cable to the LCS bus and provides the generic elements to specify the locomotive to operate, set the speed and direction as well as the function keys. Implementing a base station and a handheld is all you would need to run an engine and finally see something for your hard work of building a layout system. The cab handheld described first is a board for developing the firmware. Nevertheless it can be used as a full functioning cab handheld. Later version will build upon the firmware but use a more handy form factor.

19.1 Requirements

A cab handheld needs to be able to control the loco. This implies that there is a local non-volatile memory that allows to remember locomotives once controlled. This way one can easily switch between a small set of locomotives and their characteristics. A display will show the actual state of cab handheld and allows together with the configuration buttons to configure the cab handheld. Looking at commercially available handhelds, they all seem to resemble TV controls. A numeric keyboard, some up and down buttons and the speed knob. (No offense). In all fairness, they are built to control not only the engines but also the rest of the layout.

But how about a cab handheld that features instead of all the functions to control an entire layout just the features to control an engine. Our cab handheld will have dedicated buttons and levers for let's say a horn or whistle, a bell, and so on. There are also configuration buttons, dedicated buttons and switches, and a very small set of buttons to map to loco specific functions. Furthermore, there is of course the rotary knob for setting the locomotive speed. The following figure shows a rough sketch of the cab handheld elements.

!Cab-Handheld-Sketch.png

Configuration and part of operation takes place with four buttons. The MENU button allows to toggle through the menus defined. To select a menu, the SELECT button is used. The menu toggle and select scheme can be nested. Within a menu screen, the MENU, UP and DOWN buttons are used screen specific and the SELECT button typically confirms the selected action. The direction switch (REV-NEUTRAL-FWD) and the SPEED knob set the speed and direction of the locomotive or consist. F1 to F4 are four general buttons that can be mapped to special functions of the particular locomotive. The Horn and Bell button are rounding up the initial design.

20 Signaling Block Control

The previous chapters got us to understand the LCS concepts, to build LCS nodes and power units on different hardware platforms. It all resulted in building an essential piece necessary for any layout, the LCS base station. So far so good. The next big part to address is to build hardware and software to control devices such as signal, turnouts, and so on. Also, we would need to get feedback from the layout where trains exactly and what they currently do. After all, a train needs to safely get from A to B. Before diving into block controllers, sensors and actors, here is a brief chapter on where the journey will go.

Safely getting a train from point A to B is the basic unit of work for a railway. In contrast to let's say a car, trains run on orders and there is a planned schedule. It is a planned movement. A train is also different to a car in that it cannot react rather quick to events such as a blocking signal. Try stopping a 15000 ton freight train half mile long. As a consequence, there is a concept of train authorization and train movement. The authorization represents the train order and the train movement gets the train safely from A to B obeying the rules of the signaling system. In the early days, there was one train moving from A to B and the authorization instance just authorized the train movement from A to B. A follow-on train moving from station A to station B had to wait for the complete route cleared over many miles as for a train moving in the opposite direction on a single line. With rising traffic the throughput was limited and new concepts were needed. One obvious solution was higher train speed and using two lines.

To increase track utilization, the real railways developed the concept of **signaling block control**. A route is divided into several blocks. A block has signals that indicate the state of the block ahead, if the block is occupied, the signal tells the following train to stop until the train ahead cleared the block. This way trains are protected from running into each other. In the early days the block signals were controlled manually by men living with their families in small houses built along the line, communicating the status of their block to their neighbor blocks by Telegraph. Trains had to carry red flags or lights on the last car so that men watching the passing train was sure that no car has been left in his block due to a broken coupler. Today automated block occupancy detectors are used.

With increased speed however, the status of the block signal has to be communicated to the engineer ahead, so he can securely rely on the block signal. In Europe this is done by a backing signal usually one kilometer ahead from the main signal. In the US the status of the block following the actual block is indicated by a multi aspect signal so that the engineer is alerted to expect a red signal after the green one right ahead. With modern state of the art systems the physical signals along the route have been replaced by the status of the line ahead directly to the engineer's cab by radio transmission.

So in general, a route from A to B is a series of blocks each governed by a signal, which shows the state of the block ahead. A train will not enter a block that is occupied by another train. Since a train movement is a planned movement, there needs to be an authorization system that locks the route and sends the train. Sounds straightforward for a single line between A and B. But consider a block with a turnout. In that case there are two follow on blocks, depending on the setting of the turnout. Inside a station or freight yard, this can get even more complex. On the other hand, a lot of safety is introduced into the operations of a layout. Typically, the track lines between stations are under a block control while a station or freight yard form one block under manual control. On a model railroad, such a block system does not only prevent trains from crashing into each other on track segments which cannot be seen by the operator in tunnels or "shadow yards", but also allows a basic "automation mode" for both DCC and DC

where you just can watch one train following the other, especially for European stile main lines with a dedicated track for each direction.

This chapter will present the signaling block concepts and gives an overview how they are implemented for the layout control system at a high level. The appendix contains further references to detailed literature about the subject. A later chapter will discuss the block controller hardware and firmware.

20.1 Requirements

As discussed already, a layout is divided into blocks, and a block further divided into subsections. Once the layout is built and configured the relation between the blocks, i.e. which blocks to enter from and to leave top are fixed. There is a static relation and a dynamic one, called routes. Consider a block with an entry and a turnout at the other end. The entry block is fixed and the two exit block too. The actual turnout setting determines which is the follow-on block. Since a block can possibly be entered from both directions, the fundamental block is a piece of track with an optional turnout at each end.

Blocks needs to be uniquely identifiable across the layout, so the configuration tool can describe the overall layout in terms of blocks as the unit of control. The block has a state. At a high level, there are free, locked and occupied blocks. Consider a train in Block A on route from block A to B to C. Block A is marked "occupied". The follow on block B will be marked "locked" and block C is "free". Once the train has left block A into block B, block A will be marked free after a short delay, block B is "occupied" and block C is "locked". Trains always look ahead two blocks, if not more.

Each block has attributes for current speed and direction of a train occupying the block. Each block further needs an attribute that specifies the entry speed when moving into this block. A speed of zero means that this block is occupied and a follow-on train cannot enter the block. If for example block C in our example is occupied, a train being in block A needs to enter block B at reduced speed. The algorithms will be described in a further section of this chapter.

A block is divided into one more subsections. While they all share the same power module, power consumption can be measured to determine in which subsection the current train is. A block should have at least two subsection if unidirectional, and three, better four, when trains can go in both directions.

A block is guarded by signals. They indicate information to the locomotive engineer about the state of the blocks ahead. The setting of the signals is a consequence of the block events broadcasted among the blocks about trains, speed and direction. Also, blocks broadcast the actual setting of entry or exit turnouts if there are any. A layout control tool needs to be able to set the turnouts as part of setting a route across several blocks.

20.2 Block Element Concept

A block is the fundamental element of the block signaling system. The following figure shows the possible configuration of a block element.

From a moving direction viewpoint, each block can be seen as having exactly one entry point and optional two exits, depending whether a turnout is par of the block or not. If there is a turn out it is typically the last subsection of the block element. A block that can be entered from both directions potentially has a turnout at each end. In theory, any possible layout can

be realized with this block element. The layout configuration and block control software will thus center around this basic element.

20.3 Turnouts

At a maximum, there are two possible turnouts for a block element. A block can have no turnouts, a turnout when entered in forward direction, and a turnout when entered in backward direction. Each turnout position, **ahead** and **turn** leads to another block. While the dependency between neighboring blocks are statically configured without turnouts (called switches in the UK), these dependency will become dynamic as soon there is a turnout at the beginning or end of a block. If there is a turnout (or group of turnouts) at the beginning of the block, the status of the block determines if the signal of the block the turnout is pointing can show red or green while the signal from the other direction have to show red by default. The same is true for a turnout at the end respectively. What the signal can display depends always on the status of the block the turnout points to.

20.4 Signals

Signals, also called semaphores in the US, communicate to the train engineer what is ahead. There are two kind of signals, the main signal at the end of a section and a distant signal associated with the main signal. The distant signals is placed in breaking distance seen from the associated main signal.

Besides the usual red for stop and green for full steam ahead there can be a additional yellow light for go ahead at lower speed indicating there will be a turnout pointing to a siding.

20.5 *ABS and APB*

With the basic elements block, subsection, turnouts and signals in place, there are several ways to implement a block control system. The first algorithms is the **Automated Block System** (**ABS**). In an ABS system the route is divided into blocks, each of which can at least hold the train length. A signal at the beginning controls whether that block can be entered at what speed. For example, a halt signal means that the train speed to enter the block is zero. The condition will is also shown with the distant signal. A distant signal is the signal ahead of the main signal that indicated the state of the main signal in breaking distance for the trains. So, a distant signal indicating a "slow" condition shows the condition of main signal ahead, which is the condition to enter the block ahead. The distance between the distance signal and the main signal is called the approach indication. Another variant is to just have main signals, one for each block entry. The approach indication in this case is the length of the block, the signal is the main signal of the block before. This variant can often be found in American railways.

So far, the signals refer to the series of blocks in one direction. Multiple trains can follow each other in one direction. If the route is to be used in both directions, signals need to be placed also in the other direction. Also the route needs to be reserved, such that two trains do not enter the route from both ends. When a train is moving from A to B, signals guarding the route B to A will all show a halt condition and are cleared on a block by block fashion according to the movement of the train from block to block. To avoid that two trains enter a line from both directions and then have to stop in the middle, a kind of reservation system is needed to manage the current route direction. This system is called **Absolute Permissive Blocks** (**APB**).

Both ABS and APB still have a fixed block length that accommodates the largest train length. But consider longer and shorter trains. Consider their speed. One approach go further traffic density would be to build a kind of moving block around the trains with a safety distance length at the rear and a breaking distance length at the front. A train following a train ahead is at least the own breaking distance and the safety distance f the train ahead away from that train. Such a system, called **Communication Based Train Control (CBTC)** requires to know the precise location and speed of each train. Also, there is a central control center that authorizes and monitors the movements of all trains. Signals are no longer necessary.

// ??? **note** this is the general concept. Can we come up with a scheme that uses less blocks ? On a train station for example there are many turnouts on entry and exit and individual lanes in between them. Perhaps the turnouts can be combined to one block at the price of blocking the whole field when a train is entering or leaving ...

20.6 Block Control Algorithms

/// ??? **note** fill in ...

20.6.1 Single Direction Line

Imagine a single direction line with several blocks. The control element is the speed at which the train can enter to block ahead. If for example the block speed is zero, the train cannot enter this block. The following tasks need to be done periodically for each block.

- If the block is occupied, the block target speed to enter the block is zero. If not, the target speed is the target block speed currently specified.
- If the target speed to enter the block is greater than zero, the next block is checked. If the target speed of the next block is lower than the target speed of this block, the target speed of this block is set to lower value of either the target speed of the next block plus an increment or the allowed maximum speed.
- The entrance signal to this block is set according to the new target speed. If the target speed is lower than the previous one, the signal is set immediately. Otherwise, set the signal is set after a short delay time.

20.6.2 Double Direction Line

// ??? **note** fill in ...

20.6.3 Absolute Permissive Blocks

/// ??? **note** fill in ...

20.7 Summary

This chapter gave a brief overview on block signaling control concepts and the key algorithms to implement. Our block signaling system will implement single direction ABS, single line ABS and APB using signals on the line. The layout will be divided into blocks with subsections. The length of the block accommodates the largest train length and subsections are granularity

CHAPTER 20. SIGNALING BLOCK CONTROL

of knowing where the trains actually is. Each block is managed by a block controller node, which manages the train hosted, associated signal and turnouts for the block. It also manages communicates the state to the other block controls and the central train authorization system. (CTC).

A LCS Nodes and EasyEda

The schematics and boards shown were all developed using the EasyED software. EasyEDA is a design tool for developing the schematics and PCB layouts. A PCB can then be ordered at very reasonable prices. Even during LCS node early design stages it is therefore sometimes worthwhile to just produce a PCB and avoid searching software bugs that are actually just loose connection on a breadboard. To ease the development, there are experimental boards. However when it comes to a final design, PCB boards need to be developed and ordered in larger quantities. The LCS Node design introduced contains a main controller board and extension boards. The sizes and location of the connectors have been standardized. This appendix contains the PCB drawings of the most common LCS boards to give you a head start in developing your own boards, ensuring that all boards fit together.

A.1 Symbols and Footprints

EasyEDA allows you to create symbols that represent components and can be placed in a schematic. To each symbol there should be a footprint that is used to put the component on to the PCB. The connection between the two is a list of assignments that associate a **pin** on the symbol with a **pad** on the footprint. For LcsNodes there is a list of symbols and footprints to ensure that the PCBs do have all their connectors at the exact place, so that they fit together.

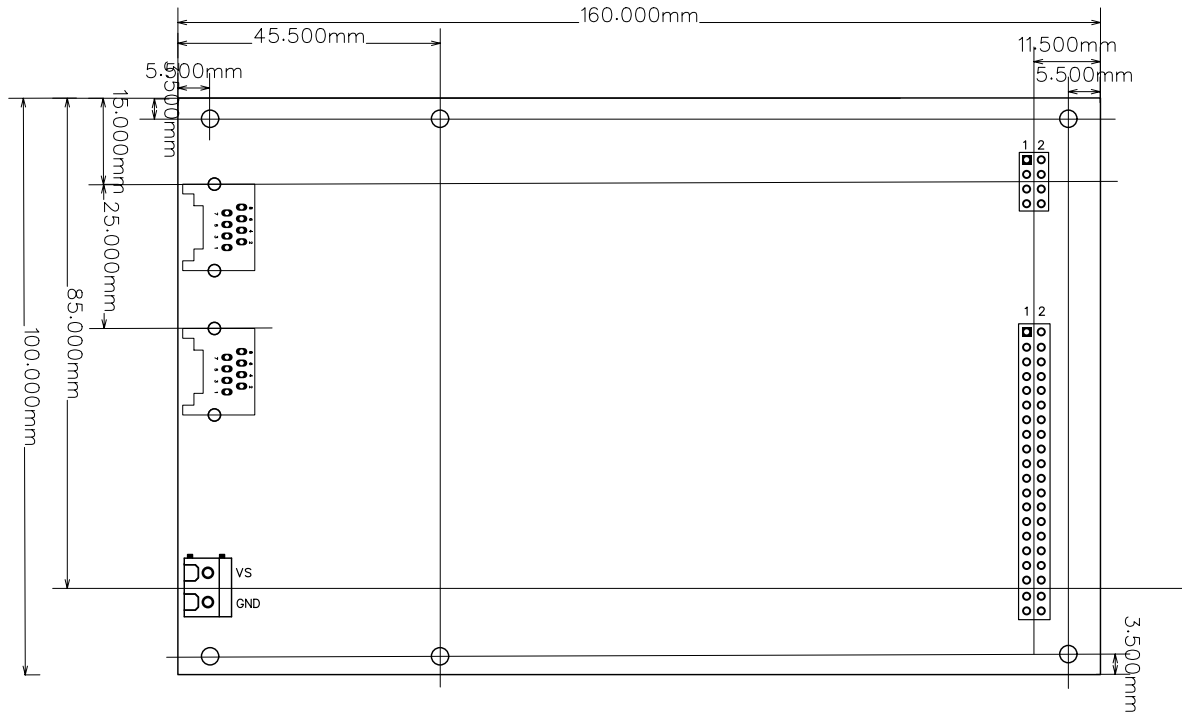
A.1.1 Symbols

To ease the development of LCS boards, the entire board and its connectors are available as a symbol. Depending on the category, the symbol features the connection end points for the connectors found on the board. This symbol is associated with the corresponding footprint described in the next section. Note that the footprint needs to match the symbol. That is the number, position and meaning of the connectors found on the board map, only length of the PCB board varies.

A.1.2 Main Controller Board Footprints

This section contains all the footprints available so far. There are three main categories. The first is anything that represents an LCS Controller portion. There are the connections to the LCS bus and the power input connector. On the left side are two connectors. The upper connector is reserved for up to four tack pow lines. Below is the LCS extension board connector. The basic LCS Main Controller Board for example is the 16cm x 10cm board shown below.

APPENDIX A. LCS NODES AND EASYEDA

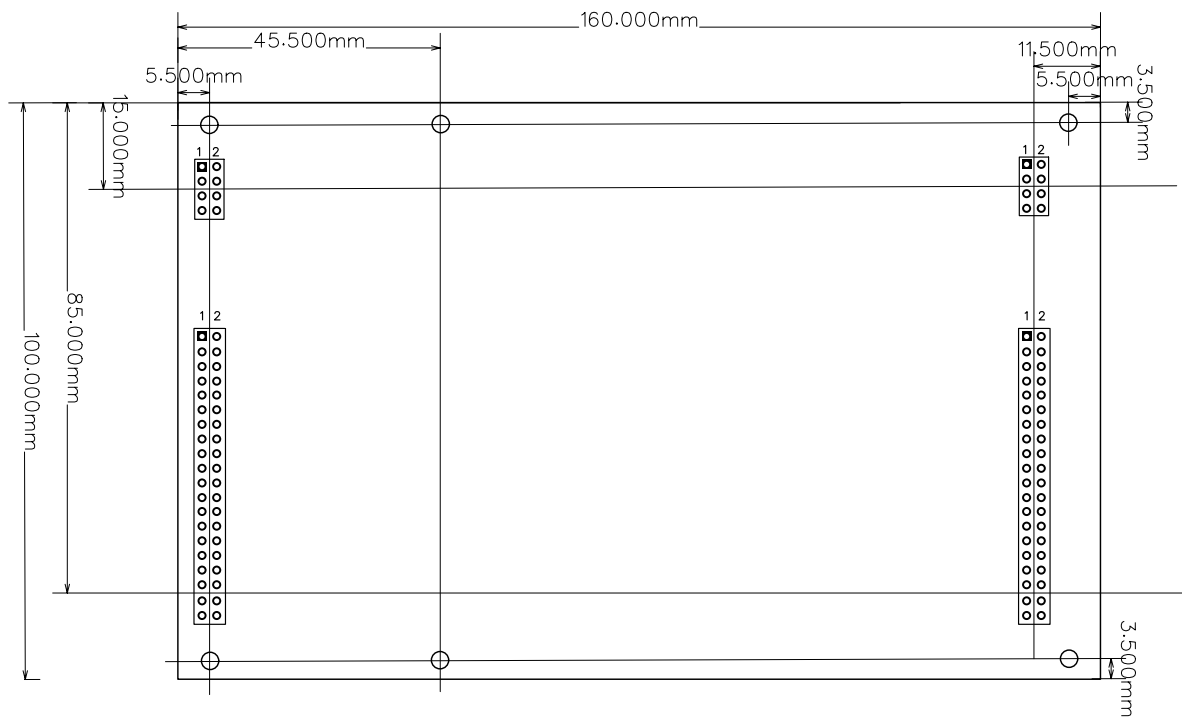
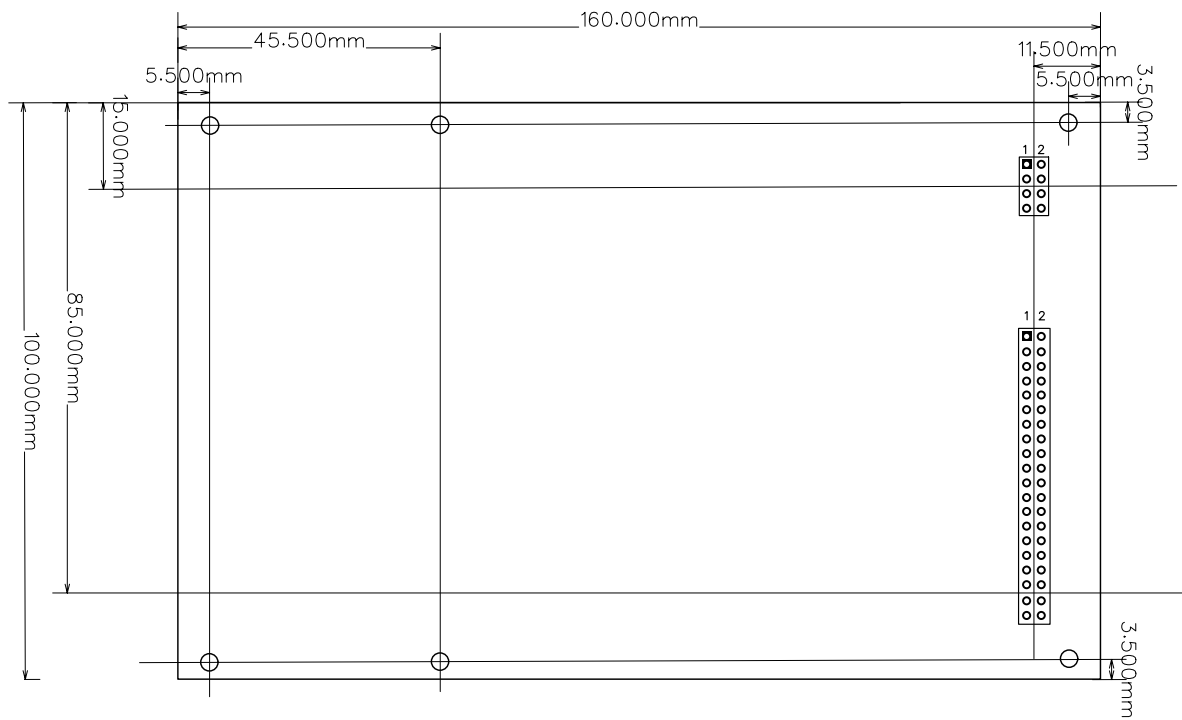


The mounting holes may look a little odd. As shown in the text to follow, there are extension boards with a form factor of 12cm x 10cm. When they are mounted on top of the 16cm board, the holes nicely match.

A.2 Extension Boards Footprints

Next, there are the extension boards. They are straightforward and just offer the two connector lines on the right and optional on the left. Just the length varies. Boards with connectors on the left and right are boards that can be just connected to a main controller board or another extension board.

APPENDIX A. LCS NODES AND EASYEDA



APPENDIX A. LCS NODES AND EASYEDA

In addition to the basic 16cm x 10cm form factor, is a set of 12cm x 10cm boards. They have exactly the same layout, except that their length is 12cm instead of 16cm.

As always, there could be many more combinations as new boards with different demands are developed. Nevertheless it is important that when connectors are used, that they have the same meaning and are placed at the same location. This is the whole idea of using footprints to ensure this exact fitting.

A.3 Links

Table A.1: ...

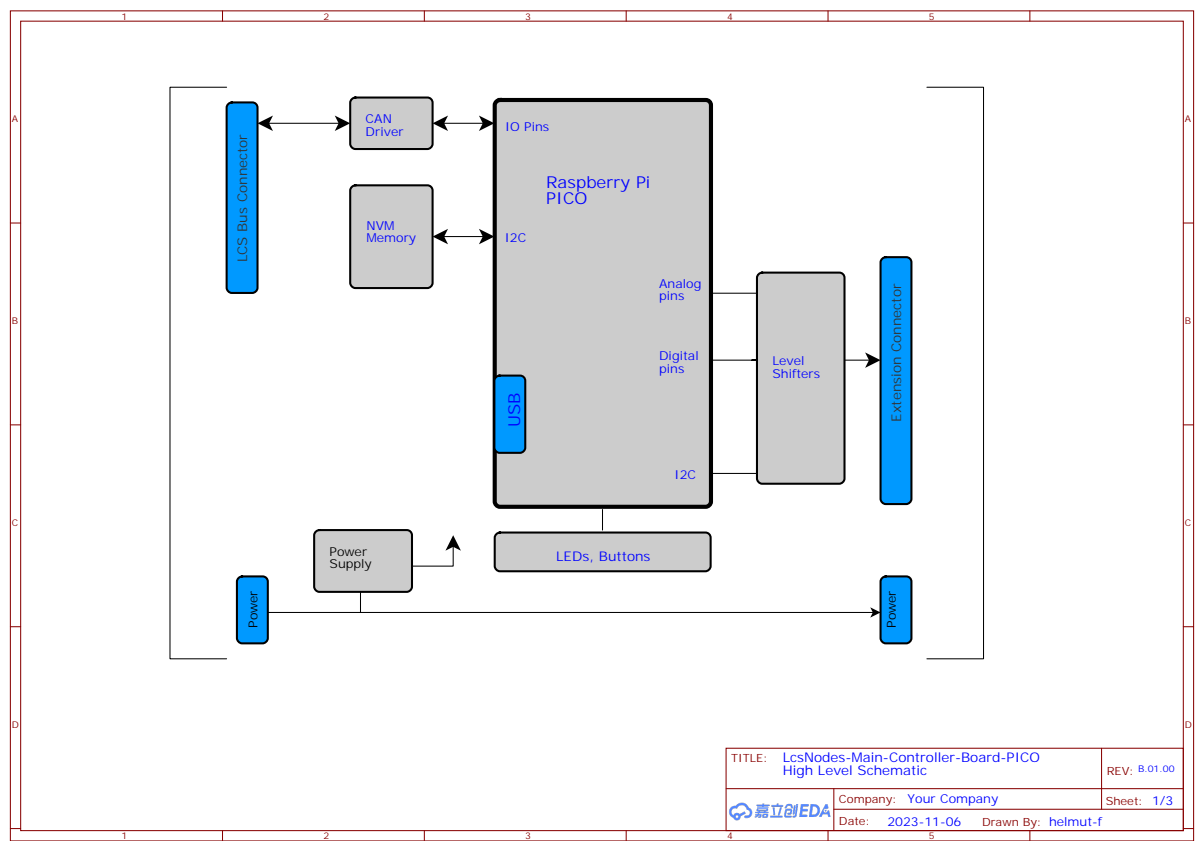
Tool	Link	Comment
EasyEDA	http://easyeda.com/de	Design tool for schematics and PCB layouts
JLCPCB	-	part of EasyEDA that manufactures PCB boards, order from within EasyEDA

B Tests

B.1 Schematics

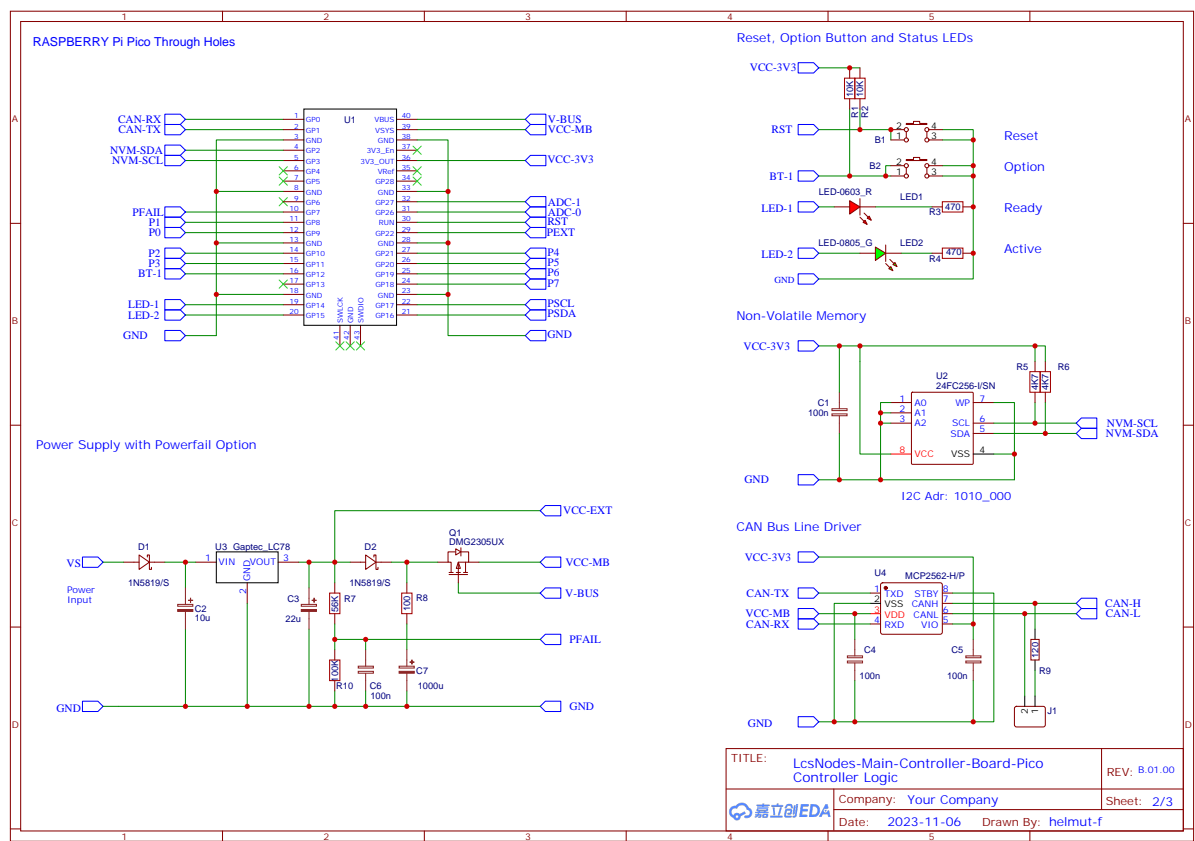
float barrier command to ensure that text stays close to the picture but no text from after the picture.

B.1.1 part 1



B.1.2 part 2

APPENDIX B. TESTS



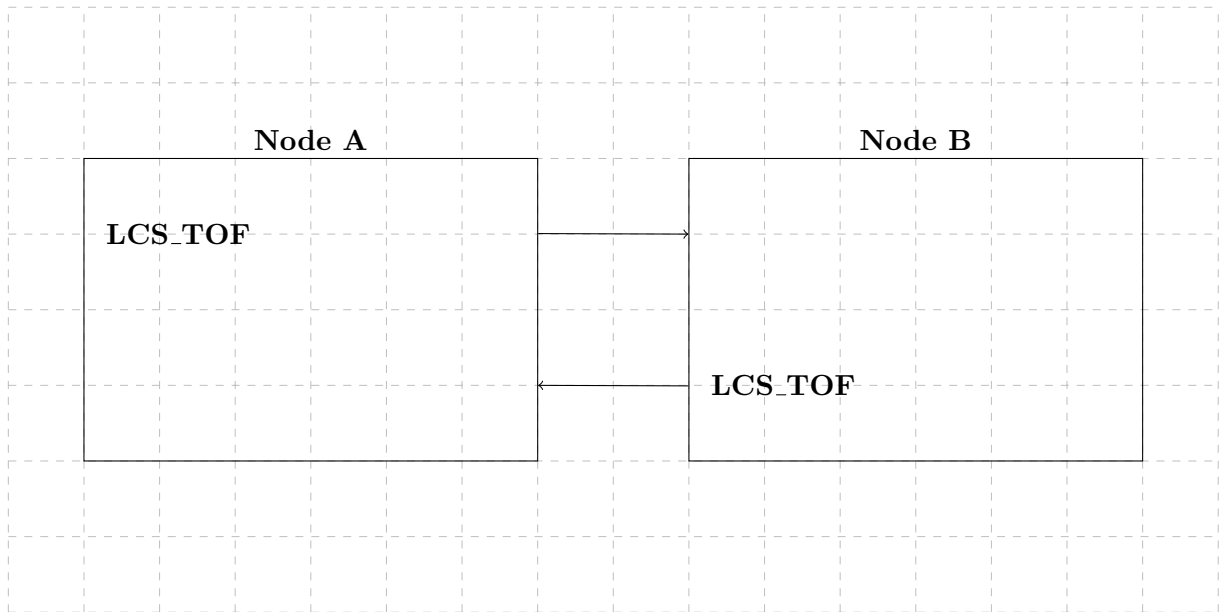
B.1.3 part 3

B.2 Pictures

A little test for a word layout ... will be a bit fiddling work ...



A bit cumbersome and we would need to have text at defined locations. Perhaps keep the simple table in the protocol chapter.



B.4 Split rectangle

We would need the split rectangle for the runtime area maps....

